

MySQL for Developers

1

Revision 2.2

PRESENTED BY
Sun Microsystems
MySQL Training Services

www.mysql.com/training

INSTRUCTOR GUIDE

© Sun Microsystems, Inc. 2009



MySQL for Developers
Skills Matrix -- Revision 2.2

Course/Chapter/Section Name	Duration (Hours)	Track	DBA	Developer	Support	Consulting
MySQL for Developers	30.5		4	5	4	4
Day One	6.25					
1. Introduction	1.75		5	5	5	5
MySQL Overview	0		5	5	5	5
MySQL Products	0.25		5	5	5	5
MySQL Services	0.25		5	5	5	5
MySQL Enterprise Services	0.25		3	3	3	3
Supported Operating Services	0		3	3	3	3
MySQL Certification Program	0		5	4	5	5
Training Curriculum Paths	0		5	5	5	5
MySQL Website	0.25		5	5	5	5
Installing MySQL	0.5		5	5	5	5
Installing the 'world' Database	0.25		5	5	5	5
2. MySQL Client/Server Concepts	0.25		5	4	5	5
MySQL General Architecture	0.25		5	5	5	5
How MySQL Uses Disk Space	0		5	3	5	5
How MySQL Uses Memory	0		5	3	5	5
3. MySQL Clients	1		4	4	3	4
Invoking Client Programs	0		5	5	5	5
Using Option Files	0.25		5	5	3	5
The MySQL Client	0.25		5	5	5	5
MySQL Query Browser	0.25		5	5	5	5
MySQL Connectors	0.25		5	4	4	4
Third-Party API's	0		4	4	3	3
4. Querying for Table Data	2		3	5	4	4
The SELECT Statement	1		3	5	4	4
Aggregating Query Results	1		3	5	4	4
Using UNION	0		3	5	4	4
5. Handling Errors and Warnings	1.25		3	5	4	4
SQL Modes	0.5		3	5	4	4
Handling Missing or Invalid Data Values	0.5		3	5	4	4
Interpreting Error Messages	0.25		3	5	4	4
Day Two	6.5					
6. SQL Expressions	1		4	5	5	5
SQL Comparisons	0.5		3	5	4	4
Functions in SQL Expressions	0.5		4	5	5	5
Comments in SQL Statements	0		4	5	5	5
7. Data Types	1.25		5	6	5	5
Data Type Overview	0		4	5	5	5
Numeric Data Types	0.5		4	5	4	4
Character String Data Types	0.25		4	5	4	4
Binary String Data Types	0.25		4	5	4	4
Temporal Data Types	0.5		4	5	4	4
NULLs	0		4	5	4	4
8. Obtaining Metadata	0.5		4	5	3	3
Metadata Access Methods	0		4	4	3	3
The INFORMATION_SCHEMA Database/Schema	0.25		4	5	3	3
Using SHOW and DESCRIBE	0		4	4	3	3
The mysqlshow Command	0.25		4	4	3	3

MySQL for Developers
Skills Matrix -- Revision 2.2

9. Databases	2		3	5	4	4
Database Properties	0.25		3	5	4	4
Good Design Practices	0.5		4	5	4	4
Identifiers	0.25		3	5	4	4
Creating Databases	0.5		3	5	4	4
Altering Databases	0.25		3	5	4	4
Dropping Databases	0.25		3	5	4	4
10. Tables	1.75		3	5	4	4
Creating Tables	0.5		3	5	5	5
Table Properties	0.25		3	5	4	4
Column Options	0.25		3	5	4	4
Creating Tables Based on Existing Tables	0.5		3	5	5	5
Altering Tables	0.25		3	5	4	4
Dropping Tables	0.25		3	5	5	5
Foreign Keys	0.25		3	5	4	4
Day Three	6.5					
11. Manipulating Table Data	1.25		3	5	5	5
The INSERT Statement	0.5		3	5	5	5
The DELETE Statement	0		3	5	5	5
The UPDATE Statement	0.25		3	5	5	5
The REPLACE Statement	0.25		3	5	5	5
INSERT with ON DUPLICATE KEY UPDATE	0.25		3	5	5	5
The TRUNCATE TABLE Statement	0		3	5	5	5
12. Transactions	1.25		4	5	5	5
What is a Transaction?	0.25		4	5	5	5
Transaction Commands	0.5		4	5	5	5
Isolation Levels	0.25		4	5	5	5
Locking	0.25		4	5	5	5
13. Joins	2.5		4	5	4	4
What is a Join?	0.5		4	5	4	4
Joining Tables in SQL	0.75		4	5	4	4
Basic Join Syntax	0.25		4	5	4	4
Inner Joins	0.25		4	5	4	4
Outer Joins	0.25		4	5	4	4
Other Types of Joins	0.25		4	5	4	4
Joins in UPDATE and DELETE statements	0.25		4	5	4	4
14. Subqueries	1.5		4	5	4	4
Types of Subqueries	0.25		4	5	4	4
Table Subquery Operators	0.5		4	5	4	4
Correlated and Non-Correlated Subqueries	0.25		4	5	4	4
Converting Subqueries to Joins	0.5		4	5	4	4
Day Four	5.75					
15. Views	1		3	5	3	3
What Are Views?	0		3	5	3	3
Creating Views	0.25		3	5	3	3
Updatable Views	0.25		3	5	3	3
Managing Views	0.25		3	4	3	3
Obtaining View Metadata	0.25		3	4	3	3
16. Prepared Statements	0.75		3	5	3	3
Why Use Prepared Statements?	0		3	4	3	3
Using Prepared Statements from the mysql Client	0.25		3	4	3	3
Preparing a Statement	0.25		3	5	3	3
Executing a Prepared Statement	0.25		3	5	3	3
Deallocating a Prepared Statement	0		3	4	3	3

MySQL for Developers
Skills Matrix -- Revision 2.2

17. Exporting and Importing Data	1		5	4	4	4
Exporting and Importing Data	0.25		5	4	4	4
Exporting and Importing Data Using SQL	0.25		5	4	4	4
Exporting and Importing Data Using MySQL Client Programs	0.25		5	4	4	4
Import Data with the SOURCE Command	0.25		5	4	4	4
18. Stored Routines	3		3	5	4	4
What is a Stored Routine?	0.25		3	5	4	4
Creating Stored Routines	0.5		3	5	4	4
Compound Statements	0.25		3	5	4	4
Assign Variables	0.25		3	5	4	4
Parameter Declarations	0		3	5	4	4
Execute Stored Routines	0.25		3	5	4	4
Stored Routine Characteristics	0.25		3	5	4	4
Examine Stored Routines	0.25		3	5	4	4
Delete Stored Routines	0.25		3	4	4	4
Flow Control Statements	0.25		3	5	4	4
Declare and Use Handlers	0.25		3	5	4	4
Cursors	0.25		3	5	4	4
Day Five	5.5					
19. Triggers	1		4	4	4	4
What are Triggers?	0.25		4	4	4	4
Delete Triggers	0.25		3	4	4	4
Restrictions on Triggers	0.5		4	4	4	4
20. Storage Engines	1.75		4	4	4	4
SQL Parser and Storage Engine Tiers	0.25		4	4	4	4
Storage Engines and MySQL	0.5		5	5	5	5
The MyISAM Storage Engine	0.25		5	5	5	5
The InnoDB Storage Engine	0.25		5	5	5	5
The MEMORY Storage Engine	0.25		5	5	5	5
Other Storage Engines	0.25		4	4	4	4
21. Optimization	2		5	5	4	4
Overview of Optimization Principles	0.25		5	5	4	4
Using Indexes for Optimization	0.5		5	5	4	4
Using EXPLAIN to Analyze Queries	0.25		5	5	4	4
Query Rewriting Techniques	0.25		5	5	4	4
Optimizing Queries by Limiting Output	0		5	5	4	4
Using Summary Tables	0.25		5	5	4	4
Optimizing Updates	0.25		5	5	4	4
Choosing Appropriate Storage Engines	0.25		5	5	4	4
22. Conclusion	0.75		4	4	3	3
Course Overview	0.25		3	3	3	3
Training and Certification Website	0		3	3	3	3
Course Evaluation	0.25		5	5	3	3
Thank You!	0		5	5	3	3
Q&A Session	0.25		5	5	3	3



MySQL for Developers – Training Guide

This is the Instructor training guide for the MySQL for Developers training course. This guide was developed using the MySQL 5.1 Community Edition - Generally Available (GA) Release.

Legal

Copyright © 2009 by Sun Microsystems, Inc.

All rights reserved. No part of this training guide shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without the written permission of Sun Microsystems. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this training guide, Sun Microsystems and the associated contributors assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

First Printing: January 2009

Trademarks

The copyright to this training guide is owned by Sun Microsystems. MySQL ® and the MySQL logo are registered trademarks of Sun Microsystems. Other trademarks and registered trademarks referred to in this manual are the property of their respective owners, and are used for identification purposes only.

Warning and Disclaimers

Every effort has been made to make this training guide as complete and accurate as possible to meet the needs of the training herein, but no warranty or fitness is implied. The information provided is on an “as is” basis.

Conventions used in this training guide

The following typographical conventions are used throughout this training guide:

- Computer input and output is printed in this format: Computer input or output. This is also used for the names of executable programs and file locations.
- Keywords from the SQL language appear in this format: **SQL KEYWORD**. SQL keywords are not case sensitive and may be written in any letter case, but the training guide uses uppercase.
- Placeholders for user input inside appearing inside computer input appear in this format: <*user input*>
- For emphasis, the following style is used: *Emphasis*
- For extra emphasis, the following style is used: ***Extra Emphasis***

When commands are shown that are meant to be executed from within a particular program, the prompt shown preceding the command indicates which command to use. For example, `shell>` indicates a command that you execute from your shell, and `mysql>` indicates a statement that you execute from the mysql client program:

```
shell> mysql -u root -h 127.0.0.1
mysql> SELECT * FROM world.City;
```

The “shell” is your command interpreter. On Linux, this is typically a program such as sh, csh, or bash. On Windows, the equivalent program is command.com or cmd.exe, typically run in a console window. When you enter a command or statement shown in an example, do not type the prompt shown in the example.

Database, table, and column names must often be substituted into statements. To indicate that such substitution is necessary, this manual uses `db_name`, `tbl_name`, and `col_name`. For example, you might see a statement like this:

```
mysql> SELECT col_name FROM db_name.tbl_name;
```

This means that if you were to enter a similar statement, you would supply your own database, table, and column names for the placeholders `db_name`, `tbl_name`, and `col_name`, perhaps like this:

```
mysql> SELECT author_name FROM biblio_db.author_list;
```

In syntax descriptions, square brackets ([and]) indicate optional words or clauses. For example, in the following statement, **IF EXISTS** is optional:

```
DROP TABLE [IF EXISTS] tbl_name;
```

When a syntax element consists of a number of alternatives, the alternatives are separated by vertical bars (pipe, |). When one member from a set of choices *may* be chosen, the alternatives are listed within square brackets ([and]):

```
TRIM( [ [ BOTH | LEADING | TRAILING ] [remstr] FROM ] str)
```

When one member from a set of choices *must* be chosen, the alternatives are listed within braces ({ and }):

```
{DESCRIBE | DESC} tbl_name [col_name | wild]
```

An ellipsis (...) indicates the omission of a section of a statement, typically to provide a shorter version of more complex syntax. For example, **INSERT ... SELECT** is shorthand for the form of **INSERT** statement that is followed by a **SELECT** statement.

An ellipsis can also indicate that the preceding syntax element of a statement may be repeated. In the following example, multiple `reset_option` values may be given, with each of those after the first preceded by commas:

```
RESET reset_option[, reset_option] ...
```

Commands for setting shell variables are shown using Bourne shell syntax. For example, the sequence to set the CC environment variable and run the configure command looks like this in Bourne shell syntax:

```
shell> CC=gcc ./configure
```

If you are using csh or tcsh, you must issue commands somewhat differently:

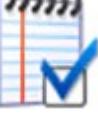
```
shell> setenv CC gcc
```

and

```
shell> ./configure
```

Supporting images used in this Training Guide

The following is a summary of the standard images used in this manual to support the instruction:

IMAGE	NAME	DESCRIPTION
	Preparation	This image is used to describe the steps required to be completed prior to performing a hands-on exercise.
	Written exam	This image is used to identify that the student is going to be tested upon the material previously presented in the instructional material
	Inline Lab	Throughout the course the instructor will conduct labs in line with the instruction, which are designed to help you to understand the “nuts and bolts” (inner-workings) of the topic.
	Further Practice Lab	This image is used to convey to the student that there is a final exercise to complete prior to the completion of the chapter.
	Student notes	This image identifies an area on a page designated for students to write notes associated with the class.
	Slide number box	Indicates the number of an existing slide that corresponds to the text.

Instructor Note: Boxes like this one are used throughout the course manual to give instructors tips on how to deliver the material and to give them extra information that is not included in the student manual. Please be advised that instructors are requested to cover *all* the material in this manual, as prompted by the slides. Although some slides will mimic the manual exactly, **most will not**. In those cases, instructors will need to use the manual, along with these notes to deliver the required information to the students. These note boxes will always be placed at the bottom of the page, and are *only* in the instructor guide.

Acknowledgments

Sun Microsystems would like to thank the many individuals that played a part in bringing this training material to the numerous students who will benefit from the knowledge and effort that each of these contributors put into the training. Even though there were a large number of contributions from many Sun Microsystems' employees, the following list of contributors played a vital role in developing this material and ensuring that its contents were accurate, timely and most of all presented in a way that would benefit those that are utilizing it for the benefit of improving their skills with MySQL.

- **Max Mether**, Course Development Manager
- **Sarah Sproehnle**, Subject Matter Expert
- **Kai Voigt**, Subject Matter Expert
- **Dave Stokes**, Certification Consultant
- **Roland Bouman**, Course Developer
- **Jeffrey Gorton**, Course Developer and Editor



2

Class Delivery Preparation

There are a few steps taken at the beginning of class to prepare the students and instructor for the class:

3

- Getting to know the students and their expectations
- Checklist of important logistical information
- Explanation of the format and usage of the slides and student guide

4



5

Course Objectives

This instructor-led course is designed for students planning on developing applications that make use of MySQL. This course covers essential SQL statements for data design, querying, and programming. In addition, it will prepare students for the MySQL Developer certification.

- Describe the MySQL client/server architecture
- Invoke MySQL client programs
- Describe the MySQL connectors that provide connectivity for client programs
- Select the best data type for table data
- Manage the structure of your databases
- Use the SELECT statement to retrieve information from database tables
- Use expressions in SQL statements to retrieve more detailed information
- Utilize SQL statements to modify the contents of database tables
- Write multiple table queries
- Use nested queries in your SQL statements
- Transactions
- Create "virtual tables" of specific data
- Perform bulk data import and export operations
- Create user defined variables, prepared statements and stored routines
- Create and manage triggers
- Acquiring metadata
- Debug MySQL applications
- Use of available storage engines
- Optimizing Queries

6



Table of Contents

1 Introduction.....	1-1
1.1 Learning Objectives.....	1-2
1.2 MySQL Overview.....	1-3
1.2.1 Sun Acquisition.....	1-3
1.2.2 MySQL Partners.....	1-4
1.3 MySQL Products.....	1-5
1.3.1 MySQL Database Products.....	1-5
1.3.2 MySQL GUI Tools.....	1-5
1.3.3 Other MySQL Tools.....	1-6
1.3.4 MySQL Drivers.....	1-6
1.3.5 Solutions for Embedding MySQL.....	1-7
1.4 MySQL Services.....	1-8
1.4.1 MySQL Training.....	1-8
1.4.2 MySQL Certification.....	1-8
1.4.3 MySQL Consulting.....	1-8
1.4.4 MySQL Support.....	1-8
1.5 The MySQL Enterprise Subscription.....	1-10
1.5.1 MySQL Enterprise Server.....	1-10
1.5.2 24x7 Production Support.....	1-10
1.5.3 MySQL Enterprise Monitor.....	1-11
1.5.4 Obtaining a MySQL Enterprise Subscription.....	1-14
1.6 Supported Operating Systems.....	1-16
1.7 MySQL Certification Program.....	1-17
1.8 Training Curriculum Paths.....	1-18
1.9 MySQL Website.....	1-22
1.9.1 MySQL Community Web Page.....	1-23
1.9.2 MySQL Online Documentation.....	1-24
1.10 Installing MySQL.....	1-27
1.11 Installing the 'world' database.....	1-28
1.12 Chapter Summary.....	1-29
2 MySQL Client/Server Concepts.....	2-1
2.1 Learning Objectives.....	2-2
2.2 MySQL General Architecture.....	2-3
2.2.1 MySQL Server	2-3
2.2.2 Client Programs.....	2-3
2.2.3 Communication Protocols.....	2-4
2.2.4 MySQL Non-Client Utilities	2-5
2.3 How MySQL Uses Disk Space.....	2-6
2.3.1 How MySQL Uses Memory.....	2-6
2.4 Chapter Summary.....	2-8
3 MySQL Clients.....	3-1
3.1 Learning Objectives.....	3-2
3.2 Invoking Client Programs.....	3-3
3.2.1 General Command Option Syntax	3-3
3.2.2 Connection Parameter Options	3-4
3.3 Using Option Files.....	3-10

3.4 The mysql command line client.....	3-13
3.4.1 Using mysql interactively.....	3-13
3.4.2 Statement Terminators	3-13
3.4.3 The mysql Prompts	3-15
3.4.4 Using Editing Keys in mysql	3-16
3.4.5 Using Script Files with mysql.....	3-20
3.4.6 MySQL Output Formats.....	3-21
3.4.7 Client Commands and SQL Statements.....	3-22
3.4.8 Using Server-Side Help	3-22
3.4.9 Using the Safe Updates Option	3-24
3.5 MySQL Connectors.....	3-28
3.5.1 Native C.....	3-28
3.5.2 MySQL Connector/ODBC	3-28
3.5.3 MySQL Connector/J.....	3-29
3.5.4 MySQL Connector/.NET.....	3-29
3.5.5 MySQL Connector/MXJ	3-29
3.6 Third-Party API's.....	3-30
3.7 Chapter Summary.....	3-32
4 Querying for table Data.....	4-1
4.1 Learning Objectives.....	4-2
4.2 The SELECT Statement.....	4-3
4.2.1 Basic Uses of SELECT.....	4-4
4.2.2 Using FROM.....	4-5
4.2.3 Using DISTINCT.....	4-6
4.2.4 Using WHERE.....	4-8
4.2.5 Using ORDER BY in SELECT statements.....	4-12
4.3 Aggregating Query Results.....	4-19
4.3.1 Why Use Aggregate Functions?.....	4-19
4.3.2 Grouping with SELECT and GROUP BY.....	4-20
4.4 Using UNION	4-26
4.5 Chapter Summary.....	4-31
5 Handling Errors and Warnings.....	5-1
5.1 Learning Objectives.....	5-2
5.2 SQL modes affecting syntax.....	5-3
5.2.1 Setting the SQL Mode.....	5-3
5.3 Handling Missing or Invalid Data Values.....	5-6
5.3.1 Handling Missing Values.....	5-6
5.3.2 Handling invalid values in non-strict mode.....	5-7
5.3.3 Handling Invalid Values in Strict Mode.....	5-9
5.3.4 Additional Input Data Restrictions.....	5-9
5.3.5 Interpreting Error Messages.....	5-10
5.3.6 The SHOW WARNINGS Statement	5-11
5.3.7 The SHOW ERRORS Statement.....	5-14
5.3.8 The perror Utility.....	5-14
5.4 Chapter Summary.....	5-19

6 Data Types.....	6-1
6.1 Learning Objectives.....	6-2
6.2 Data Type Overview.....	6-3
6.2.1 The ABC's of Data Types.....	6-3
6.2.2 Creating Tables with Data Types.....	6-3
6.3 Numeric Data Types.....	6-4
6.3.1 Integer Types.....	6-4
6.3.2 Floating-Point Types.....	6-5
6.3.3 Fixed-Point Types.....	6-7
6.3.4 BIT Types.....	6-8
6.4 Character String Data Types.....	6-13
6.4.1 Unstructured Character String Data Types.....	6-13
6.4.2 Text Type Comparison.....	6-15
6.4.3 Structured Character String types.....	6-16
6.4.4 Character Set and Collation Support.....	6-17
6.5 Binary String Data Types.....	6-25
6.5.1 Binary Types.....	6-25
6.5.2 Binary String Type Comparison.....	6-25
6.6 Temporal Data Types.....	6-27
6.6.1 Temporal Data Types.....	6-27
6.7 NULLs.....	6-32
6.7.1 The Meaning of NULL.....	6-32
6.7.2 When to Use NULL.....	6-32
6.7.3 When NOT to Use NULL.....	6-33
6.8 Chapter Summary.....	6-37
7 SQL Expressions.....	7-1
7.1 Learning Objectives.....	7-2
7.2 SQL Comparisons.....	7-3
7.2.1 Components of SQL Expressions.....	7-3
7.2.2 Numeric Expressions.....	7-4
7.2.3 String Expressions.....	7-5
7.2.4 Temporal Expressions.....	7-15
7.3 Functions in SQL Expressions.....	7-20
7.3.1 Comparison Functions.....	7-20
7.3.2 Control Flow Functions.....	7-21
7.3.3 Numeric Functions.....	7-25
7.3.4 String Functions.....	7-28
7.3.5 Temporal Functions.....	7-34
7.3.6 NULL-Related Functions.....	7-38
7.3.7 Comments in SQL Statements.....	7-41
7.3.8 Comments on database objects.....	7-43
7.4 Chapter Summary.....	7-46
8 Obtaining Metadata.....	8-1
8.1 Learning Objectives.....	8-2
8.2 Metadata Access Methods.....	8-3
8.3 The INFORMATION_SCHEMA database.....	8-4
8.3.1 INFORMATION_SCHEMA Tables.....	8-4
8.3.2 Displaying INFORMATION_SCHEMA Tables.....	8-5

8.4 Using SHOW and DESCRIBE	8-11
8.4.1 SHOW Statements.....	8-11
8.4.2 DESCRIBE Statements.....	8-14
8.5 The mysqlshow Client Program.....	8-19
8.6 Chapter Summary.....	8-23
9 Databases.....	9-1
9.1 Learning Objectives.....	9-2
9.2 Database Properties.....	9-3
9.3 Design Practices.....	9-4
9.3.1 Keys.....	9-4
9.3.2 Common Diagramming Systems.....	9-5
9.3.3 Normalization.....	9-7
9.4 Identifiers.....	9-12
9.4.1 Identifier Syntax.....	9-12
9.4.2 Reserved Words as Identifiers.....	9-12
9.4.3 Using Qualified Names.....	9-13
9.4.4 Case Sensitivity.....	9-14
9.5 Creating Databases.....	9-15
9.6 Altering Databases.....	9-17
9.7 Dropping Databases.....	9-20
9.7.1 CAUTION: When Using DROP DATABASE.....	9-20
9.8 Chapter Summary.....	9-23
10 Tables.....	10-1
10.1 Learning Objectives.....	10-2
10.2 Creating Tables.....	10-3
10.2.1 Table Properties.....	10-4
10.2.2 Column Options.....	10-5
10.2.3 Constraints.....	10-6
10.2.4 SHOW CREATE TABLE.....	10-8
10.2.5 Creating Tables Based on Existing Tables.....	10-11
10.2.6 Temporary Tables.....	10-16
10.3 Altering Tables.....	10-18
10.3.1 Add Columns.....	10-18
10.3.2 Remove Columns.....	10-18
10.3.3 Altering column definitions.....	10-19
10.3.4 Changing Columns.....	10-22
10.3.5 Renaming Tables.....	10-22
10.4 Dropping Tables.....	10-25
10.5 Foreign Key Constraints.....	10-27
10.5.1 Foreign keys and relationships.....	10-27
10.5.2 Foreign Key Constraints and Referential Integrity.....	10-29
10.5.3 Foreign Key Constraints and MySQL Storage Engines.....	10-32
10.6 Chapter Summary.....	10-41
11 Manipulating Table Data.....	11-1
11.1 Learning Objectives.....	11-2
11.2 The INSERT Statement.....	11-3
11.2.1 INSERT ... VALUES Syntax.....	11-3
11.2.2 INSERT ... SET Syntax.....	11-3
11.2.3 INSERT ... SELECT Syntax.....	11-4
11.2.4 INSERT with LAST_INSERT_ID.....	11-4
11.3 The DELETE Statement.....	11-8

11.3.1 Using DELETE with ORDER BY and LIMIT.....	11-8
11.4 The UPDATE Statement.....	11-11
11.5 The REPLACE Statement.....	11-15
11.6 INSERT with ON DUPLICATE KEY UPDATE.....	11-17
11.7 The TRUNCATE TABLE Statement.....	11-20
11.8 Chapter Summary.....	11-23
12 Transactions.....	12-1
12.1 Learning Objectives.....	12-2
12.2 What is a Transaction?.....	12-3
12.2.1 ACID.....	12-4
12.3 Transaction Control Statements.....	12-5
12.3.1 The autocommit mode.....	12-5
12.3.2 Statements causing an Implicit COMMIT.....	12-6
12.3.3 Finding a Storage Engine that supports transactions.....	12-7
12.4 Isolation Levels.....	12-12
12.4.1 Consistency issues.....	12-12
12.4.2 Four Levels.....	12-13
12.5 Locking.....	12-18
12.5.1 Locking Concepts.....	12-18
12.5.2 Locking Reads.....	12-18
12.6 Chapter Summary.....	12-23
13 Joins.....	13-1
13.1 Learning Objectives.....	13-2
13.2 What is a Join?.....	13-3
13.2.1 The limitation of single table queries.....	13-3
13.2.2 Combining two simple tables.....	13-4
13.2.3 Cartesian product.....	13-5
13.2.4 General properties of the Cartesian product.....	13-7
13.2.5 Filtering out undesired rows.....	13-10
13.2.6 Columns.....	13-11
13.2.7 Joins and foreign keys.....	13-11
13.3 Joining tables in SQL.....	13-16
13.3.1 SQL and the Cartesian product.....	13-16
13.3.2 Using a WHERE clause for the join condition.....	13-16
13.3.3 Qualifying ambiguous column names.....	13-20
13.3.4 Using table aliases.....	13-21
13.4 Basic JOIN syntax.....	13-24
13.4.1 Non-join conditions in the ON clause.....	13-25
13.5 Inner joins.....	13-27
13.5.1 Inner join pseudocode.....	13-27
13.5.2 Omitting the join condition.....	13-28
13.6 Outer joins.....	13-30
13.6.1 Again: Two simple tables.....	13-30
13.6.2 Retaining rows even when no match is found.....	13-32
13.6.3 Left outer join.....	13-33
13.6.4 Right join.....	13-34
13.6.5 The join condition of outer join operations.....	13-35
13.6.6 Choosing between inner and outer joins.....	13-38

13.7 Other types of joins.....	13-44
13.8 Joins in UPDATE and DELETE statements.....	13-49
13.8.1 Multi-table UPDATE syntax.....	13-49
13.8.2 Multi-table DELETE syntax.....	13-50
13.8.3 Using multi-table UPDATE and DELETE statements?.....	13-52
13.9 Chapter Summary.....	13-57
14 Subqueries.....	14-1
14.1 Learning Objectives.....	14-2
14.2 Overview.....	14-3
14.3 Types of subqueries.....	14-5
14.3.1 Scalar subqueries.....	14-5
14.3.2 Row subqueries.....	14-6
14.3.3 Table subqueries.....	14-8
14.4 Table subquery operators.....	14-10
14.4.1 The IN operator.....	14-10
14.4.2 The EXISTS Operator.....	14-11
14.4.3 ALL, ANY and SOME.....	14-12
14.5 Correlated and non-Correlated Subqueries.....	14-17
14.5.1 Non-Correlated Subqueries.....	14-17
14.5.2 Correlated Subqueries.....	14-17
14.5.3 Other Subquery Uses.....	14-23
14.6 Converting Subqueries to Joins.....	14-24
14.6.1 Rewriting IN and NOT IN.....	14-24
14.6.2 Limitations to Rewriting Subqueries to Joins	14-26
14.7 Chapter Summary.....	14-31
15 Prepared Statements.....	15-1
15.1 Learning Objectives.....	15-2
15.2 Why Use Prepared Statements?.....	15-3
15.3 Using Prepared Statements from the mysql Client.....	15-4
15.3.1 User Defined Variables:.....	15-4
15.4 Preparing a Statement.....	15-6
15.5 Executing a Prepared Statement.....	15-8
15.6 Deallocation a Prepared Statement.....	15-10
15.7 Chapter Summary.....	15-12
16 Prepared Statements.....	16-1
16.1 Learning Objectives.....	16-2
16.2 Why Use Prepared Statements?.....	16-3
16.3 Using Prepared Statements from the mysql Client.....	16-4
16.3.1 User Defined Variables:.....	16-4
16.4 Preparing a Statement.....	16-6
16.5 Executing a Prepared Statement.....	16-8
16.6 Deallocation a Prepared Statement.....	16-10
16.7 Chapter Summary.....	16-12
17 Exporting and Importing Data.....	17-1
17.1 Learning Objectives.....	17-2
17.2 Export and Import Data Using SQL.....	17-3
17.2.1 Export Data Using SELECT with INTO OUTFILE.....	17-3
17.2.2 Importing Data Using LOAD DATA INFILE.....	17-8

17.3 Export and Import Using MySQL Client Programs.....	17-13
17.3.1 Export Data with ‘mysqldump’.....	17-13
17.3.2 Import Data with mysqlimport.....	17-14
17.3.3 Import Data with the SOURCE Command.....	17-16
17.4 Chapter Summary.....	17-18
18 Stored Routines.....	18-1
18.1 Learning Objectives.....	18-2
18.2 What is a Stored Routine?.....	18-3
18.3 Creating Stored Routines.....	18-5
18.3.1 Creating Procedures.....	18-5
18.3.2 Creating Functions.....	18-5
18.4 Compound Statements.....	18-7
18.5 Assign Variables.....	18-9
18.6 Parameter Declarations.....	18-11
18.7 Execute Stored Routines.....	18-15
18.8 Examine Stored Routines.....	18-16
18.9 Delete Stored Routines.....	18-19
18.10 Flow Control Statements.....	18-22
18.11 Declare and Use Handlers.....	18-29
18.12 Cursors.....	18-33
18.13 Chapter Summary.....	18-38
19 Triggers.....	19-1
19.1 Learning Objectives.....	19-2
19.2 What are Triggers?.....	19-3
19.2.1 Creating Triggers.....	19-3
19.2.2 Triggers Events.....	19-4
19.2.3 Trigger Error Handling.....	19-6
19.3 Delete Triggers.....	19-7
19.4 Restrictions on Triggers.....	19-8
19.5 Chapter Summary.....	19-14
20 Storage Engines.....	20-1
20.1 Learning Objectives.....	20-2
20.2 SQL Parser and Storage Engine Tiers.....	20-3
20.2.1 Storage Engine Breakdown	20-3
20.3 Storage Engines and MySQL.....	20-5
20.3.1 Available Storage Engines.....	20-5
20.3.2 The Most Common Storage Engines.....	20-5
20.3.3 View Available Storage Engines.....	20-6
20.3.4 Setting the Storage Engine.....	20-6
20.3.5 Displaying Storage Engine Information.....	20-6
20.4 The MyISAM Storage Engine.....	20-10
20.4.1 MyISAM Row Storage Formats.....	20-10
20.4.2 Compressing MyISAM Tables.....	20-11
20.4.3 MyISAM Locking.....	20-12
20.5 The InnoDB Storage Engine.....	20-16
20.5.1 The InnoDB Tablespace and Logs.....	20-16
20.5.2 InnoDB and ACID Compliance.....	20-17
20.5.3 InnoDB Locking.....	20-17

20.6 The MEMORY Storage Engine.....	20-19
20.6.1 MEMORY Indexing Options.....	20-19
20.6.2 Storage Engine Summary.....	20-19
20.7 Other Storage Engines.....	20-20
20.8 Chapter Summary.....	20-24
21 Optimization.....	21-1
21.1 Learning Objectives.....	21-2
21.2 Overview of Optimization Principles.....	21-3
21.3 Using Indexes for Optimization.....	21-4
21.3.1 Types of Indexes.....	21-4
21.3.2 Creating Indexes.....	21-5
21.3.3 Adding Indexes to Existing Tables.....	21-8
21.3.4 Dropping Indexes.....	21-11
21.3.5 Principles for Index Creation.....	21-14
21.3.6 Indexing Column Prefixes.....	21-15
21.3.7 Leftmost Index Prefixes.....	21-16
21.3.8 FULLTEXT Indexes.....	21-17
21.4 Using EXPLAIN to Analyze Queries.....	21-21
21.4.1 How EXPLAIN Works.....	21-21
21.4.2 EXPLAIN Output Columns.....	21-22
21.5 Query Rewriting Techniques.....	21-27
21.6 Optimizing Queries by Limiting Output.....	21-29
21.7 Using Summary Tables.....	21-30
21.8 Optimizing Updates.....	21-35
21.9 Choosing Appropriate Storage Engines.....	21-36
21.10 Chapter Summary.....	21-38
22 Conclusion.....	22-1
22.1 Learning Objectives.....	22-2
22.2 Training and Certification Website.....	22-3
22.3 Course Evaluation.....	22-5
22.4 Thank you!.....	22-6
22.5 Q&A Session.....	22-7
Appendix A.....	A-1
Appendix B.....	B-1

1 INTRODUCTION

Instructor note:

The purpose of the first chapter is to...

- Get to know MySQL as a company
- Learn a bit about the Sun acquisition
- Get to know how to participate in the MySQL community

8

1.1 Learning Objectives

This chapter introduces you to MySQL and the support organization for MySQL. Upon completion of this chapter you will be able to:

- Explain the origin and status of the MySQL product
- List the available MySQL products and professional services
- Describe the MySQL Enterprise subscription
- List the currently supported operating systems
- Describe the MySQL Community web page
- Describe the MySQL Certification program
- List all the available MySQL courses



1.2 MySQL Overview

9

MySQL is a Relational Database Management System (RDBMS) that was originally developed by ‘MySQL AB’. It was (and continues to be) developed and marketed as a family of high performance, affordable database servers and tools. Contributing to building the mission-critical, high-volume systems and products worldwide is what makes MySQL the world’s most popular open source database, as well as its reliability, excellent performance and ease of use.

MySQL is not only the world’s most popular open source database. It is also the fastest growing database in the industry, with over 70,000 downloads per day. Ranging from large corporations to specialized embedded applications.

MySQL AB was founded in Sweden by two Swedes and a Finn: David Axmark, Allan Larsson and Michael "Monty" Widenius who have worked together since the 80’s. MySQL AB(Swedish for “Inc.”) was the sole owner of the MySQL server source code, the MySQL trademark and the mysql.com domain worldwide.



**MySQL is installed on every continent in the world
(Yes, even Antarctica!)**

10

In early 2008, MySQL AB was acquired by Sun Microsystems, Inc., giving the MySQL product line the considerable resources of a major mainstream company. Together, Sun and MySQL will now provide global enterprise-class support 24x7x365. Our enterprise-class customers worldwide can now take advantage of our market-leading open source

database on their choice of hardware, operating system and language with up to a 90% lower total cost of ownership than many traditional database solutions.

MySQL has always been, and will continue to be, a strong supporter of the open source philosophy and the open source community, and strives to work with partners who share the same values and mindset.

This will be no different within Sun. Sun is a world-leader in open source, with products and technologies such as Java, OpenSolaris, Open Office, NetBeans, OpenSparc and many more.

The MySQL mission is still the same...



Make superior database software available and affordable to all!



1.2.2 MySQL Partners

11

MySQL has had the privilege of forming alliances with excellent partners and attracting some of the most impressive customers in the industry! When you join our ranks, you are joining a winning team with a wide variety of MySQL implementations. It never ceases to amaze us, the innovative and powerful ways in which our tools are being used. To name only a few;

Web / Web 2.0

OEM / ISV's

On Demand, SaaS, Hosting

Telecommunications

Enterprise 2.0

Open-source is powering the World!

"We have used MySQL far more than anyone expected. We went from experimental to mission-critical in a couple of months."

Jeremy Zawodny--MySQL Database Expert Yahoo! Finance



Instructor Notes: For more information on our Partners and Solutions go to our website:

<http://solutions.mysql.com/>

1.3 MySQL Products

1.3.1 MySQL Database Products

12

Sun provides database products to meet the needs of ISV/OEM, Enterprise, and Community Server users. MySQL database products are recognized for superior ease of use, performance, and reliability.

- **MySQL Enterprise Server**

The most reliable, secure and up-to-date version of the world's most popular open source database for cost-effectively delivering E-commerce, Online Transaction Processing (OLTP), and multi-terabyte Data Warehousing applications. (Available only with the MySQL Enterprise subscription).

- **MySQL Community Server**

The MySQL database server for open source developers and technology enthusiasts who want to get started with MySQL. Supported by the large MySQL open source community. Under the General Public License (GPL), benefits to the open source community include a commercial-grade framework that is free of charge.



- **MySQL Embedded Database**

The most popular choice for OEMs/ISVs who want to cost-effectively embed or bundle a reliable and high-performance relational database.

- **MySQL Cluster**

A fault tolerant database clustering architecture for deploying highly available mission-critical database applications.

13

1.3.2 MySQL GUI Tools

The MySQL GUI Tools form a comprehensive graphical user interface to your MySQL database. These easy to use graphical tools enable database Developers and Database Administrators (DBAs) to be more productive.



- **MySQL Migration Toolkit**

Using a wizard-driven interface, the MySQL Migration Toolkit implements a proven methodology and walks you through the necessary steps to successfully complete a database migration project.

- **MySQL Administrator**

A powerful graphical administration console that enables you to easily administer your MySQL environment and gain significantly better visibility into how your databases are operating.

- **MySQL Query Browser**

An extremely user-friendly graphical tool for creating, executing, and optimizing SQL queries for your MySQL Database Server.

The MySQL graphical user interface (GUI) tools discussed here come bundled together when downloaded from the MySQL website.

Instructor Notes: The above **GUI tools** are currently being minimally maintained as of the 5.0-r13 release (October 2008). There are no plans to improve these tools in their current state. Due to limited resources we also can only do this maintenance for the **Windows** part of the GUI tools. However, we are planning to move the entire code base of the legacy GUI tools to Launchpad in early 2009, so it should become much easier for external contributors to help out with patches or to create a new release for Linux and Mac OS X.

The plan is to move their features step-by-step into **MySQL Workbench** to have one central location for design and management. Until this integration is done we will bring out a new release of the legacy tools at least once per month, providing our users with bug fixes and a few sensitive improvements.



14

1.3.3 Other MySQL Tools

MySQL has also developed additional tools to assist in the use of the MySQL server:

- **MySQL Workbench**

MySQL Workbench is a visual, database design tool developed by MySQL. It enables model-driven database design and maintenance, automates time-consuming and error-prone tasks, and improves communication among DBA and developer teams. MySQL Workbench is offered in the following editions:

- **Community Edition** — available under the open source GPL license.
- **Standard Edition** — requires purchase of annual subscription. Includes additional modules and plugins to improve DBA productivity.

- **MySQL Proxy**

MySQL Proxy is a simple program that sits between your client and MySQL server(s) that can monitor, analyze or transform their communication. Its flexibility allows for unlimited uses. Some of the more common uses include; load balancing, failover, query analysis, query filtering and modification.

1.3.4 MySQL Drivers

15

MySQL database drivers (also known as connectors), provide database client connectivity for a wide range of programming languages. MySQL includes the following connectors:

- **MySQL C API**

Our native client library ([libmysql](#)), which can be wrapped by other languages.

- **MySQL Connector/ODBC**

Connect to a MySQL database server using the Open Database Connectivity (ODBC) API on all Microsoft Windows and most Unix-like platforms. The ODBC driver builds on the client/server protocol implementation provided by [libmysql](#).

- **MySQL Connector/J**

A JDBC (Java Database Connectivity) 4.0 driver for Java 1.4 and higher. Provides a java native implementation of the MySQL client/server protocol.

- **MySQL Connector/Net**

A fully managed ADO.NET provider for the .NET framework (version 1.1 and 2.0). Provides a .NET implementation of the MySQL client/server protocol.

- **MySQL Connector/PHP**

Provides MySQL connectivity for PHP programs. Currently there are two MySQL specific PHP extensions available that use [libmysql](#): the [mysql](#) and [mysqli](#) extensions. There is also MySQL support for the generic PHP Database objects (PDO) extension. In addition there is the PHP native driver called [mysqlnd](#) which can replace [libmysql](#) in the [mysqli](#) extension.



Instructor Notes: Connectors development is a work in progress. In addition to these generally available connectors, there are alpha preview implementations for Connector/OpenOffice (which provides native database connectivity from within the Open Office suite) and a PHP Database Objects extension based on [mysqlnd](#).

Additionally, the third-party C++ and PERL connectors are also widely used.

16

1.3.5 Solutions for Embedding MySQL

Apart from these Connectors that provide client side connectivity, MySQL also provides libraries to embed a MySQL database server within a program. Currently the following solutions can be used to embed MySQL:

- **libmysqld**

The embedded edition of the **mysqld** server program wrapped in a shared library. Allows the MySQL Server to be embedded in C programs.

- **MySQL MXJ**

A JAR wrapper around **mysqld** binaries. This allows java programs and J2EE environments to instantiate (and install) a MySQL server.

17

1.4 MySQL Services

1.4.1 MySQL Training

Sun Microsystems offers a comprehensive set of MySQL training courses that give you a competitive edge in building world-class database solutions.

Courses can be chosen individually, as part of a bundle, and/or following our suggested curriculum path for Developers and Database Administrators (DBAs).

1.4.2 MySQL Certification

MySQL Certification Program is a high quality certification program that provides Developers and DBAs with the credentials to prove they have the knowledge, experience and skills to use and manage MySQL Server.

With MySQL personal certifications, you get to show that you are among the best-of-breed MySQL users and lay the foundation for becoming a trusted and valuable resource for your company and customers.

1.4.3 MySQL Consulting

We offer a full range of consulting services. We have an affordable solution for you whether you are starting a new project, needing to optimize an existing MySQL application, or migrating from a proprietary database to MySQL. Using industry best practices and proven methodologies, your MySQL certified consultant will help you deliver on-time and on-budget.



1.4.4 MySQL Support

MySQL offers a full range of support options. MySQL Technical Support is designed to save you time and to ensure you achieve the highest levels of performance, reliability, and uptime.

- **Community support**

- Mailing Lists
- Forums (<http://forums.mysql.com/>)
- Community Articles (<http://dev.mysql.com/tech-resources/articles>)
- Bugs Database (<http://bugs.mysql.com/>)
- No direct access to support engineers
- PlanetMySQL Blogs (<http://www.planetmysql.org/>)
- MySQL Reference Manuals (<http://dev.mysql.com/doc>)
- MySQLForge (<http://forge.mysql.com/>)

- **Purchased support**

- Enterprise subscription
- Support for MySQL Cluster
- Support for MySQL Embedded (OEM/ISV)
- Online Knowledge Base

1.5 The MySQL Enterprise Subscription

18

A MySQL Enterprise *subscription* gives you access to the Enterprise Server as well as other premium products and services. It is a comprehensive set of enterprise-grade software, support and services to ensure the highest levels of reliability, security and uptime. As a proactive service that helps you eliminate problems before they occur, it gives you everything you need in a single, unified offering to successfully develop and deploy business critical applications using MySQL.



19

1.5.1 MySQL Enterprise Server

The most reliable, secure and up-to-date version of the world's most popular open source database for cost-effectively delivering E-commerce, Online Transaction Processing (OLTP), and multi-terabyte Data Warehousing applications. With an Enterprise subscription you also get the following in addition to the Enterprise Server:

- **Updates and Service Packs**

- Automatically receive both monthly rapid updates and quarterly service packs with the latest bug-fixes and security updates.



- **Emergency Hot Fix Builds**
 - Can request a special hot fix build that resolves a critical issue that is not addressed by a service pack.
- **Drivers**
 - Many different drivers are supported. See MySQL Enterprise website for details.
- **MySQL Workbench**
 - A graphical user interface (GUI) that simplifies database design and maintenance, automates time-consuming and error-prone tasks, and improves communication among DBA and developer teams.
- **Platforms**
 - Many different platforms are supported. See MySQL Enterprise website for details.

20

1.5.2 24x7 Production Support

- **Service Level**
 - MySQL Enterprise gives you the flexibility to choose a service level that matches your requirements.
 - **Basic, Silver, Gold, or Platinum** (with incrementally increasing quantity and quality of support)
- **Problem Resolution Support**
 - Ensures you receive high priority service from the MySQL Support Team for quick resolution of technical problems as they occur. The services available to you are dependent on your chosen service level.
- **Consultative Support**
 - Expert MySQL Consultants provide you with proactive advice on how to properly design and tune your MySQL servers, schema, queries and replication to ensure the highest possible reliability.
- **Technical Account Manager (TAM)**
 - You have the option of adding a Technical Account Manager (*Platinum level only*) to be your liaison within MySQL. Your TAM be your single point of contact, providing a custom review of your systems, regular phone calls and on-site visits, to guarantee that you get the most out of MySQL Support Services.
- **Online Knowledge Base**
 - For quick self-help knowledge, you will have access to a comprehensive and easily searchable knowledge base library with hundreds of technical articles regarding difficult problems on popular database topics such as performance, replication, configuration, and security.

NOTE: For more detailed information regarding Enterprise support, please see our Enterprise web page:
<http://www.mysql.com/products/enterprise/support.html>



1.5.3 MySQL Enterprise Monitor

21

The **MySQL Enterprise Monitor** is a web-based monitoring and advising system. The Enterprise Monitor helps MySQL DBAs manage more MySQL servers in a scale-out environment, tune their current MySQL servers and find and fix problems with their MySQL database applications before they can become serious problems or costly outages. Running completely within the corporate firewall, the Enterprise Monitor pro-actively monitors enterprise database environments and provides expert advice on how MySQL can tighten security, optimize performance and reduce downtime of their MySQL powered systems. EM accomplishes all this while reducing DBA time and effort.

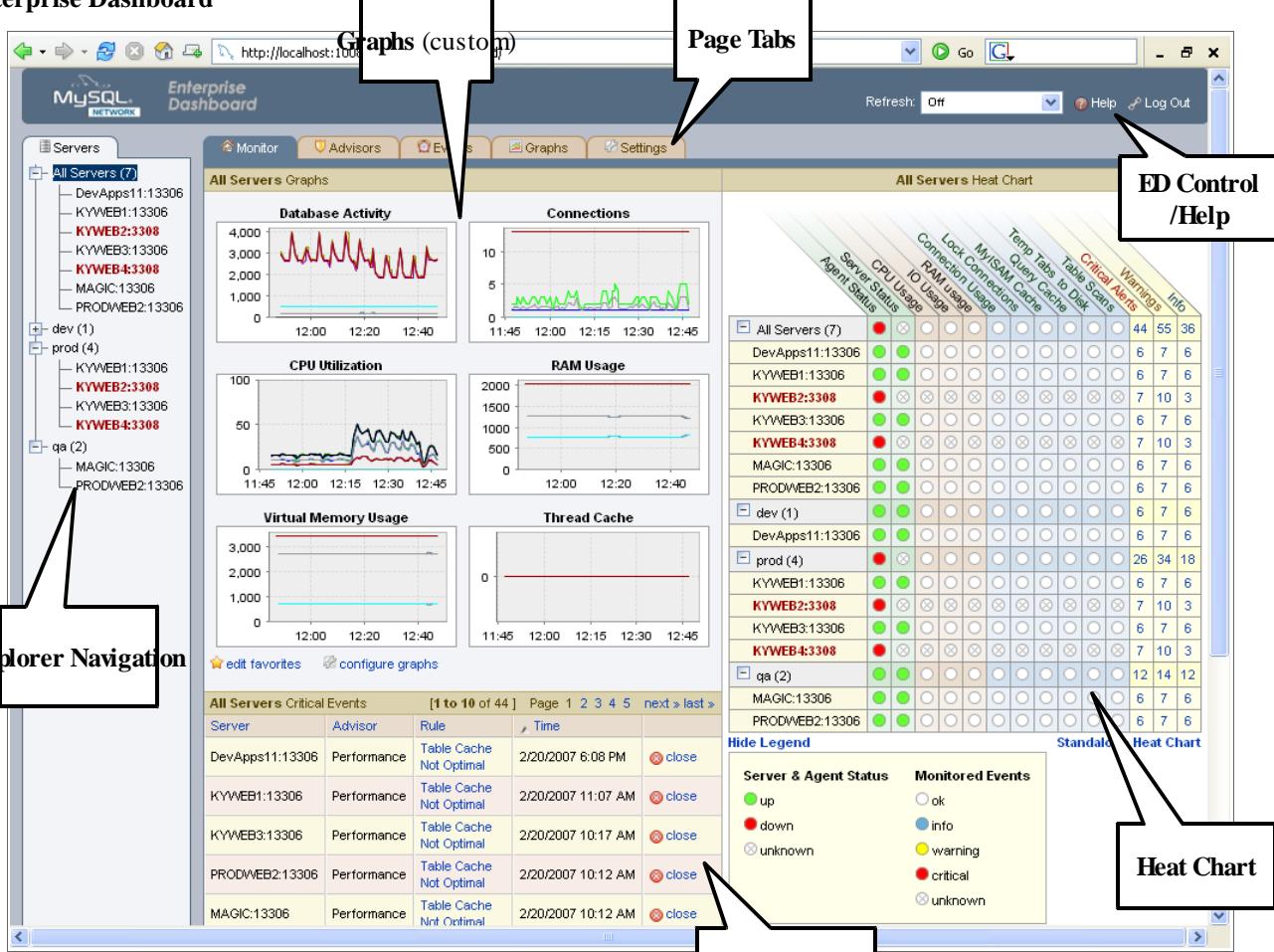
EM provides a rich GUI Enterprise Dashboard that contains “pages” which allow the DBA to have an immediate, graphic view of administrative tasks, server and database status, and advisory information.

The principle features of Enterprise Monitor:

- Enterprise Dashboard
 - Manage all MySQL servers from a consolidated console
- Server or Server-group management
 - Auto detection, grouping and monitoring of replication and scale-out topologies
- Monitoring page
 - “At a glance” global health check of key systems
- MySQL provided Advisors and Advisor Rules
 - Enforce MySQL Best Practices
- Advisor Rule Scheduler
 - Schedule unattended operations
- Customizable Thresholds and Alerts
 - Identify Advisor Rule violations
- Custom Advisor Rule
 - User defined advisor rules
- Events and Alert history
 - Lists *all* Advisor Rules executions
- Specialized Scale-Out assistance



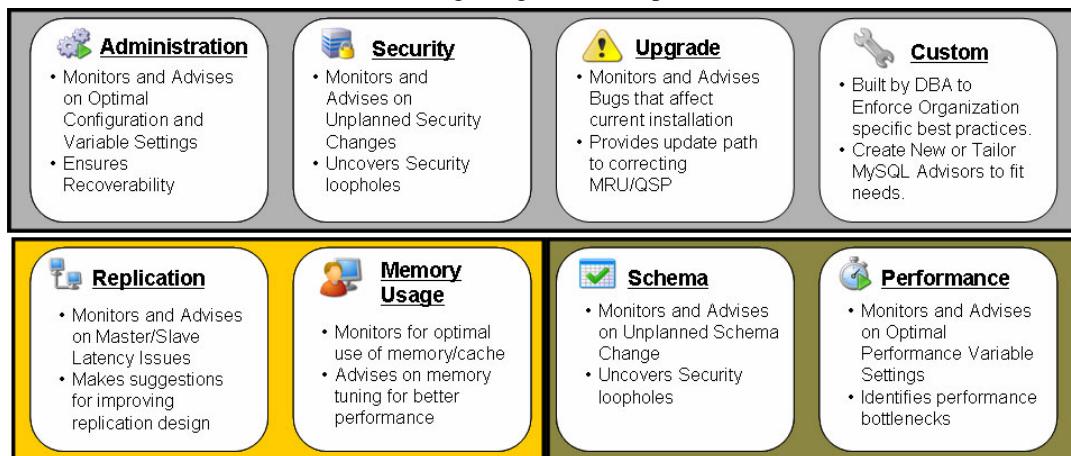
22

Enterprise Dashboard

23

MySQL Enterprise Advisors

The MySQL advisors are divided into the following categories (and specific functions):



24

Enterprise Monitor Subscription Levels

A “Silver” or higher level is required to get Enterprise Monitor. Some features are only available with the “Gold” and “Platinum” levels, respectively. A list of features per level is shown below:

	Basic	Silver	Gold	Platinum
Enterprise Dashboard		✓	✓	✓
Notifications and Alerts	✓	✓	✓	✓
Custom Advisor	✓	✓	✓	✓
Upgrade Advisor	✓	✓	✓	✓
Administration Advisor	✓	✓	✓	✓
Security Advisor	✓	✓	✓	✓
Replication Monitor		✓	✓	✓
Replication Advisor		✓	✓	✓
Query Analyer		✓	✓	✓
Memory Usage Advisor			✓	✓
Schema Advisor				✓
Performance Advisor				✓



25

1.5.4 Obtaining a MySQL Enterprise Subscription

For volume discounts or to order by phone, the MySQL Enterprise subscription can be obtained by contacting Sun Microsystems/MySQL sales personnel:

MySQL Enterprise Sales Inquiries

Please provide your contact information so a MySQL Sales representative can help you.

Got a question that can't wait?

- USA - Toll Free: +1-866-221-0634
- USA - From abroad: +1-408-701-9009
- USA - Subscription Renewals: +1-866-830-4410
- Latin America: +1 512 535 7751
- UK: +44 845 399 1124
- Ireland: +353 1 6919191
- Germany: +49 89 420 95 98 95
- France: +33 1 70 61 48 95
- Sweden: +46 730 207 871
- Benelux: +358 50 5710 528
- Italy: +39 06-99268193
- Israel: +358 50 5710 528
- Spain & Portugal: + 34 933905461
- Other EMEA countries: +353 1 6919191
- Asia Pacific: +81 3 5918 7507

A permanent MySQL Enterprise subscription can be purchased through our website:
<https://shop.mysql.com/enterprise/>.

A 30-day trial (with limited features) is also available: <http://www.mysql.com/trials/>.



26

1.6 Supported Operating Systems

MySQL runs on more than 20 platforms, giving users the kind of flexibility that puts them in control. Windows, Linux and Solaris are the most popular operating systems for running MySQL. Versions of MySQL are currently available for the following operating systems:

- FreeBSD
- HP-UX
- IBM AIX and i5
- Linux (multiple)
- Mac OS/X
- Microsoft Windows (multiple)
- Novell netware
- Open BSD
- QNX
- SGI Irix
- Solaris
- Source code
- Special builds



This list is continually being updated. For the most current information, please check our website: <http://www.mysql.com>. Source code and special builds are also available.



1.7 MySQL Certification Program

The MySQL Certification Program is a high quality certification program that provides developers and DBAs with the credentials to prove they have the knowledge, experience and skills to use and manage MySQL Server. MySQL provides several certification types and levels:

1. **Certified MySQL Associate (CMA)** - an entry level certification. It is intended for those that are relatively new to using the MySQL database server and covers basic database management system concepts as well as basic SQL. We recommend the CMA certification for MySQL users that know the basics, but have not yet obtained the experience gained by professional MySQL DBAs or Developers.
2. **Certified MySQL 5.0 Developer (CMDEV)** - proves mastery of the fundamental skills of using MySQL including creating and using databases and tables, inserting, modifying, deleting, and retrieving data.
3. **Certified MySQL 5.0 Database Administrator (CMDBA)** - proves mastery of the ability to manage MySQL Server including such advanced areas of database management, installation, security, disaster prevention and optimization.
4. **Certified MySQL Cluster DBA (CMCDBA)** - part of the DBA track which represents an advancement level exceeding CMDBA certification. In order to attain CMCDBA certification, you must attain CMDBA certification and pass one CMCDBA exam.

Certification Web Page

For more information on the certification program and the content of the exams, see our Certification web page; <http://www.mysql.com/certification/>.

Exam Administration

All exams are administered through one of more than 3,000 **Pearson VUE** testing centers available world-wide. Visit the MySQL certification web page, online forum or email certification@mysql.com for more information.

 **Instructor Note:** Suggestion for extra exercise: go to www.vue.com/mysql and locate your nearest testing center. Information about all the exams can be found on the **Certification** web page. Have students find this page now, if the majority is interested.

1.8 Training Curriculum Paths

Sun Microsystems offers the most comprehensive set of MySQL training courses, providing a competitive advantage in creating high-quality database solutions. In addition to our open courses, we also offer in-house training.

The MySQL training services staff has put together great courses designed for success, and an excellent training path for each individual to reach his/her training goals. There are two curriculum paths:

28

Developer Path

- **Introductory Courses**

- **MySQL and PHP : Developing Dynamic Web Applications** – This course provides the tools needed for the development of dynamic web applications. This course takes the student from the basics to the advanced features in four hands-on, heavy duty training days. Each student will have the opportunity to build real-world applications during the course. *Prerequisites: Basic experience with designing HTML pages including HTML forms and experience with any programming language. Basic PHP skills.* 4 days in length.
- **MySQL for Beginners** – This course covers the fundamentals of SQL and relational databases, using MySQL as a teaching tool. *Prerequisites: Basic computer literacy is required. Previous experience with any command-line program (such as MS-DOS or the MS Windows command prompt) is beneficial.* 4 Days in length.

- **Intermediate Courses**

- **MySQL for Developers** – This instructor-led course is designed for students planning on developing applications that make use of MySQL 5.0 (and higher). This course covers essential SQL statements for data design, querying, and programming. In addition, it will prepare students for the MySQL Developer certification. *Prerequisites: Some experience with Relational Databases and SQL.* 5 days in length.
- **MySQL 5.0 Upgrading and New Features** – This instructor-led course will provide in-depth knowledge needed to become proficient using MySQL 5.0. This training course will provide quality time both on the topic, hands-on labs and with the expert instructor. *Prerequisites: Existing MySQL users who want to become proficient using MySQL 5.0.* 3 Days in length.

- **Advanced Courses**

- **MySQL Stored Procedures Techniques** – This instructor-led course with focus on labs is designed to teach you how to maximize the use of stored procedures along with the knowledge to discern when an application should contain stored procedures and when they should not. *Prerequisites: Having attended the MySQL for Developers course, or similar knowledge.* 2 Days in length.
- **MySQL Developer Techniques** – This instructor-led course will provide database developers with additional skills in the design, development and maintenance of data and the queries to mine such data. This course places an emphasis on best practices for developers that improve response time of query executions. The database developer will also learn additional query writing techniques for creating reports, and creating trees and hierarchies, that can be used within MySQL to support the needs of the end users. *Prerequisites: Having attended the MySQL for Developers course, or similar knowledge.* 3 Days in length.



Instructor Note: The virtual courses are not mentioned on the slides, so please DO MENTION that we have virtual courses available now. And more are being developed all the time!

Database Administrator (DBA) Path

● Introduction Courses

- **MySQL for Beginners** – This course covers the fundamentals of SQL and relational databases, using MySQL as a teaching tool. *Prerequisites: Basic computer literacy is required. Previous experience with any command-line program (such as MS-DOS or the MS Windows command prompt) is beneficial.* 4 Days in length.

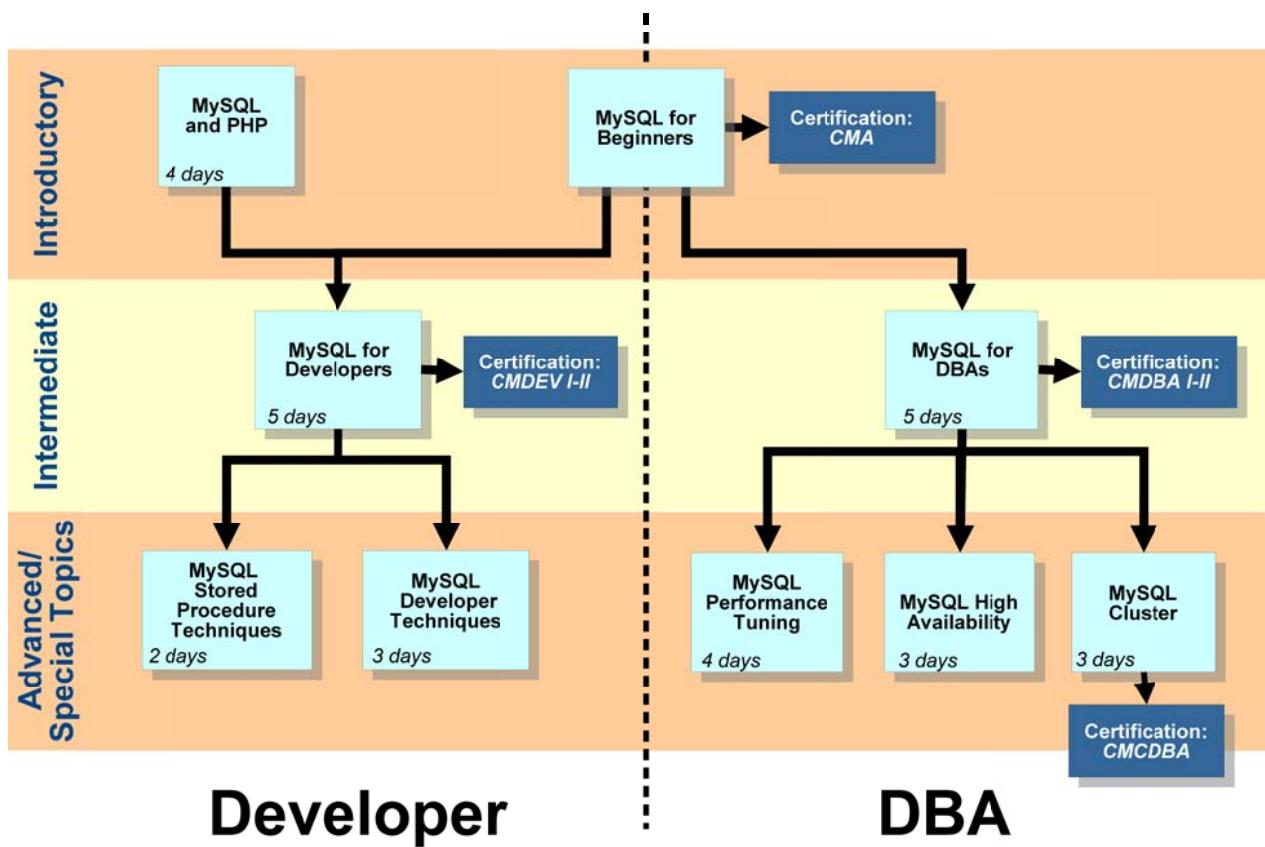
● Intermediate Courses

- **MySQL 5.0 Upgrading and New Features** – This instructor-led course will provide in-depth knowledge needed to become proficient using MySQL 5.0. This training course will provide quality time both on the topic, hands-on labs and with the expert instructor. *Prerequisites: Existing MySQL users who want to become proficient using MySQL 5.0.* 3 Days in length.
- **MySQL for Database Administrators** – This course covers essential DBA tasks such as, installation and upgrading, user management, disaster recovery, and optimization. In addition, it will prepare students for the MySQL Database Administrator certification. *Prerequisites: Some experience with Relational Databases and SQL.* 5 days in length.

● Advanced Courses

- **MySQL Performance Tuning** – The MySQL Performance Tuning course is designed for Database Administrators and others who wish to monitor and tune mysql. This course will prepare each student with the skills needed to utilize tools for monitoring, evaluating and tuning. Students will evaluate the architecture, learn to use the tools, configure the database for performance, tune application and SQL code, tune the server, examine the storage engines, assess the application architecture, and learn general tuning concepts. *Prerequisites: Attendance to the MySQL for Database Administrators or an equivalent mastery of database concepts, SQL and the MySQL server.* 4 Days in Length.
- **MySQL Cluster** - Learn how to install and configure the cluster nodes to ensure high availability. Also learn about all of the high availability features that are implemented in the MySQL Cluster storage engine — fail-over between storage nodes, network partitioning protocol, two-phase commit and much more. *Prerequisites: Attendance to the MySQL for Database Administrators or an equivalent mastery of database concepts, SQL and the MySQL server.* 3 Days in length. **Note:** This is the sole course offering in the Cluster certification (CMCDBA) path.
- **MySQL High Availability** – This course is designed for experienced database administrators and system architects that want to analyze and form a basis of understanding different high availability options, including clustering and replication solutions within MySQL. This course will provide the tools required to make the decision of what high availability solution is appropriate and how to implement a system with the correct design. *Prerequisites: Attendance to the MySQL for Database Administrators or an equivalent mastery of database concepts, SQL and the MySQL server.* 3 days in length.

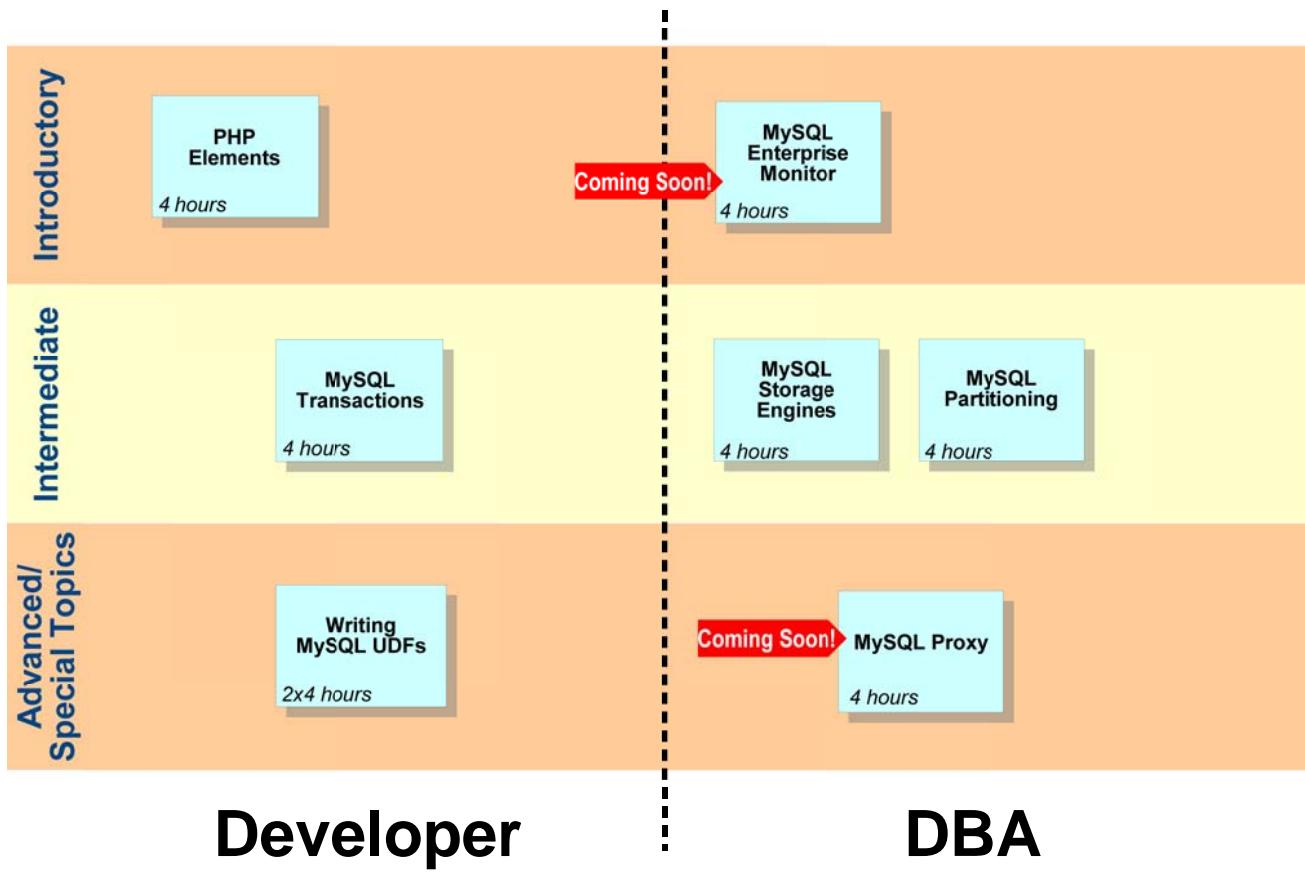


Curriculum Path Chart**Developer****DBA**

29

Virtual Classroom

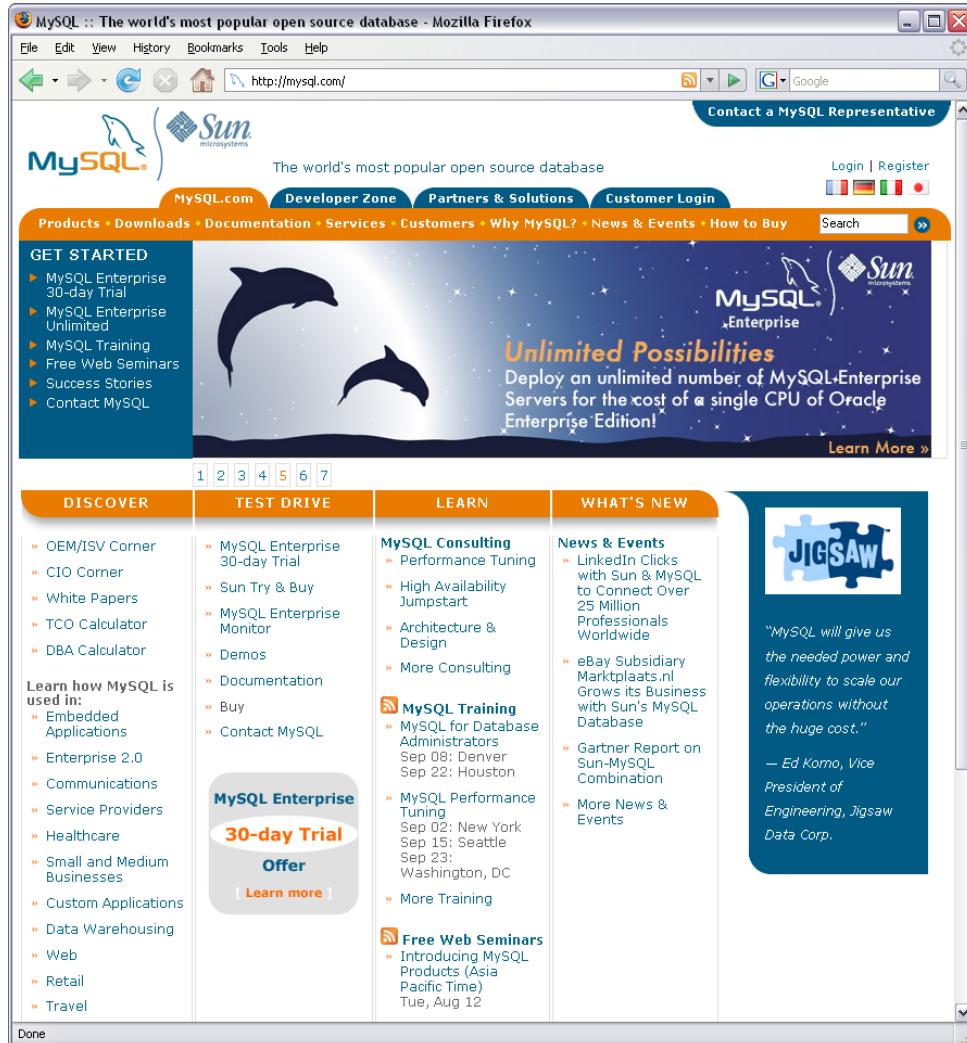
MySQL also offers virtual (online) courses covering various topics in relation to the MySQL suite of tools. These classes are instructor-led and delivered synchronously, via the web. Information regarding the available courses can be found on the MySQL training web page.



30

1.9 MySQL Website

Everything you ever wanted to know about MySQL (and more) can be found on our website: <http://www.mysql.com>*. From the home page, you may navigate the web site with the tabs (pull-down menus) across the top, the menu along the left side, and/or the many links on the page.



* MySQL information, products and services can also be accessed through the **Sun Microsystems** website: <http://www.sun.com/software/products/mysql/>.



Instructor Note: Tell students that the original MySQL website is fully operational, accessible and up-to-date, and will continue to be supported indefinitely. When a full transition to the Sun website is being made, we will notify our customers via the original website, as well as by other means of communication.

You can show the Sun website (and the link the MySQL info) at this time, if desired.

31

1.9.1 MySQL Community Web Page

The MySQL Community web page is located at <http://dev.mysql.com> and is maintained by MySQL AB. The “Developer Zone” page, as it is called, is the main support tool for the MySQL open source community and can also provide valuable insight for Enterprise users.

The screenshot shows the MySQL Developer Zone homepage. At the top, there's a navigation bar with links for File, Edit, View, History, Bookmarks, Tools, and Help. The address bar shows the URL <http://dev.mysql.com/>. On the right side of the header, there are links for "Contact a MySQL Representative", "Login | Register", and a search bar. The main content area features the MySQL logo and the Sun Microsystems logo. A banner for "MySQL Workbench Visual Database Design" is prominently displayed, along with a "Learn More" button. To the left, there's a sidebar with links for "Get Started with MySQL", "Developing with:", and "Quality Contribution Program". Below that is a section for "MySQL Server Community Edition" releases, including GA 5.0.67, RC 5.1.26, and Alpha 6.0.5. Another section lists "New Releases" such as MySQL Server 5.0 (GA 5.0.67), Connector/ODBC 3.51 (GA 3.51.26), MySQL Server 5.1 (RC 5.1.26), MySQL Server 6.0 (Alpha 6.0.5), and MySQL Workbench (GA 5.0.23). The central part of the page contains developer articles, including one about helping Andrii Nikitin's son Ivan, a MySQL 5.1 Use Case Competition, and a Getting started with Bazaar for MySQL code article. A sidebar on the right shows a visual representation of a database schema with tables like mysql.user, mysql.db, and mysql.proc.



32

Information such as the following can be found on the **Developer Zone** web page:

- **Current product and service promotions**
- **Get Started with MySQL**
 - Installation information page for MySQL beginners
- **Developing with:**
 - Links to specific information on using MySQL with; PHP, Perl Python, Ruby, Java/JDBC and .Net/C#/Visual Basic
- **Quality Contribution Program**
 - Structured program for software contributions and bug reports
 - Contributors acknowledged/rewarded
- **MySQL Server Community Edition**
 - All current General Available (GA), Release Candidate (RC) and Alpha versions of the MySQL Server
- **New Releases**
 - Latest code release for all MySQL products
- **Software Previews**
 - New features available for preview by users
- **What's New**
 - Latest news about the company, products, services, etc.
- **MySQL Training**
 - List of upcoming courses provided by MySQL Training Services
- **MySQL Quickpoll**
 - Give your opinions about current MySQL-related issues
- **Stay Connected**
 - Blogs, Lists, Guilds, MySQL Meetups and Quality Contributions
- **Resources**
 - White Papers, Articles, Training, Webinars, and MySQL Newsletter
- ***And much more!***



1.9.2 MySQL Online Documentation

Documentation for all MySQL products can be found from our website, on the “Documentation” page: <http://dev.mysql.com/doc/>. The page includes links for downloading the following;

- MySQL Reference Manual
- Excerpts from the Reference Manual
- MySQL GUI Tools Manuals
- Expert Guides
- MySQL Help Tables
- Example Databases
- Meta Documentation
- Community Contributed Documentation
- Printed Books
- Additional Resources



Instructor Note: Demonstrate the use of the online **MySQL Reference Manual**. Make sure to stress the use of the reference manual throughout the class. Also, show the use of the “shortcut” to the online reference manual: mysql.com/<topic>. If a user can't get an answer for their questions from our website resources, one option is to pay for support from Sun Microsystems, which puts the user in direct contact with MySQL support personnel (e.g., MySQL Enterprise).



Inline Lab 1-A

In this exercise you will review some web pages on the MySQL website using the web browser of your choice.

Step 1. Place cursor in web browser URL field.

1. Click in the text area of the Address/Location Toolbar.

The URL currently listed will be selected.

Step 2. MySQL web address

1. In the toolbar text area, type:

www.mysql.com

The MySQL Homepage is displayed.

Step 3. Products

1. Click on the **Products** tab located at the top of the MySQL home page. Scroll down the list to review the various product information provided.

A list of currently available products will appear, with links for further information and downloads.

Step 4. MySQL Enterprise

1. Review the details of the new **MySQL Enterprise** program, by clicking on the **Learn More >>** link.

MySQL® Enterprise™ provides a comprehensive set of enterprise-grade software, support and services directly from the developers of MySQL to ensure the highest levels of reliability, security and uptime.

Step 5. Services

1. Click on the **Services** tab located near the top of the MySQL home page.

A list of currently available services will appear, with links for further information and downloads.

Step 6. MySQL Training and Certification

1. Review the details of the **MySQL Training & Certification** program, by clicking on the **Learn More >>** link.

Featured information on this page will be updated periodically. For specific Training sub-topics select one of the links in the sub-menu in the upper-left corner of the page.

Step 7. Certification

1. From the **MySQL Training & Certification** web page, select the **Certification** link in the sub-menu and review the contents.

Featured information on this page will be updated periodically. For specific Certification sub-topics select one of the links in the sub-menu in the upper-left corner of the page.



Step 8. Return to Services

1. Click on the **Services** tab located at the top of the MySQL home page.

Returns to the top level Services page.

Step 9. Support Services

1. Review the details of MySQL **Support** programs, by clicking on the **Learn More >>** link. (*Continued on next page.*)

Shows the various support programs available.

Step 10. MySQL Community

1. Click on the **Developer Zone** tab located at the top of the MySQL home page.

The community page contains many links to more information and access to various open communication forums.

Step 11. News & Events

1. Return to the MySQL home page using the **MySQL.com** tab at the top of the page, then click on the **News & Events** tab located near the top of the MySQL home page.

Shows the latest MySQL happenings, from news stories to upcoming seminars.



34

1.10 Installing MySQL

It is recommended to download the MySQL database server from the MySQL web site downloads page: <http://dev.mysql.com/downloads/>. There are several different platforms supported, and there are differences in the specific installation details for each.



Inline Lab 1-B

This lab requires you to use the **Windows OS Wizard** for installing the MySQL server.

Step 1. Install MySQL

1. For installations on the Windows and Linux Operating systems, see **Appendix A** at the back of this training guide.

See **Appendix A**.



Instructor Note: If the MySQL server has not yet been loaded on the student computers, go through the installation steps with them now. The steps can be found in Appendix A at the back of the book, which will load a recent version of the 5.1 code. (Keep in mind that the labs are based on the 5.1.19 release.)

- Show the installation directions in the Reference manual, if time permits: <http://dev.mysql.com/doc/refman/5.1/en/installing.html> – Chapter 2.

35

1.11 Installing the 'world' database

MySQL provides three example databases to be used for testing server features and training. These databases can be found on the MySQL website documentation page; <http://dev.mysql.com/doc/>. The 'world' database will be used throughout this course.



Inline Lab 1-C

This lab requires you to use the MySQL command line client in order to load the world database. In order to use the pre-made tables in the **world** database, you will need to create the database (empty) and then upload the file containing the table data.

Step 1. Create 'world' database

1. Type the following in the MySQL command line client:

```
mysql> CREATE DATABASE world;
```

Creates the (empty) world database. Returns following message;

```
Query OK, 1 row affected (0.02 sec)
```

Step 2. Select the 'world' database

1. Type:

```
mysql> USE world;
```

Instructs client to use the newly created **world** database. Returns following message;

```
Database changed
```

Step 3. Build 'world' database tables

1. Type.*

```
mysql> SOURCE C:/world.sql
```

* Instructor will give you the file path if it is different than shown here. Several Query messages will scroll passed while tables and data are being uploaded for the **world** database.

 **Instructor Note:** You should download the **world.sql** file prior to the above lab and share it with the students so they do not have to download it from the website.

36

1.12 Chapter Summary

This chapter introduced you to MySQL and the support organization for MySQL. You should now be able to:

- Explain the origin and status of the MySQL product
- List the available MySQL products and professional services
- Describe the MySQL Enterprise subscription
- List the currently supported operating systems
- Describe the MySQL Community web page
- Describe the MySQL Certification program
- List all the available MySQL courses



2 MYSQL CLIENT/SERVER CONCEPTS

38

2.1 Learning Objectives

This chapter introduces the Client/Server model used in MySQL. In this chapter, you will learn to:

- Describe the MySQL Client/Server model
- Understand how the server supports storage engines
- Explain the basics of how MySQL uses memory and disk space



39

2.2 MySQL General Architecture

MySQL operates in a networked environment using a client/server architecture. In other words, a central program acts as a server, and various client programs connect to the server to make requests. A MySQL installation has the following major components: MySQL server, client programs and MySQL non-client utilities.

40

2.2.1 MySQL Server

The executable `mysqld` is the MySQL database server program. In this context, the term *MySQL Server* denotes one running instance of the executable `mysqld`.

The server manages access to the actual collections of data that physically reside on disk and in memory. Such a collection of data is called a *database*. In the context of MySQL, the terms *database* and *schema* can be used interchangeably.

On a logical level, relational databases manage data in the form of *tables*. The MySQL server features a modular architecture that supports multiple *storage engines* to handle the mapping between the logical representation of the data (the tables) and the physical storage of the data (such as files on disk or main memory). These different storage engines give rise to different types of tables. For example, it is dependent upon the storage engine whether a table can handle data in a transactional manner.

The MySQL Server is multi-threaded and supports many simultaneous client connections. Clients can connect via several connection protocols.

New Version Documentation

The exact feature configuration of MySQL Server may change over time, so whenever you download a new version, it's wise to check the documentation.

Keep in mind the difference between a *server* and a *host*. The server is software (the MySQL server program `mysqld`). Server characteristics include its version number, whether certain features are included or excluded, and so forth. The host is the physical machine on which the server program runs. Host characteristics include its hardware configuration, the operating system running on the machine, its network addresses, and so forth.

41

2.2.2 Client Programs

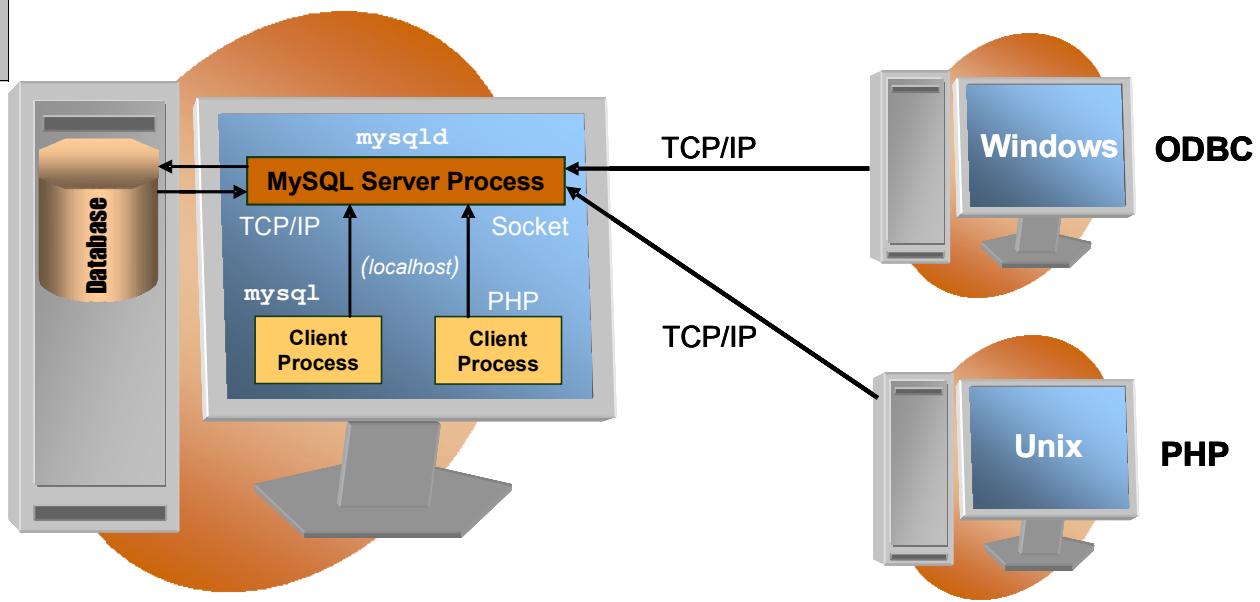
These are programs that are used for communicating with the server to manipulate the databases that are managed by the server. Sun Microsystems, Inc. provides several client programs for the mysql server. The following list describes a few of them:

- `mysql` is a command-line program that acts as a text-based front end for the server. It's used for issuing queries and viewing the results interactively from a terminal window.
- Other command-line clients include `mysqlimport` for importing data files, `mysqldump` for making backups, `mysqladmin` for server administration, and `mysqlcheck` for checking the integrity of the database files.

MySQL runs on many varieties of Windows, Unix and Linux. However, client/server communication is not limited to environments where all computers run the same operating system. Client programs can connect to a server running on the same host or a different host, and the client and server host need not have the same operating system. For example, client programs can be used on Windows to connect to a server that is running on Linux.



42



Most of the concepts discussed here apply universally to any system on which MySQL runs. Platform-specific information is so indicated. Unless otherwise specified, “Unix” as used here includes Linux and other Unix-like operating systems.

43

2.2.3 Communication Protocols

The following describes the different communication protocols that are used to interact with the MySQL server in greater detail:

- **TCP/IP** - Transmission Control Protocol/Internet Protocol, the suite of communications protocols used to connect hosts on the Internet. TCP/IP is built into the UNIX operating system and is used by the internet, making it the de facto standard for transmitting data over networks. Even network operating systems that have their own protocols, such as Netware, also support TCP/IP.
- **Unix Socket** - In the world of computers, a Unix socket is a form of inter-process communication used to form one end of a bi-directional communication link between processes on the same machine. (Requires a physical file on the local system.)
- **Shared Memory** - an efficient means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access. This Windows explicit ‘passive’ mode only works within a single (Windows) machine.



- **Named Pipes** - the design of named pipes is biased towards client-server communication, and they work much like sockets: other than the usual read and write operations, Windows named pipes also support an explicit "passive" mode for server applications. Only works within a single (Windows) machine. Not all versions of Windows support named pipes or shared memory connections. And even if it is supported, you may have to enable it in the configuration.

44

2.2.4 MySQL Non-Client Utilities

These are programs that act independently of the server. `myisamchk` is an example. It performs table checking and repair operations. Another program in this category is `myisampack`, which creates compressed read-only versions of MyISAM tables. Both utilities operate by accessing MyISAM table files directly, independent of the `mysqld` database server.

45

2.3 How MySQL Uses Disk Space

MySQL Server uses disk space in several ways, primarily for directories and files that are found under a single location known as the server's data directory. The server uses its data directory to store all the following:

- Database directories. Each database corresponds to a single directory under the data directory, regardless of what types of tables you create in the database.
- Table format files (`.frm` files) that contain a description of table structure. Every table has its own `.frm` file, located in the appropriate database directory. This is true no matter which storage engine manages the table.
- Data and index files are created for each table by some storage engines and placed in the appropriate database directory. For example, the MyISAM storage engine creates a data file and an index file for each table.
- The InnoDB storage engine has its own tablespace and log files.
- Server log files and status files. These files contain information about the statements that the server has been processing. Logs are used for replication and data recovery, to obtain information for use in optimizing query performance, and to determine whether operational problems are occurring.
- Triggers are stored in a database directory along with the affected table.
- Authentication information is stored on disk (`mysql` database).
- `mysqld` itself needs space for the program code.



46

2.3.1 How MySQL Uses Memory

MySQL server memory use includes data structures that the server sets up to manage communication with clients and to process the contents of databases. The server allocates memory for many kinds of information as it runs:

- Thread handlers. The server uses several buffers (caches) to hold information in memory for the purpose of avoiding disk access when possible.
- The MEMORY storage engine creates tables that are held in memory. These tables are very fast because no transfer between disk and memory need be done to access their contents.
- The server might create internal temporary tables in memory during the course of query processing.
- The server maintains several buffers for each client connection.
- Global buffers and caches.





Quiz

In this exercise you will answer questions pertaining to MySQL client/server Concepts.

Note: Choose the best answer from the answers given for each question.

1. All of the MySQL client programs and utilities communicate with the MySQL server. (True or False)
2. When running a MySQL server under Windows, client programs accessing that server also must run under Windows. (True or False)
3. A command-line program commonly used to communicate with the server is called mysqld. (True or False)
4. Every table has its own _____ file (containing a description of table structure), located in the appropriate database directory.
5. The server executes each statement using a two-tier processing model:
 - The upper tier includes the SQL parser and optimizer.
 - The lower tier comprises a set of _____.



47

2.4 Chapter Summary

In this chapter, you have learned to:

- Describe the MySQL client/server model
- Understand how the server supports storage engines
- Explain the basics of how MySQL uses memory and disk space



3 MYSQL CLIENTS

49

3.1 Learning Objectives

This chapter introduces the MySQL Client Programs. In this chapter, you will learn to:

- Invoke client programs within the MySQL Client/Server architecture
- Use many of the features of the mysql client
- Download and use the MySQL Query Browser to connect to the mysql server
- Describe the client interfaces provided by Sun Microsystems, Inc.
- Distinguish between the client interfaces and choose according to need
- Use MySQL website for downloading the MySQL client interface programs
- Understand the relationship to third-party client interfaces



50

3.2 Invoking Client Programs

MySQL client programs can be invoked from the command line, such as from a Windows console prompt or a Unix shell prompt. When a client program is invoked, it is possible to specify options following the program name to control its behavior. Options also can be given in option files. Some options tell the client how to connect to the MySQL server. Other options tell the program what actions to perform.

Most examples in this section use the `mysql` program, but the general principles apply to other MySQL client command-line programs as well.

To determine the options supported by a MySQL program, invoke it with the `--help` option. For example, to find out how to use `mysql`, use this command:

```
shell> mysql --help
```

To determine the version of a program, use the `--version` option. For example, the following output from the `mysql` client indicates that it is from MySQL 5.1.30:

```
shell> mysql --version
mysql Ver 14.14 Distrib 5.1.30, for Win32 (ia32)
```

It is not necessary to run client programs that have the same version as the server. In most cases, clients that are older or newer than the server can connect to it successfully.

51

3.2.1 General Command Option Syntax

Options to MySQL programs have two general forms:

- Long options consist of a word preceded by double dashes.
- Short options consist of a single letter preceded by a single dash.

In many cases, a given option has both a long and a short form. For example, to display a program's version number, the long `--version` option or the short `-v` option can be used. These two commands are equivalent:

```
shell> mysql --version
shell> mysql -v
```

Options are case sensitive: `--version` is recognized, but lowercase variations such as `--Version` or `--VERSION` are not. This applies to short options as well: `-V` and `-v` are both legal options, but mean different things.

Instructor Notes:

- 4.0 or earlier clients can run into problems connecting to a 4.1 or later server (because of the password hashing algorithm).
- You may need to invoke MySQL programs using the pathname to the `bin` directory in which they are installed. For example, if you get a “program not found” error whenever you attempt to run from other than the `bin` directory. You can add the pathname of the `bin` directory to your `PATH` environment variable setting. That enables you to run a program by typing only its name. For example, if `mysql` is installed in `/usr/local/mysql/bin`, you'll be able to run it by invoking it as `mysql`; it will not be necessary to invoke it as `/usr/local/mysql/bin/mysql`.
- The **Windows** installer will ask if the `bin` directory should be included in the search path. If you choose this option, you will be able to invoke MySQL programs from any directory location.



Some options are followed by values. For example, when the **--host** or **-h** option is specified to indicate the host machine where the MySQL server is running, the option must be followed with the machine's host name. For a long option, separate the option and the value by an equals sign (=). For the short form, the **-h** option and the value can but need not be separated by a space. The option formats in the following three commands are equivalent. Each one specifies myhost.example.com as the host machine where the MySQL server is running:

```
shell> mysql --host=myhost.example.com
shell> mysql -h myhost.example.com
shell> mysql -hmyhost.example.com
```

In most cases, if an option is not explicitly specified, a program uses a default value. This makes it easier to invoke MySQL client programs because it is only necessary to specify those options for which the defaults are unsuitable. For example, the default server host name is `localhost`, so if the MySQL server necessary to connect to is running on the local host, it is not required to specify any **--host** or **-h** option.

Exceptions to these option syntax rules are noted in the following discussion wherever relevant. The most important exception is that password options have a slightly different behavior than other options.

52 3.2.2 Connection Parameter Options

To connect to a server using a client program, the client must know upon which host the server is running. A connection may be established locally to a server running on the same host as the client program, or remotely to a server running on a different host. To connect, it is necessary for a user to identify themselves to the server with a user name and (optionally) a password.

Each MySQL client has its own program-specific options, but all command-line clients support a common set of options for making a connection to the MySQL server. This section describes the options that specify connection parameters, and how to use them if the default values aren't appropriate. The discussion lists each option's long form and short form, as well as its default value.

The primary options for connecting to the server specify the type of connection to make and identify the MySQL account that is going to be used.

Options for establishing a connection

Option syntax	Meaning
<code>--protocol</code>	The protocol to use for the connection
<code>--host</code>	The host where the server is running
<code>--port</code>	The port number for TCP/IP connections
<code>--shared-memory-base-name</code>	The shared-memory name for shared-memory connections
<code>--socket</code>	The Unix socket filename or Windows named-pipe name
<code>--compress</code>	Can be used to compress communication

Instructor Notes:

There are also several `ssl-*` options available for encrypted connections

By default, **socket** will be used on UNIX for localhost connections, **tcp** on Windows for localhost connections.



- `--protocol=protocol_name`

This option, if given, explicitly selects the communication protocol that the client program should use for connecting to the server. (In the absence of a `--protocol` option, the protocol used for the connection is determined implicitly based on the server host name value and the client operating system, as discussed later.) The allowable values for the `--protocol` option are given in the following table.

--protocol value	Protocol	Applicable to
tcp	TCP/IP connection to local or remote server	All
socket	Unix socket file connection to local server	Unix only
pipe	Named-pipe connection to local server	Windows only
Memory	Shared-memory connection to local server	Windows only

- `--host=host_name` or `-h host_name`

This option specifies the machine where the MySQL server is running. The value can be a hostname or an IP address. The `hostname localhost` means the local host (that is, the computer on which the client program is running). On Unix, `localhost` is treated in a special manner. On Windows, the value `.` (period) also means the local host and is treated in a special manner as well.

The default host value is `localhost`.

- `--port=port_number` or `-P port_number`

This option indicates the port number to which to connect on the server host. It applies only to TCP/IP connections. The default MySQL port number is 3306.

- `--shared-memory-base-name=memory_name`

This option can be used on Windows to specify the name of shared memory to use for a shared-memory connection to a local server. The default shared-memory name is `MYSQL` (case sensitive).

- `--socket=socket_name` or `-S socket_name`

This option's name comes from its original use for specifying a Unix domain socket file. On Unix, for a connection to the host `localhost`, a client connects to the server using a Unix socket file. This option specifies the pathname of that file.

On Windows, the `--socket` option is used for specifying a named pipe. For Windows NT-based systems that support named pipes, a client can connect using a pipe by specifying `.` as the hostname. In this case, `--socket` specifies the name of the pipe. Pipe names aren't case sensitive.

- `--compress` or `-C`

This option causes all communication between the client and the server to be compressed. The result is a reduction in the number of bytes sent over the connection, which can be helpful on slow networks. This option adds to the computational cost for both client and server due to compression and decompression processes involved in the transmission process. The options `--compress` and `-C` do not take a value after the option name. MySQL uses the `gzip` algorithm for the compression.



53

Options for MySQL user identification

Option	Meaning
--user	The name of the user connecting to MySQL
--password	The password for the user connecting to MySQL

Two options provide identification information. These indicate the `username` and `password` of the account that is going to be used to start a new session on the server. The server rejects a connection attempt unless a value is provided for these parameters that correspond to a MySQL account that is listed in the server's grant tables.

- `--user=user_name` or `-u user_name`

This option specifies the username for the MySQL account. To determine which account applies, the server uses the `username` value in conjunction with the name of the host from which the connection is made. This means that there can be different accounts with the same `username`, which can be used for connections from different hosts.

On Windows, the default MySQL account name is ODBC. On Unix, client programs use the operating system login name as default MySQL account `username`.

- `--password=pass_value` or `-ppass_value`

This option specifies the password for the users MySQL account. There is no default password. If this option is omitted, the MySQL account must be set up to allow connecting without a password.

Password options are special in two ways, compared to the other connection parameter options:

- The password value can be omitted after the option name. This differs from the other connection parameter options, each of which requires a value after the option name. If the password value is omitted, the client program prompts the user interactively for a password, as shown here:

```
shell> mysql -p
Enter password:
```

When the `Enter password:` prompt is displayed, the user must type in their password and press Enter. The password isn't echoed as the user types, to prevent other people from seeing it.

- If the short form of the password option (`-p`) is used and the password value is given on the command line, there must be no space between the `-p` and the value. That is, `-ppass_val` is correct, but `-p pass_val` is not. This differs from the short form for other connection parameter options, where a space is allowed between the option and its value.

For example, `-hhost_name` and `-h host_name` are both valid. This exceptional requirement that there be no space between `-p` and the password value is a logical necessity of allowing the option parameter to be omitted.

Avoid entering your password on the command line

Never specify the password on the command line as it's visible on screen and might appear in the process list.



How to specify connection parameters

- Connect to the server using the default hostname and username values with no password:

```
shell> mysql
```

- Connect to the local server via shared memory (this works only on Windows). Use the default username and no password:

```
shell> mysql --protocol=memory
```

- Connect to the server on the local host with a username of myname, asking mysql to prompt the user for a password:

```
shell> mysql --host=localhost --password --user=myname
```

- Connect with the same options as the previous example, but using the corresponding short option forms:

```
shell> mysql -h localhost -p -u myname
```

- Connect to the server at a specific IP address, with a username of myname and password of mypass:

```
shell> mysql --host=192.168.1.33 -user=myname --password=mypass
```

- Connect to the server on the local host, using the default username and password and compressing client/server traffic:

```
shell> mysql --host=localhost --compress
```





Inline Lab 3-A

This lab requires you to invoke the mysql command line client using various options.

Step 1. Display the mysql client help features

1. Invoke the mysql client help feature from the command prompt, and determine the option and syntax used to indicate a specific database for use with the mysql connection:

```
shell> mysql --help
```

Outputs a list of mysql client options and their specific uses. The option requested is on the list:

```
-D, --database=name Database to use.
```

Step 2. Invoke the MySQL command line client

1. Invoke the mysql client with the long form of the user and password options. Use the password from the download (or as provided by instructor):

```
shell> mysql --user=root --password=<password>
```

Upon entering the command the mysql default prompt will appear as follows:

```
mysql>
```

Step 3. Exit the MySQL command line client

1. Exit the mysql client by typing exit at the mysql prompt

```
mysql> exit
```

Will return to the default command prompt.

```
shell>
```



Step 4. Invoke the MySQL command line client with multiple options

1. Invoke the mysql client with the short form of the port, no beep, user, and password options. Set the port to 3306:

```
shell> mysql -P 3306 -b -uroot -p<password>
```

Upon entering the command and options the mysql default prompt will appear as follows;

```
mysql>
```

Now exit the client

```
mysql> exit
```

Upon exiting:

```
Bye  
shell>
```



55

3.3 Using Option Files

As an alternative to specifying options on the command line, they can be placed in an option file. The standard MySQL client programs look for option files at startup time and use any appropriate options they find there. Putting an option in a file saves time and effort because it is not necessary to specify the option on the command line each time the program is invoked.

Options in option files are organized into groups, with each group preceded by a `[group-name]` line that names the group. Typically, the group name is the name of the program to which the group of options applies. For example, the `[mysql]` and `[mysqldump]` groups are for options to be used by `mysql` and `mysqldump`, respectively. The special group named `[client]` can be used for specifying options that all client programs must use. A common use for the `[client]` group is to specify connection parameters because typically all clients connect to the same server no matter which client program they use.

To write an option in an option file, use the long option format that would be used on the command line, but omit the leading dashes. If an option takes a value, spaces are allowed around the `=` sign, something that isn't true for options specified on the command line. Here's a sample option file:

```
[client]
host = myhost.example.com
compress

[mysql]
safe-updates
```

In this example, the `[client]` group specifies the server hostname and indicates that the client/server protocol should use compression for traffic sent over the network. Options in this group apply to all standard clients. The `[mysql]` group applies only to the `mysql` client. The group shown indicates that `mysql` should use the `--safe-updates` option. (`mysql` uses options from both the `[client]` and `[mysql]` groups, so it would use all three options shown.)

56

Where an option file should be located depends on the operating system. The standard option files are as follows:

- On Windows, programs look for option files in the following order: `my.ini` and `my.cnf` in the Windows directory (for example, the `C:\Windows` or `C:\WinNT` directory), and then `C:\my.ini` and `C:\my.cnf`. Whereas the Windows install wizard will place the configuration file in the `C:\Program Files\MySQL\MySQL Server <version>` directory.
- On Unix, the file `/etc/my.cnf` serves as a global option file used by all users. Also, a user-specific option file can be set-up by creating a file named `.my.cnf` in the users home directory. If both exist, the global file is read first.
- Also, MySQL command line programs search for option files in the MySQL base directory (for example, `../MySQL 5.1 Server/.`).

MySQL programs can access options from multiple option files. Programs look for each of the standard option files and read any that exist. No error occurs if a given file is not found.



To use an option file, create it as a plain text file using an editor. To create or modify an option file, the user must have write permission for it. Client programs need only read access.

57

To tell a program to read a single specific option file instead of the standard option files, the following options are available (*must be the first option on the command line*):

Option	Meaning
--defaults-file= <i>file_name</i>	Uses option file at specified location
--defaults-extra-file= <i>file_name</i>	Uses additional option file at specified location
--no-defaults	Tells program to ignore all option files

For example, to use only the file C:\my-opts for mysql and ignore the standard option files, invoke the program like this:

```
shell> mysql --defaults-file=C:\my-opts
```

Option files can reference other files to be read for options by using !include and !includedir directives:

- A line that says !include *file_name* suspends processing of the current option file. The file *file_name* is read for additional options, and then processing of the suspended file resumes.
- A line that says !includedir *dir_name* is similar except that the directory *dir_name* is searched for files that end with a .cnf extension (.cnf and .ini on Windows). Any such files are read for options, and then processing of the suspended file resumes.

If an option is specified multiple times, either in the same option file or in multiple option files, the option value that occurs last takes precedence.

Overriding configuration file options

When command line options are used when starting the MySQL client, they will override any config file options of the same name.





Inline Lab 3-B

This lab requires you to invoke the `mysql` command line client using option files.

Step 1. Construct an option file

1. Construct an option file (using text editor) with the following client settings; password, username, turn on safe updates mode, and turn on compress mode. Save this file in the C:\ drive directory. The text file should contain the following:

```
[mysql]
password=<password>
user=root
safe-updates
compress
```

Place all four commands in a text file, using syntax shown above.

2. Invoke the `mysql` client with the above additional option file:

```
mysql --defaults-extra-file=C:\my_opts.txt
```

Upon entering the command and options, the `mysql` default prompt will appear.

Step 2. Confirm settings

1. Confirm that the settings from the new option file are being used for this `mysql` client session by typing the following at the `mysql>` prompt;

```
mysql> STATUS;
```

A list of server connection specifications will be shown, including the particular settings from the option file. Check the list for the appropriate options settings:

```
...
Current user:          root@localhost
...
Protocol:              Compressed
...
Note that you are running in safe_update_mode:
UPDATEs and DELETEs that do not use a key in the WHERE clause are not
allowed.
...
```



3.4 The mysql command line client

This chapter discusses `mysql`, a general-purpose client program for issuing queries and retrieving their results. It can be used interactively or in batch mode to read queries from a file.

58

3.4.1 Using mysql interactively

The `mysql` client program enables users to send queries to the MySQL server and receive their results. It can be used interactively or it can read query input from a file in batch mode:

- Interactive mode is useful for day-to-day usage, for quick one-time SQL statements, and for testing how SQL statements work.
- Batch mode is useful for running SQL statements stored in a text file (also known as an *SQL script*). Batch mode is especially valuable for issuing a complex series of queries that is difficult or cumbersome to enter manually. SQL scripts are also used when queries need to be executed automatically according to a particular timed schedule without user intervention.

MySQL statements such as the version query below, shown below as executed within the `mysql` client, can also be run from the shell command prompt as part of the `mysql` client startup:

```
mysql> SELECT VERSION();  
+-----+  
| VERSION() |  
+-----+  
| 5.1.30-community |  
+-----+
```

A statement can be executed directly from the command line by using the `-e` or `--execute` option:

```
shell> mysql -u user_name -ppassword -e "SELECT VERSION()"  
+-----+  
| VERSION() |  
+-----+  
| 5.1.30-community |  
+-----+
```

No statement terminator is necessary unless the string following `-e` consists of multiple statements. In that case, separate the statements by semicolon characters.



59 3.4.2 Statement Terminators

Several terminators may be used to end a statement. Two terminators are the semicolon character (;) and the \g sequence. These are equivalent and may be used interchangeably:

```
mysql> SELECT VERSION(), DATABASE();
+-----+-----+
| VERSION() | DATABASE() |
+-----+-----+
| 5.1.30-community | world |
+-----+-----+

mysql> SELECT VERSION(), DATABASE()\g
+-----+-----+
| VERSION() | DATABASE() |
+-----+-----+
| 5.1.30-community | world |
+-----+-----+
```

For the mysql command line client, the sequence \g always works as a normal statement terminator. The usage of the semi-colon as terminator is a default that can be changed using the mysql command **DELIMITER**. To change the terminator, type **DELIMITER** right after the mysql> prompt, followed by a space and then the new delimiter:

```
mysql> DELIMITER go
mysql> SELECT COUNT(*)
      -> FROM world.City
      -> go
+-----+
| COUNT(*) |
+-----+
|     4079 |
+-----+
```

In the previous example, we first changed the statement delimiter to the string go. Then we issued the SQL statement. Now, typing go rather than the semi-colon triggered the statement to be sent to the server. An important thing to note is that the **DELIMITER** command is recognized and dealt with by the client tool – not the server.

Changing the default delimiter and the **DELIMITER** command is something we will discuss later in the sections on stored routines. Changing the default delimiter might also be useful for compatibility with other command line client tools.



Note that the semi-colon is not just the default delimiter for the mysql command line client. It has a meaning on the server as well. It is seen predominantly as the statement delimiter delimiting the statements appearing in stored routines, but it may also be used to batch a number of pure SQL statements. Obviously, this will only work if the client is instructed not to interpret the semi-colon as statement terminator, and this is what the **DELIMITER** command is used for most.

The `\G` sequence is also recognized as statement terminator, but has the additional effect of causing `mysql` to display the result in a vertical style, showing each column value on a separate line, separating each row with a line of asterisks:

```
mysql> SELECT VERSION(), DATABASE() \G
***** 1. row *****
VERSION(): 5.1.30-community
DATABASE(): world
```

The `\G` terminator is especially useful in case the default horizontal layout of the rows is wider than the screen. In that case, the output is wrapped, making it much more difficult to interpret the result.

60

Statement terminators are necessary for another reason as well: `mysql` allows a single query to be entered using multiple input lines. This makes it easier to issue a long query because it is possible to enter a statement over the course of several lines. `mysql` does not send the query to the server unless it sees the statement terminator. For example:

```
mysql> SELECT Name, Population FROM City
-> WHERE CountryCode = 'IND'
-> AND Population > 3000000;
+-----+-----+
| Name           | Population |
+-----+-----+
| Mumbai (Bombay) | 10500000 |
| Delhi          | 7206704  |
| Calcutta [Kolkata] | 4399819 |
| Chennai (Madras) | 3841396 |
+-----+-----+
```

In the preceding example, notice what happens when a statement is not completed on a single input line: `mysql` changes the prompt from `mysql>` to `->` to provide feedback that it's still waiting to see the end of the statement.

If a statement results in an error, `mysql` displays the error message returned by the server:

```
mysql> This is an invalid statement;
ERROR 1064 (42000): You have an error in your SQL syntax.
```

61

If it is necessary to terminate a statement that is being composed, enter `\c` and `mysql` will cancel the statement and return a new `mysql>` prompt:

```
mysql> SELECT Name, Population
-> FROM City
-> WHERE \c
mysql>
```

To quit `mysql`, use `\q`, `QUIT`, or `EXIT`:

```
mysql> \q
```



DELIMITER, **QUIT** and **EXIT** are not SQL statements. They are not interpreted by the server: they are interpreted only by the client tool and never even reach the server.

62

3.4.3 The mysql Prompts

The mysql> prompt displayed by mysql is just one of several different prompts that might be visible when entering queries. Each type of prompt has a functional significance because mysql varies the prompt to provide information about the status of the statement that is being entered. The following table shows each of these prompts.

Prompt	Meaning
mysql>	Ready for new statement
->	Continue entering statement until a terminator like is entered
'>	Waiting for end of single-quoted string (')
">	Waiting for end of double-quoted string or identifier (")
`>	Waiting for end of backtick-quoted identifier (`)
/*>	Waiting for end of C-style comment (* /)

The mysql> prompt is the main (or primary) prompt. It signifies that mysql is ready to accept a new statement.

If in fact a current statement has been mistyped by forgetting to close a quote, a statement can be canceled by entering the closing quote followed by the \c clear-statement command.

63

To redefine the mysql> prompt, use the **prompt** command;

```
mysql> prompt win 1>
win 1>
```

Or for example, place current information in the prompt such as the user, host and database;

```
mysql> prompt (\u@\\h) [\d]\\>
PROMPT set to '(\u@\\h) [\d]\\>'
(root@localhost) [world]>
```

Everything following the first space after the prompt keyword becomes part of the prompt string, including other spaces. The string can contain special sequences.

To revert the prompt to the default, specify **prompt** or **\R** with no arguments.

```
(root@localhost) [world]> prompt
mysql>
```

Trailing spaces will be ignored. To have trailing spaces in the prompt, use _ at the end of the prompt setting.



64

3.4.4 Using Editing Keys in mysql

The mysql client supports input-line editing, which enables the recalling of previous lines that can then be edited. On Unix/Linux, it also supports tab-completion to make it easier to enter queries. With tab-completion, it is possible to enter part of a keyword or identifier and complete it using the `<Tab>` key.

Keyboard Editing with mysql

- The four arrow keys:
 - Right/Left moves the cursor one step Forward/Backward
 - Up/Down cycles the command history in the mysql client
- The mysql client has full readline capabilities under Unix in general
 - Ctrl+A or Home – to the beginning of the line (Home works under Windows too)
 - Ctrl+E or End – moves to the end of the line (End works under Windows too)
 - Ctrl+K – deletes characters from the position of the cursor to the end of the line
 - Ctrl+R – searches in reverse through the command history
- Command history is preserved between sessions
 - Persistent in Linux
 - .mysql_history file in user's home directory
 - History clears in Windows if the command prompt window is closed





Inline Lab 3-C

This lab requires you to invoke the mysql command line client to practice statement terminators and editing keys.

Step 1. Execute a MySQL statement from the shell

1. Invoke the mysql client non-interactively using the **--execute** option, passing a statement to show the current time:

```
shell> mysql -uroot -p --execute="SELECT CURTIME () "
```

The program connects, executes the statement, and immediately returns to the shell from whence it was invoked:

```
+-----+  
| CURTIME () |  
+-----+  
| 15:09:57 |  
+-----+  
shell>
```

Step 2. Display MySQL query outputs in different formats

1. Invoke the mysql client interactively by using the following shell command (use the password given to you by your instructor):

```
shell> mysql --user=root --password=<password>
```

2. After successfully logging into the mysql client, enter the following statement to retrieve the current date and time in two separate columns:

```
mysql> SELECT CURDATE (), CURTIME ();
```

Returns the date and time in a tabular output format:

```
+-----+-----+  
| CURDATE () | CURTIME () |  
+-----+-----+  
| 2008-02-06 | 15:18:21 |  
+-----+-----+  
1 row in set (#.## sec)
```



3. Repeat the previous step, but this time make the mysql client print vertically, with each column result on a new line:

```
mysql> SELECT CURDATE(), CURTIME() \G
```

We get a similar result, but now each column is printed on its own line:

```
***** 1. row *****
CURDATE(): 2008-02-06
CURTIME(): 15:23:37
1 row in set (#.# sec)
```

Step 3. Dealing with MySQL prompt types

1. Start to enter the following statement:

```
mysql> SELECT Name, Population FROM City WHERE CountryCode = 'IND'
```

Leave out the closing single quote (') and hit the Enter button. Notice that the prompt will change to a waiting-for-a-quote prompt. Enter the cancel terminator and notice that the same prompt will return, still waiting for a single quote. Enter a single quote and hit enter. The prompt now changes to a default waiting prompt. Re-enter the cancel terminator.

The following shows the sequence of events when using the various prompts and terminators;

```
mysql> SELECT Name, Population FROM City WHERE CountryCode = 'IND
      '> \c
      '> '
      -> \c
mysql>
```

2. Using the keyboard editing keys, go back up the command window history to the above SELECT statement and bring back the first line, add the closing single quote and hit Enter. Now continue the statement by adding the last line and hit Enter:

```
mysql> SELECT Name, Population FROM City WHERE CountryCode = 'IND'
      -> AND Population > 3000000;
```

Note: This statement will return an error since there is no database yet loaded. The database will be loaded in a later lab.

Note: The UNIX/Linux behavior is slightly different.



3. Change the prompt to your name followed by a colon (:) by using the `prompt` command:

```
mysql> prompt name:  
PROMPT set to name:
```

Then change it back to the default prompt by entering the `prompt` command without a value:

```
name:prompt  
Returning to default PROMPT of mysql>  
mysql>
```

4. Change the prompt to include the current database and time.

```
mysql> prompt \R:\m \d>\  
PROMPT set to '\R:\m \d>\'
```

Then change it back to the default prompt by entering the `prompt` command without a value:

```
15:45 (none)>prompt  
Returning to default PROMPT of mysql>  
mysql>
```

5. Quit the MySQL client:

```
mysql> \g  
Bye  
shell>
```



65 3.4.5 Using Script Files with mysql

When used interactively, mysql reads queries entered at the keyboard. mysql can also accept input from a file. An input file containing SQL statements to be executed is known as a *script file* or a *SQL batch file*. A script file should be a plain text file containing statements in the same format that would be used to enter the statements interactively. In particular, each statement must end with a terminator.

One way to process a script file is by executing it with a **SOURCE** command from within mysql:

```
mysql> SOURCE input_file
```

NOTE: There are no quotes around the name of the file or semi-colons at the end of the statement.

In this case, mysql will execute the SQL statements in the file and display any output.

The file must be located on the client host where mysql is running. The filename must either be an absolute pathname listing the full name of the file, or a pathname that's specified relative to the directory in which mysql was invoked. For example, if mysql was started on a Windows machine in the C:\mysql directory and this script file to be run is my_commands.sql in the C:\scripts_directory, both of the following **SOURCE** commands tell mysql to execute the SQL statements in the file:

```
mysql> SOURCE C:/scripts/my_commands.sql  
mysql> SOURCE ../scripts/my_commands.sql
```

If a statement in a script file fails with an error, mysql ignores the rest of the file. To execute the entire file regardless of whether errors occur, invoke mysql with the **--force** or **-f** option.

A script file can contain **SOURCE** commands to execute other files, but be careful not to create a **SOURCE** loop. For example, if file1 contains a **SOURCE** file2 command, file2 should not contain a **SOURCE** file1 command.

Which slash should I use?

With Windows, you should use forward slashes (/), because generally back slashes (\) are used to escape special sequences.

66 3.4.6 MySQL Output Formats

By default, mysql produces output in one of two formats, depending on whether it is used in interactive or batch mode:

- When invoked interactively, mysql displays query output in a tabular format that uses bars and dashes to display values lined up in boxed columns.
- When mysql is invoked with a file as its input source on the command line, mysql runs in batch mode with query output displayed using tab characters between data values.



To override the default output format, use these options:

- `--batch` or `-B`: Produce batch mode (tab-delimited) output, even when running interactively.
- `--table` or `-t`: Produce tabular output format, even when running in batch mode.

In batch mode, the `--raw` or `-r` option can be used to suppress conversion of characters such as newline and carriage return to escape-sequences such as `\n` or `\r`. In raw mode, the characters are printed literally.

To select an output format different from either of the default formats, use these options:

- `--html` or `-H`: Produce output in HTML format.
- `--xml` or `-X`: Produce output in XML format.

67

3.4.7 Client Commands and SQL Statements

When an SQL statement is issued while running `mysql`, the program sends the statement to the MySQL server to be executed. `SELECT`, `INSERT`, `UPDATE` and `DELETE` are examples of this type of input. `mysql` also understands a number of its own commands that aren't SQL statements. The `QUIT` and `SOURCE` commands that have already been discussed are examples of `mysql` commands. Another example is `STATUS`, which displays information about the current connection to the server, as well as status information about the server itself. Here is what a status display might look like:

```
mysql> STATUS
-----
mysql Ver 14.14 Distrib 5.1.30, for Win32 (ia32)
Connection id:          2
Current database:       world
Current user:          root@localhost
SSL:                  Not in use
Using delimiter:        ;
Server version:         5.1.30-community MySQL Community Server (GPL)
Protocol version:       10
Connection:             localhost via TCP/IP
...
```

A full list of `mysql` commands can be obtained using the `HELP` command.

In general, `mysql` commands have both a long form and a short form. The long form is a full word (such as `SOURCE`, `STATUS`, or `HELP`). The short form consists of a backslash followed by a single character (such as `\.`, `\s`, or `\h`). The long forms may be given in any lettercase. The short forms are case sensitive.

Unlike SQL statements, `mysql` commands cannot be entered over multiple lines. For example, if a `SOURCE` `input_file` command is issued to execute statements stored in a file, `input_file` must be given on the same line as `SOURCE`. It cannot be entered on the next line.



To log queries and their output, use the `tee` command. All the data displayed on the screen is appended into a given file. This can be very useful for debugging purposes also. This feature can be enabled on the command line with the `--tee` option, or interactively with the `tee` command. The `tee` file can be disabled interactively with the `notee` command. Executing `tee` again re-enables logging. Without a parameter, the previous file is used. Note that `tee` flushes query results to the file after each statement, just before `mysql` prints its next prompt.

68

3.4.8 Using Server-Side Help

The `mysql` program can access server-side help. That is, lookups can be performed in the MySQL Reference Manual for a particular topic, right from the `mysql>` prompt. The general syntax for accessing server-side help is `HELP` keyword. To display the topmost entries of the help system, use the `contents` keyword:

```
mysql> HELP contents
You asked for help about help category: "Contents"
For more information, type 'help <item>', where <item> is one of the
following
categories:
  Account Management
  Administration
  Data Definition
  Data Manipulation
  Data Types
  Functions
  Functions and Modifiers for Use with GROUP BY
  Geographic Features
  Language Structure
  Plugins
  Storage Engines
  Stored Routines
  Table Maintenance
  Transactions
  Triggers
```

69

It is not necessary to step through the items listed in the contents list to get help on a specific subject. Suppose that it is necessary to know how to get status information from the server, but the user cannot remember the command. Typing in the following command yields some hints:

```
mysql> HELP STATUS;
Many help items for your request exist
To make a more specific request, please type 'help <item>',
where <item> is one of the following topics:
  SHOW
  SHOW ENGINE
  SHOW INNODB STATUS
```



To get the more specific information offered, use the **HELP** command with the **SHOW** keyword:

```
mysql> HELP SHOW;
Name: 'SHOW'
Description:
SHOW has many forms that provide information about databases, tables,
columns, or status information about the server. This section describes
those following:

SHOW AUTHORS
SHOW CHARACTER SET [LIKE 'pattern']
SHOW COLLATION [LIKE 'pattern']
...
```

Server-side help requires the help tables in the `mysql` database to be loaded, but normally these files will be loaded by default unless MySQL was installed manually by compiling it.

Using the **HELP** command requires that the help tables in the `mysql` database be initialized with help topic information. This is done by processing the contents of the `fill_help_tables.sql` script, which can be found in the scripts directory. To load the file manually, it is necessary to have the `mysql` database initialized by running `mysql_install_db`, and then process the file with the `mysql` client as follows:

```
shell> mysql -u root mysql < C:\Program Files\MySQL\MySQL Server 5.1
\scripts\fill_help_tables.sql
```

70

3.4.9 Using the Safe Updates Option

It's possible to inadvertently issue statements that modify many rows in a table or that return extremely large result sets. The **--safe-updates** option helps prevent these problems. The option is particularly useful for people who are just learning to use MySQL. **--safe-updates** has the following effects:

- **UPDATE** and **DELETE** statements are allowed only if they include a **WHERE** clause that specifically identifies which records to update or delete by means of a key value, or if they include a **LIMIT** clause.
- Output from single-table **SELECT** statements is restricted to no more than 1,000 rows unless the statement includes a **LIMIT** clause.
- Multiple-table **SELECT** statements are allowed only if MySQL will examine no more than 1,000,000 rows to process the query.

The **--i-am-a-dummy** option is a synonym for **--safe-updates**.





Inline Lab 3-D

This lab requires you to invoke the mysql client interactively as well as in batch mode passing options for several output formats. You will also use a `tee` file and server-side help.

Step 1. Execute a MySQL statement from the shell that produces HTML output

1. Invoke the mysql client non-interactively passing an option to obtain HTML output and have it execute a statement to obtain the current data and time:

```
shell> mysql -uroot -p<password> -H -e "SELECT CURDATE(), CURTIME()"
```

Returns the output for the date and time stamp query in HTML format, instead of the default tabular format;

```
<TABLE BORDER=1><TR><TH>CURDATE () </TH><TH>CURTIME () </TH></TR><TR><TD>2008-02-07</TD><TD>14:56:40</TD></TR></TABLE>
```

And you will be returned to the shell.

```
shell>
```

Step 2. Execute the MySQL client with HTML output

1. Invoke the mysql client in the default manner, but add the output type option for HTML:

```
shell> mysql -uroot --html -p
```

Will put you inside the mysql client and return the usual mysql prompt:

```
mysql>
```

2. Create a new tee file called t1.txt in the C:\ directory to record the rest of the current mysql session:

```
shell> tee C:\t1.txt
```

Shows that you are logging into the tee file;

```
Logging to file 'C:\t1.txt'
```



3. Type the following:

```
mysql> SHOW DATABASES;
```

Note that the output is in HTML format, per the option setting;

```
<TABLE BORDER=1><TR><TH>Database</TH></TR><TR><TD>information_schema</TD></TR><TR><TD>test</TD></TR><TR><TD>world</TD></TR></TABLE>6 rows in set (#.## sec)
```

4. Enter the command to view the server-side help list.

```
mysql> HELP contents;
```

The output lists the available help topics:

```
You asked for help about help category: "Contents"
For more information, type 'help <item>', where <item> is one of the
following categories:
```

```
Account Management
Administration
Compound Statements
Data Definition
Data Manipulation
Data Types
Functions
Functions and Modifiers for Use with GROUP BY
Geographic Features
Language Structure
Plugins
Storage Engines
Table Maintenance
Transactions
User-Defined Functions
Utility
```



5. Enter the command to view the server-side help for the SHOW COLLATION statement:

```
mysql> HELP SHOW COLLATION;
```

A complete description of the command is given, including an example.

```
Name: 'SHOW COLLATION'
Description:
Syntax:
SHOW COLLATION
    [LIKE 'pattern' | WHERE expr]
The output from SHOW COLLATION includes all available character sets.
The LIKE clause, if present, indicates which collation names to match.
The WHERE clause can be given to select rows using more general
conditions, as discussed in
http://dev.mysql.com/doc/refman/5.1/en/extended-show.html. For example:

mysql> SHOW COLLATION LIKE 'latin1%';
+-----+-----+-----+-----+-----+
| Collation      | Charset | Id | Default | Compiled | Sortlen |
+-----+-----+-----+-----+-----+
| latin1_german1_ci | latin1 | 5 |           |           | 0 |
| latin1_swedish_ci | latin1 | 8 | Yes     | Yes     | 0 |
| latin1_danish_ci | latin1 | 15 |          |          | 0 |
| latin1_german2_ci | latin1 | 31 |          | Yes     | 2 |
| latin1_bin        | latin1 | 47 |          | Yes     | 0 |
| latin1_general_ci | latin1 | 48 |          |          | 0 |
| latin1_general_cs | latin1 | 49 |          |          | 0 |
| latin1_spanish_ci | latin1 | 94 |          |          | 0 |
+-----+-----+-----+-----+-----+
URL: http://dev.mysql.com/doc/refman/5.1/en/show-collation.html
```

6. Type the following to disable the tee file to abort recording the session:

```
mysql> note;
```

Message shows that the file has been disabled. The rest of the current mysql session will no longer be recorded in this file;

```
outfile disabled
```

Open the above tee file and confirm the contents. The contents should look like the output shown in the mysql client for steps 2.3 thru 2.5.



71

3.5 MySQL Connectors

MySQL provides several application programming interfaces (APIs) for accessing the MySQL server. Some APIs are used for writing programs and others are simply drivers. MySQL provides several drivers that act as bridges to the MySQL server for client programs that communicate using a particular protocol. These drivers comprise the family of MySQL Connectors. They are available as separate packages.

The MySQL connectors are available for Windows and Unix. To use a connector, it is necessary to install it on the client host. It isn't necessary for the server to be running on the same machine or for the server to be running the same operating system as the client. This means that MySQL connectors are very useful for providing MySQL connectivity in the heterogeneous environments. For example, people who use Windows machines can run client applications that access MySQL databases located on a Linux server host.

Each of these connectors is officially supported by Sun Microsystems, Inc. The following connectors are available:

- C API for MySQL (mysqlclient)
- ODBC (Connector / ODBC)
- JDBC (Connector / J)
- ADO.NET (Connector / NET)
- MXJ (Connector / MXJ)
- PHP (PHP)

For more information and to download the MySQL connectors, see the following MySQL website:

<http://www.mysql.com/products/connector>

3.5.1 Native C

The interface included with distributions of MySQL itself is `libmysqlclient`, the C client library. This API may be used for writing MySQL-based C programs. It is frequently used as basis for other higher-level APIs written for other languages (the Java and .NET interfaces are notable exceptions).

3.5.2 MySQL Connector/ODBC

MySQL Connector/ODBC acts as a bridge between the MySQL server and client programs that use the ODBC standard. It provides a MySQL-specific driver for ODBC so that ODBC-based clients can access MySQL databases. MySQL Connector/ODBC is available for Windows and Unix.

MySQL Connector/ODBC uses the C client library to implement the client/server communication protocol. It converts ODBC calls made by the client program into C API operations that communicate with the server. Connections can be established using TCP/IP, Unix socket files, or named pipes.

For PHP, Sun Microsystems, Inc. has developed the so-called *native driver* or `mysqlnd`. Like Connector/.NET and Connector/J this is a direct implementation of the MySQL client/server protocol, built in C/C++ inside the PHP framework.

This is not a PHP script that implements the protocol – it is still a binary library but it takes advantage of the many utilities and objects provided by the PHP codebase. As such it can be used as a replacement for `libmysqlclient` and currently this is supported for the `mysqli` extension which runs by default on `libmysqlclient`.



3.5.3 MySQL Connector/J

MySQL Connector/J is a JDBC driver that may be used by Java programs. Like ODBC, JDBC is an industry standard. It is not based on the C client library. Instead, it is written in Java and implements MySQL's client/server communication protocol directly. Connections can be established using TCP/IP or named pipes. MySQL Connector/J converts JDBC calls made by the client program into the appropriate protocol operations.

MySQL Connector/J is a Type 4 (pure Java) driver that implements version 3.0 of the JDBC specification. MySQL Connector/J includes support for MySQL capabilities such as server-side prepared statements, stored routines, and Unicode. MySQL Connector/J is available for Windows and Unix. (Note: Unix sockets are not yet supported.)

3.5.4 MySQL Connector/.NET

MySQL Connector/.NET enables .NET applications to use MySQL. It is not based on the C client library. Instead, it is written in C# and implements the client/server communication protocol directly. Connections can be established using TCP/IP, Unix socket files, named pipes, or shared memory.

MySQL Connector/.NET includes support for MySQL capabilities such as server-side prepared statements, stored routines, and Unicode.

MySQL Connector/.NET is available for Windows. If Mono is used, the Open Source implementation of .NET developed by Novell, it is also available on Linux.

3.5.5 MySQL Connector/MXJ

MySQL Connector/MXJ is a Java utility package for deploying and managing a MySQL Server. Connector/MXJ may be bundled with an existing Java application as a "plain old Java object" (POJO), as a Connector/J socket factory, or as a JMX Mbean.

Deploying and using MySQL from Connector/MXJ can be as easy as adding an additional parameter to the JDBC connection url, which will result in the database being started when the first connection is made. This makes it easy for Java developers to deploy applications which require a database by reducing installation barriers for their end-users.

MySQL Connector/MXJ makes the MySQL database appear to be a java-based component. It does this by determining what platform the system is running on, selecting the appropriate binary, and launching the executable. It will also optionally deploy an initial database, with any specified parameters.

As a JMX MBean, MySQL Connector/MXJ requires a JMX v1.2 compliant MBean container, such as JBoss version 4. The MBean will use the standard JMX management APIs to present (and allow the setting of) parameters which are appropriate for that platform.

Currently supported platforms include Linux (x86), Microsoft Windows (x86), Mac OS X 10.3, and Sun Solaris (SPARC).



72

3.6 Third-Party API's

Each of the preceding connectors is officially supported by Sun Microsystems, Inc. In addition, many third-party client interfaces are available. Most of them are based on the C client library and provide a binding for some other language. These include the mysql and mysqli extensions for the following:

- PHP
 - C/C++
 - Tcl
 - Perl DBI (DBD::mysql)
 - Python
 - Eiffel
 - ODBC
 - Ruby
 - Pascal

For the convenience of its users, the MySQL web site has more information and web site links for some of these client API's on our developers page:

<http://dev.mysql.com/>

Although members of the Sun Microsystems, Inc. development team often work closely with the developers of these products, the APIs do not receive official support from Sun Microsystems, Inc.. If a user is embarking on a project that involves these APIs, they should contact the developers to determine whether future support will be available.

Correct formatting

The biggest concern with migration concerning APIs is to ensure that the request sent to the server (through the API) is formatted correctly for the server version that is being used.





Quiz

In this exercise you will answer questions pertaining to MySQL Client Interfaces.

NOTE: Choose the best answer for each question.

1. Where does MySQL Connector/ODBC have to be installed?
 - a) Connector/ODBC must be installed on every client host where programs run that should use that connector.
 - b) Connector/ODBC has to be installed on the server host only. This will make it available for all clients' connection from remote hosts.
 - c) Connector/ODBC has to be installed both on the server host and on all client hosts.
2. Which of the following statements is true?
 - a) MySQL Connectors are available for all operating systems that MySQL supports.
 - b) MySQL Connectors are shipped together with MySQL server distributions.
 - c) MySQL Connectors are shipped with the MySQL GUI tools.
 - d) MySQL Connectors are shipped separately from MySQL server distributions by Sun Microsystems, Inc.
 - e) MySQL Connectors are shipped separately from MySQL server distributions by third parties.
3. MySQL Connector/.NET runs on Windows only (True or False)
4. All MySQL Connectors are based on MySQL's C API and are implemented using the MySQL C client library. (True or False)
5. Is the following statement true? Like all other MySQL programs, MySQL Connectors are written in C.(True or False)



73

3.7 Chapter Summary

In this chapter, you learned to:

- Invoke client programs within the MySQL Client/Server architecture
- Use many of the features of the `mysql` client
- Download and use the MySQL Query Browser to connect to the MySQL Server
- Describe the client interfaces provided by Sun Microsystems, Inc.
- Distinguish between the client interfaces and choose according to need
- Use MySQL website for downloading the MySQL client interface programs
- Understand the relationship to third-party client interfaces



4 QUERYING FOR TABLE DATA

75

4.1 Learning Objectives

This chapter introduces simple MySQL queries. In this chapter, you will learn to:

- Execute basic, single table queries using the **SELECT** syntax
- Aggregating query results
- Use the **UNION** keyword to concatenate results from multiple **SELECT** statements



76

4.2 The SELECT Statement

The **SELECT** statement is primarily used to retrieve data from one or more tables in a database. As such, the **SELECT** statement belongs to the category of DML-statements (Data Manipulation Language). The **SELECT** statement is probably the most commonly used DML statement.

A **SELECT** statement denotes a particular set of rows from the database. When the database server executes a **SELECT** statement, the corresponding, rows of data are retrieved and then returned to the client. The rows thus obtained are called a *result set*.

The **SELECT** statement is constructed from a number of clauses that specify how and what data to retrieve. The basic syntax model is shown below:

```
SELECT [<clause options>] <column list>
[FROM <table>]
[<more options>]
```

77

Basic **SELECT** statement example using **world** database (and the result data):

```
mysql> SELECT Name FROM Country;
+-----+
| Name           |
+-----+
| Afghanistan   |
| :             |
| United States Minor Outlying Islands |
+-----+
239 rows in set (#.# sec)
```

The above example shows a typical query, which retrieves all rows from a specific table, reporting the value of one particular column.

The column list of the **SELECT** statement need not be based on table columns – expressions such as arithmetic calculations may appear in the column list too. In fact, the select statement need not even based on a database table. The following example illustrates how the **SELECT** statement is used to perform a simple calculation.

```
mysql> SELECT 1+2;
+---+
| 1+2 |
+---+
| 3 |
+---+
1 row in set (#.# sec)
```

The column list may contain multiple expressions such as columns or calculations as long as these are separated by a comma.





InLine Lab 4-A

In this exercise you will use the **SELECT . . . FROM** syntax using the **world** database. This will require a mysql command line client and access to a MySQL server.

Step 1. Retrieving a single column

- Issue a **SELECT** statement which will retrieve the **Continent** column data from the **Country** table:

```
mysql> SELECT Continent FROM Country;
```

Will return a list 239 continents, one for each country. (Notice that the list does not show the name of the country and many continents are repeated several times.);

```
+-----+  
| Continent |  
+-----+  
| Asia      |  
| Europe    |  
| North America |  
...  
...
```

Step 2. Retrieving multiple columns

- Issue the same **SELECT** statement and add the country **Name** to the column list , using a comma separator:

```
mysql> SELECT Continent, Name FROM Country;
```

You will notice that the same number of records is returned, but now the result is two columns with the **Name** and **Continent** headings;

```
+-----+-----+  
| Continent | Name          |  
+-----+-----+  
| Asia      | Afghanistan   |  
| Europe    | Netherlands  |  
| North America | Netherlands Antilles |  
...  
...
```



78

4.2.1 Basic Uses of SELECT

There are many clauses, and combinations thereof, that can be used with a **SELECT** statement to yield particular query results. They range from very basic, commonly-used options to very specialized and complex. The following basic, optional clauses will be covered in this section:

- **FROM**: Specifies from which tables the data is to be fetched
- **DISTINCT**: Eliminate duplicate rows
- **WHERE**: Only return rows that satisfy a particular condition (a filter)
- **ORDER BY**: Sorts rows according to a list of expressions
- **LIMIT**: Return a particular portion rather than the entire result set

Example of the **SELECT** statement with a number of clauses:

```
SELECT DISTINCT <expressionlist>
  FROM      <table-specification>
 WHERE     <condition>
 ORDER BY  <ordering-specification>
 LIMIT     <rowcount>
```

The above syntax shows the correct order and usage of each of the above optional clauses. This statement is specifically selecting only unique rows from a the specified tables, where the condition filter for only specific rows. The rows are then ordered as specified, and finally the number of rows is limited to the specified amount.

Order is important

If these clauses are used, they must be used in the order as shown here. It is a syntax error to use these clauses in a different order. In addition, the keywords that identify a particular clause or option are *not* case-sensitive. However, it is the convention to write these using upper case characters. Remember that table names may be case-sensitive

79

4.2.2 Using **FROM**

We have encountered the **FROM** clause already in the introduction of the **SELECT** statement. The **FROM** clause is an optional element of the **SELECT** statement that may appear immediately after the expressionlist that appears after the **SELECT** keyword.

The **FROM** clause specifies a table which is processed by the statement. The way in which the table is processed is dependent upon the type of statement wherein the **FROM** clause appears: In **SELECT** statements, the **FROM** clause denotes a table from which data can be retrieved. There are other types of SQL statements for the addition, removal or modification of data, and these process the **FROM** clause in their respective way.

Although the **FROM** clause specifies a single table, the specification may consist of multiple elements. For now we will only discuss those cases where the **FROM** clause specifies a single database table (also known as a *base table*). More complex **FROM** clauses can be ignored for now, and will be discussed in great detail later on in this course.



80

Unqualified table name

In the simplest case, the **FROM** keyword is followed by a single table name:

```
FROM <table-name>
```

In this case the table name refers to an actual table in the current default database. So, for example, the current **FROM** clause may be used to process rows from the **Country** table, provided the current default database is the **world** database:

```
FROM Country
```

Qualified table name

Table names in the from clause may always be explicitly *qualified* by explicitly prepending the name of database wherein the table resides, separating the database name and the table name with a dot (period):

```
FROM <database-name>. <table-name>
```

For example, to explicitly qualify the **Country** table from the **world** database, the following can be used:

```
FROM world.Country
```

In this case, the qualified table name **world.Country** can always be used to unambiguously identify the specified table (namely, the table with the name 'Country' that resides in the database called 'world'). This works even if the current default database is not the database wherein the table resides.

Table aliases

Within a SQL statement, a table reference in the **FROM** clause can be given a temporary name. This so-called *table alias* can then be used within the current statement.



81

4.2.3 Using DISTINCT

If a query returns a result that contains duplicate rows, the duplicates can be removed to produce a result set in which every row is unique. To do this, include the **DISTINCT** keyword after the **SELECT** keyword (but before the list of column expressions). **DISTINCT** will compare whole rows when processing.

In the case below, the query results in a set that includes duplicate rows:

```
mysql> SELECT Continent FROM Country;
+-----+
| Continent      |
+-----+
| Asia           |
| Europe          |
| North America   |
| Europe          |
| Africa          |
| Oceania         |
| :               |
| Antarctica     |
| Oceania         |
+-----+
239 rows in set (#.## sec)
```

When the keyword **DISTINCT** is added to the statement, it eliminates the duplicate rows from the result, returning only unique rows:

```
mysql> SELECT DISTINCT Continent FROM Country;
+-----+
| Continent      |
+-----+
| Asia           |
| Europe          |
| North America   |
| Africa          |
| Oceania         |
| South America   |
| Antarctica     |
+-----+
7 rows in set (#.## sec)
```



82

DISTINCT treats all **NULL** values within a given column as having the same value. Suppose that a table t contains the following rows:

```
mysql> SELECT i, j FROM t;
+----+----+
| i | j |
+----+----+
| 1 | 2 |
| 1 | NULL |
| 1 | NULL |
+----+----+
```

For purposes of **DISTINCT**, the **NULL** values in the second column are the same, so the second and third rows are identical. Adding **DISTINCT** to the query eliminates one of them as a duplicate:

```
mysql> SELECT DISTINCT i, j FROM t;
+----+----+
| i | j |
+----+----+
| 1 | 2 |
| 1 | NULL |
+----+----+
```





InLine Lab 4-B

In this exercise you will use the **world** database to issue a **SELECT** statement with the **DISTINCT** keyword. This will require a mysql command line client and access to a MySQL server.

Step 1. Get a result set with duplicates

1. Issue a **SELECT** statement which will retrieve all the Region column data from the **Country** table:

```
mysql> SELECT Region FROM Country;
```

Return a list 239 regions, one for each country. (Notice that many regions are repeated several times.)

Step 2. Eliminate duplicates with **DISTINCT**

1. Issue a **SELECT** statement which will retrieve only the distinct values for the Region column from the **Country** table:

```
mysql> SELECT DISTINCT Region FROM Country;
```

Will return a list 25 distinct regions. (Notice that no regions are repeated.)



83

4.2.4 Using WHERE

The **WHERE** clause filters for rows that satisfy a particular *condition*. The condition is an expression that is applied for each row in the intermediate result set, and when the expression evaluates to TRUE, the row is retained. If the condition does not evaluate to TRUE, the row is discarded and will not be processed any further.

In the statement below, the query is retrieving three columns from the city table. The **WHERE** clause specifies a condition to retain only those rows where the value in the Name column equals 'New York'. In other words, it is answering the question: "What is the ID and district for the city called New York?":

```
mysql> SELECT ID, Name, District FROM city WHERE Name = 'New York';
+----+-----+-----+
| ID | Name | District |
+----+-----+-----+
| 3793 | New York | New York |
+----+-----+-----+
1 row in set (#.## sec)
```

The condition following the **WHERE** keyword is a special case of an *expression*. For practical purposes, it is usually sufficient to consider anything that has a (scalar) value to be an expression. This includes literals like 'Hello', 4, built-in constants like **TRUE**, **FALSE** and even **NULL**, references to columns, functions etc.

84

Expressions can be built using a number of *operators*. Operators are just a special class of expressions which have the effect of taking a number of value-expressions called the *operands* as input, transforming them in a particular way to yield a new value.

NOTE: Operators are not all that different from functions – the difference is mainly syntactical.

Value-transformations performed by operators tend to be more fundamental in nature; that is, operators are syntactical shorthands for the most commonly used value-transformations.

MySQL supports a number of different types of operators, depending upon the data types of the operand or operands and the data type of the value to which the operator evaluates.

- Arithmetic – treats the operands as numerical values and performs some calculation, returning another numerical value
- Comparison – Compares operands of no particular data type and returns a boolean value to indicate how the operands return to each other. Sometimes these are called *relational* operators as the evaluation asserts a particular relationship between the operands, for example that one operand is larger than the other operand. Note that the term relational in this context has got nothing to do with the term relational in “relational databases”
- Logical – to transform boolean expressions into another single boolean expression. Logical operators have boolean operands and evaluate to a boolean value. We just explained that the condition that follows the **WHERE** keyword is a boolean expression. In many cases, this boolean expression is composed of other, smaller boolean expressions that are combined using these logical operators.



Table aliases

- `x + y` Addition; `4 + 3` evaluates to 7
- `x - y` Subtraction; `4 - 3` evaluates to 1
- `x * y` Multiplication; `4 * 3` evaluates to 12
- `x / y` Division; `4 / 3` evaluates to 1.3333
- `x DIV y` Integer Division, `4 DIV 3` evaluates to 1
- `x % y` Modulo (remainder after division); `4 / 3` evaluates to 1

Comparison Operators

- `x = y` Equal to, TRUE if `x` is the same as `y`
- `x <= y` `NULL`-safe equal to, same as `=` but considers `NULL` values to be comparable like any other value
- `x < y` Less than; TRUE if `x` is less than `y`
- `x <= y` Less than or equal to; TRUE if `x` is less than or equal to `y`
- `x > y` Greater than; TRUE if `x` is greater than `y`
- `x >= y` Greater than or equal to; TRUE if `x` is greater than or equal to `y`
- `x <> y` Not equal to; TRUE if `x` and `y` are not equal
- `x != y` Not equal to; alternative notation for `<>`
- `x BETWEEN y AND z` Range comparison, TRUE if `x` is greater than or equal to `y` and less than or equal to `z`
- `x IS NULL` TRUE if `x` is the `NULL` value.
- `x IS NOT NULL` TRUE if `x` is not the `NULL` value.
- `x IN (y1, ..., yN)` TRUE if `x` equals either of the expressions `y1` through `yN`

Logical Operators

- `x AND y` Logical AND, TRUE if both `x` and `y` are TRUE
- `x OR y` Logical OR, TRUE if `x`, `y`, or both `x` and `y` is TRUE
- `x XOR y` Logical exclusive-OR, TRUE if either `x` or `y` but not both `x` and `y` is TRUE
- `NOT y` Logical negation; TRUE if `x` is FALSE



85

Examples

The example below uses a few of the operators and techniques mentioned above (on the **Country** table):

```
mysql> SELECT Name, Population
-> FROM Country
-> WHERE Population > 50000000 AND
-> (Continent = 'Europe' OR Code ='USA');
+-----+-----+
| Name           | Population |
+-----+-----+
| United Kingdom |      59623400 |
| Italy          |      57680000 |
| France         |      59225700 |
| Germany        |      82164700 |
| Ukraine        |      50456000 |
| Russian Federation | 146934000 |
| United States   |     278357000 |
+-----+-----+
7 rows in set (#.# sec)
```

86

This following example demonstrates the use of the **IN** operator in a **WHERE** clause:

```
mysql> SELECT ID, Name, District
-> FROM city
-> WHERE Name IN ('New York', 'Rochester', 'Syracuse');
+-----+-----+-----+
| ID   | Name    | District |
+-----+-----+-----+
| 3793 | New York | New York |
| 3871 | Rochester | New York |
| 3935 | Syracuse | New York |
+-----+-----+-----+
3 rows in set (#.# sec)
```



87

A word about NULL's

Most operators evaluate to **NULL** if one of the operands evaluates to **NULL**. (There are a few exceptions to this rule, and the operators to which this exception applies are created especially with the purpose of a specific handling of **NULL** in mind.) The reason is that **NULL** is a value-expression for which it is stipulated that it is unknown to which other value it is equal to. For this reason, the evaluation of an operator that has to operate on such a value leads to another unknown value.

The handling of **NULL** gets a special twist when dealing with logical and relational operators. As we just discussed, these operators evaluate to a boolean value, that is, they are TRUE or FALSE upon evaluation. Or are they?

Whenever a **NULL**-value expression is used as operand to these functions, the operator evaluates to **NULL** as is the case for other expressions. For that reason, the relational and logical operators are defined in a manner that describes under what circumstances they will return TRUE. If they do not return TRUE, it does not automatically follow that they return FALSE, as they may evaluate to either FALSE or **NULL**. For a similar reason, the operands to logical functions are discussed from the perspective that they have a definitive value: TRUE or FALSE.

88

The same applies to the **WHERE** clause. The **WHERE** clause has the effect of discarding those rows for which the condition does not hold TRUE. That is not to say that the condition is FALSE for the rows that are discarded: it may very well be the case that the condition evaluates to **NULL**.





InLine Lab 4-C

In this exercise you will use the **world** database to practice the **WHERE** clause in a **SELECT** statement. This will require a mysql command line client and access to a MySQL server.

Step 1. Using equals in a WHERE condition

1. Issue a **SELECT** statement that includes a **WHERE** clause to retrieve all columns from the **City** table where the value in the **ID** column equals the number 3875:

```
mysql> SELECT * FROM City WHERE ID = 3875;
```

Will return a single row of data associated with the specified id. Hint: Use the * symbol to indicate that you want to retrieve all columns.

Step 2. Using less than in a WHERE condition

1. Issue a **SELECT** statement with a **WHERE** clause to retrieve the names from all countries that have a population less than 1000:

```
mysql> SELECT Name FROM Country WHERE Population < 1000;
```

Returns the names of the countries that have a population less than 1000.



89

4.2.5 Using ORDER BY in SELECT statements

By default, the rows in the result set produced by a **SELECT** statement are returned by the server to the client in no particular order. When a query is issued, the server is free to return the rows in any convenient order. This order can be affected by factors such as the order in which the rows are actually stored in the table, or which indexes are used to process the query. If it is necessary for the output rows to be returned in a specific order, include an **ORDER BY** clause that indicates how to sort the results.

What is important to remember about record retrieval order is this: There is no guarantee about the order in which the server returns rows, unless the order is explicitly specified. To do so, add an **ORDER BY** clause to the statement that defines the sort order that is required.

The following example returns country names (in the **Country** table of the **world** database) alphabetically by name;

```
mysql> SELECT Name FROM Country ORDER BY Name;
+-----+
| Name           |
+-----+
| Afghanistan   |
| Albania        |
| Algeria        |
...
...
```

90

By default, rows are sorted by ascending order of the value of the expressions specified in the **ORDER BY** clause. It is possible to specify explicitly whether to sort a column in ascending or descending order by using the **ASC** or **DESC** keywords after an individual **ORDER BY** expression. Ascending means that values are sorted from low to high; descending order is the opposite direction, from high to low.

The statement below sorts the names of the countries in order by descending alphabetical order;

```
mysql> SELECT Name
    -> FROM Country
    -> ORDER BY Name DESC;
+-----+
| Name           |
+-----+
| Zimbabwe      |
| Zambia         |
| Yugoslavia    |
| Yemen          |
| Western Sahara|
| Wallis and Futuna |
| Virgin Islands, U.S. |
...
...
```



91

It is also possible to sort for multiple expressions simultaneously, each having their own individual sorting order. Multiple expressions in the **ORDER BY** clause must be separated by a comma, just like expressions in the **SELECT** list. This shown in the following example;

```
mysql> SELECT Name, Continent
-> FROM Country
-> ORDER BY Continent DESC, Name ASC;
+-----+-----+
| Name      | Continent   |
+-----+-----+
| Argentina | South America |
| Bolivia   | South America |
| :          | :           |
| Uzbekistan | Asia         |
| Vietnam    | Asia         |
| Yemen     | Asia         |
+-----+-----+
239 rows in set (#.## sec)
```

ENUM data type ordering

In the above example, the **Continent** column is data type **ENUM**, therefore it is ordered by number of placement in list, not alphabetically.





InLine Lab 4-D

In this exercise you will be using the **world** database to practice the **ORDER BY** clause in a **SELECT** statement. This will require a mysql command line client and access to a MySQL server.

Step 1. Using a single ORDER BY expression

1. Issue a **SELECT** statement with an **ORDER BY** clause to retrieve the name of all cities from the **City** table sorted by name:

```
mysql> SELECT Name FROM City ORDER BY Name;
```

Returns a list of all city names in descending alphabetical order.

Step 2. Using WHERE and ORDER BY

1. Issue a **SELECT** statement to retrieve the **CountryCode** and **Language** from the **CountryLanguage** table, using a **WHERE** clause to retrieve only those rows where the language is Swedish and an **ORDER BY** clause to sort by **CountryCode** in descending order:

```
mysql> SELECT CountryCode, Language
-> FROM CountryLanguage
-> WHERE Language = 'Swedish'
-> ORDER BY CountryCode DESC;
```

Returns a two column list of country codes and corresponding languages in descending alphabetical order by country codes;

CountryCode	Language
SWE	Swedish
NOR	Swedish
FIN	Swedish
DNK	Swedish

4 rows in set (#.## sec)

Step 3. More WHERE and ORDER BY

1. Issue a **SELECT** statement which will retrieve the name of each country. Use an **ORDER BY** clause to sort the result in descending order by population:

```
mysql> SELECT Name FROM Country ORDER BY Population DESC;
```

Lists all countries in the world, with the largest countries (population-wise) first.



Using LIMIT in SELECT statements

92

When a query returns many rows, and there is a need to see only a few of them, add a **LIMIT** clause. This is a MySQL option that allows one to return only a particular portion of the result set instead of the entire result set.

The **LIMIT** clause may be given with either one or two arguments:

- **LIMIT row_count**
- **LIMIT skip_count, row_count**

Each argument must be represented as an integer constant. Variable expressions (like user variables, column references, calculations, other queries, etc.) can not be used. **LIMIT** is especially useful in conjunction with the **ORDER BY** clause, to put the rows in a particular order. When they are used together, MySQL applies **ORDER BY** first and then **LIMIT**.

When followed by a single integer constant, it is interpreted as the *row_count*. **LIMIT** will then return the first *row_count* rows from the result set. For example, **LIMIT 3** will return only the first 3 rows;

```
mysql> SELECT Name FROM Country ORDER BY Name LIMIT 3;
+-----+
| Name |
+-----+
| Afghanistan |
| Albania |
| Algeria |
+-----+
3 rows in set (#.# sec)
```

93

When followed by two integer constants, these are interpreted as the *skip_count* and *row_count* respectively. The **LIMIT** clause will then skip the first *skip_count* rows from the beginning of the result set and then return the next *row_count* rows. To skip the first 2 rows of a result set and then retrieve the next 3, use the following;

```
mysql> SELECT Name FROM Country LIMIT 2, 3;
+-----+
| Name |
+-----+
| Algeria |
| American Samoa |
| Andorra |
+-----+
3 rows in set (#.# sec)
```



94

One common use for this is to find the row containing the smallest or largest values in a particular column. For example, to find the country with the smallest surface area, do:

```
mysql> SELECT * FROM Country ORDER BY SurfaceArea LIMIT 1;
```

Another example: the following query selects the top 5 largest countries measured by populations:

```
mysql> SELECT Name, Population
      -> FROM Country
      -> ORDER BY Population DESC LIMIT 5;
+-----+-----+
| Name        | Population |
+-----+-----+
| China       | 1277558000 |
| India       | 1013662000 |
| United States | 278357000 |
| Indonesia   | 212107000 |
| Brazil       | 170115000 |
+-----+-----+
5 rows in set (#.# sec)
```





InLine Lab 4-E

In this exercise you will use the **world** database to practice using the **LIMIT** clause as part of a **SELECT** statement. This will require a mysql command line client and access to a MySQL server.

Step 1. Using LIMIT and ORDER BY

1. Issue a **SELECT** statement which to retrieve the name of the cities in the **City** table, sorted in ascending order by name and using the **LIMIT** clause to limit the result set to a maximum of 5 rows:

```
mysql> SELECT Name  
-> FROM City  
-> ORDER BY Name  
-> LIMIT 5;
```

Returns a list of 5 city names in ascending alphabetical order;

```
+-----+  
| Name |  
+-----+  
| A Coruña (La Coruña) |  
| Aachen |  
| Aalborg |  
| Aba |  
| Abadan |  
+-----+  
5 rows in set (#.# sec)
```

Step 2. More LIMIT and ORDER BY

1. Issue a **SELECT** statement which to retrieve five largest cities in the world (by population):

```
mysql> SELECT Name, Population  
-> FROM City  
-> ORDER BY Population DESC  
-> LIMIT 5;
```

Returns five largest cities: Mumbai, Seoul, São Paulo, Shanghai and Jakarta.



Step 3. More ORDER BY and LIMIT

1. Issue a **SELECT** statement to retrieve all columns for countries where the **GNP** (Gross National Products) is greater than the old **GNP**, by order of country name, limiting the result to 3 rows. Use the **\G** terminator to get a more readable result.

Returns all **Country** table data for three countries (3 rows, only first row shown below):

```
mysql> SELECT *
-> FROM Country
-> WHERE GNP > GNPOld
-> ORDER BY Name
-> LIMIT 3\G

***** 1. row *****
Code: ALB
Name: Albania
Continent: Europe
Region: Southern Europe
SurfaceArea: 28748.00
IndepYear: 1912
Population: 3401200
LifeExpectancy: 71.6
GNP: 3205.00
GNPOld: 2500.00
LocalName: Shqipëria
GovernmentForm: Republic
HeadOfState: Rexhep Mejdani
Capital: 34
Code2: AL
...
```



95

4.3 Aggregating Query Results

So far, we have been doing simple **SELECT** statements where each row in the result set corresponded to one row in the underlying table. However, it is possible to obtain a situation where a row in the result set corresponds to a *group* of rows from the underlying table. This process is called *aggregation*, and such a result set is called an *aggregate*. SQL allows us to apply a function over the individual rows that make up the aggregate, and these functions are called *aggregate functions*.

4.3.1 Why Use Aggregate Functions?

A **SELECT** statement can produce a list of rows that match a given set of conditions. This list provides the details about the selected rows. However, if it is necessary to know about the overall characteristics of the rows, a summary may be more useful. Aggregate functions (also known as Summary functions) perform summary operations on a set of values, such as counting, averaging, or finding minimum or maximum values. They calculate a single value based on a group of values.

Usually, aggregate functions do not take **NULL** values into account, excluding these from the calculation (rather than returning **NULL**). There are exceptions to this rule: **COUNT(*)** counts all rows, and **GROUP_CONCAT** returns **NULL** as a whole in case a **NULL** value happened to occur in a particular row.

Functions such as **AVG()** that calculate summary values for groups are known as “aggregate” functions because they are based on aggregates or groups of values. There are several aggregate functions. Some of the most common are as follows;

MIN()	Find the smallest value
MAX()	Find the largest value
SUM()	Summarize numeric value totals
AVG()	Summarize numeric value averages
STD()	Returns the standard deviation
COUNT()	Counts rows, non-null values, or the number of distinct values
GROUP_CONCAT()	Concatenates a set of strings to produce a single string

All the functions can be used with the **DISTINCT** keyword (although it's useless for **MAX()** and **MIN()**). The **DISTINCT** keyword appears directly after the left parenthesis, and ensures that each unique value is taken into account by the calculation exactly once.



96

In the following example the **COUNT(*)** function is being used to get a count of all rows in the **Country** table of the **world** database;

```
mysql> SELECT COUNT(*) FROM Country;
+-----+
| COUNT(*) |
+-----+
|      239 |
+-----+
1 row in set (#.## sec)
```

Now specifying a count of Capital yields a different result, due to the fact that not every country has a Capital associated with it: **NULL** values are not included in the count;

```
mysql> SELECT COUNT(Capital) FROM Country;
+-----+
| COUNT(Capital) |
+-----+
|        232 |
+-----+
1 row in set (#.## sec)
```

97

4.3.2 Grouping with **SELECT** and **GROUP BY**

We just showed that adding an aggregate function groups all rows into one single aggregate row. There is another construct that allows rows to be grouped, namely the **GROUP BY** clause. Like the **ORDER BY** clause, the **GROUP BY** clause takes on a comma-separated list of expressions. If present, all rows that have the same combination of values for the expressions specified in the **GROUP BY** clause, are treated as one group, and end up as one row in the result set.

The **GROUP BY** clause is applied after the **WHERE** clause, but before the **ORDER BY** clause.

An aggregate function may be used with or without a **GROUP BY** clause that places rows into groups. Without a **GROUP BY** clause, it calculates a summary value based upon the entire set of selected rows. (That is, MySQL treats all rows as a single group.) With a **GROUP BY** clause, an aggregate function calculates a summary value for each group. For example, if a **WHERE** clause selects 20 rows and the **GROUP BY** arranges them into four groups of five rows each, a summary function produces a value for each of the four groups.



Grouping can be based on the values in one or more columns of the selected rows. For example, the **Country** table (**world** database) indicates which continent each country is part of, so it is possible to group the records by continent and calculate the average population of countries in each continent:

```
mysql> SELECT Continent, AVG(Population)
-> FROM Country
-> GROUP BY Continent;
+-----+-----+
| Continent | AVG(Population) |
+-----+-----+
| Asia      |    72647562.7451 |
| Europe    |   15871186.9565 |
| North America | 13053864.8649 |
| Africa    |   13525431.0345 |
| Oceania    |   1085755.3571 |
| Antarctica |       0.0000 |
| South America | 24698571.4286 |
+-----+-----+
7 rows in set (#.# sec)
```

Like in the **ORDER BY**, any expression can appear in the **GROUP BY** list, not just column references.

98

GROUP BY with GROUP_CONCAT()

The following example combines the **GROUP_CONCAT()** function with **GROUP BY**. The **GROUP_CONCAT()** function produces a concatenated result from each group of strings. In the following example, it creates lists of the countries that have a particular form of government on the South American continent:

```
mysql> SELECT GovernmentForm, GROUP_CONCAT(Name) AS Countries
-> FROM Country
-> WHERE Continent = 'South America'
-> GROUP BY GovernmentForm\G
***** 1. row *****
GovernmentForm: Dependent Territory of the UK
  Countries: Falkland Islands
***** 2. row *****
GovernmentForm: Federal Republic
  Countries: Argentina,Venezuela,Brazil
***** 3. row *****
GovernmentForm: Overseas Department of France
  Countries: French Guiana
***** 4. row *****
GovernmentForm: Republic
  Countries: Chile,Uruguay,Suriname,Peru,Paraguay,Bolivia,Guyana,
Ecuador,Colombia
4 rows in set (#.# sec)
```



99

GROUP BY with WITH ROLLUP

The **WITH ROLLUP** modifier can be used in the **GROUP BY** clause to produce multiple levels of summary values. Suppose that it is necessary to generate a listing of the population of each continent, as well as the total of the population on all the continents. One way to do this is by running one query to get the per-continent totals and another to get the total for all continents. Another way to get the same results is to use **WITH ROLLUP**. This enables the use of a single query to get both the detailed results as well as the total sum of all rows, eliminating the need for multiple queries or extra processing on the client side:

```
mysql> SELECT Continent, SUM(Population) AS pop
      -> FROM Country
      -> GROUP BY Continent
      -> WITH ROLLUP;
+-----+-----+
| Continent | pop   |
+-----+-----+
| Asia      | 3705025700 |
| Europe    | 730074600  |
| North America | 482993000 |
| Africa    | 784475000  |
| Oceania   | 30401150   |
| Antarctica | 0      |
| South America | 345780000 |
| NULL      | 6078749450 |
+-----+-----+
8 rows in set (#.## sec)
```

The difference in the output from this statement compared to one without **WITH ROLLUP** occurs in the last line, where the **Continent** value contains **NULL** and the **pop** value contains the total sum (grand total) of all population values.

The last row in the previous result is special as it does not correspond directly to a particular group of rows as defined by the **GROUP BY** clause. Rather, it is a row that is generated as a result of an additional application of the **SUM()** function on all individual rows, representing the group of all rows. **WITH ROLLUP** performs a *super-aggregate* operation. It does not simply generate a sum of the numbers that appear in the **pop** column. Instead, the final line comprises applications of the given aggregate function, as it is written in the **SELECT** clause, on every single row selected.



100

To illustrate this, consider the following example in which we calculate columns using the **AVG()** function rather than **SUM()**. The final roll-up line contains the overall average (and not the sum of averages or the average of averages):

```
mysql> SELECT Continent, AVG(Population) AS avg_pop
-> FROM Country
-> GROUP BY Continent WITH ROLLUP;
+-----+-----+
| Continent | avg_pop |
+-----+-----+
| Asia      | 72647562.7451 |
| Europe    | 15871186.9565 |
| North America | 13053864.8649 |
| Africa    | 13525431.0345 |
| Oceania   | 1085755.3571 |
| Antarctica | 0.0000 |
| South America | 24698571.4286 |
|           | 25434098.1172 |
+-----+-----+
8 rows in set (#.## sec)
```

101

HAVING

The **HAVING** clause can be used to eliminate rows based on aggregated values. It can be used with or without **GROUP BY**. The **HAVING** clause should be used only when there is a need to apply a condition that refers to an aggregated function. If the condition does not refer to an aggregate function, the condition should be handled in the **WHERE** clause, not in the **HAVING** clause.

The following statement results in a list of continents whose sum (aggregate value) of country populations is greater than 100,000,000;

```
mysql> SELECT Continent, SUM(Population) AS pop
-> FROM Country
-> GROUP BY Continent
-> HAVING SUM(Population) > 100000000;
+-----+-----+
| Continent | pop    |
+-----+-----+
| Asia      | 3705025700 |
| Europe    | 730074600  |
| North America | 482993000 |
| Africa    | 784475000  |
| South America | 345780000 |
+-----+-----+
5 rows in set (#.## sec)
```





InLine Lab 4-F

In this exercise you will use the **world** database to practice the **GROUP BY** clause in a **SELECT** statement. This will require a mysql command line client and access to a MySQL server.

Step 1. Using GROUP BY and COUNT(*)

1. Issue a **SELECT** statement with a **GROUP BY** clause to retrieve the name of all continents from the **Country** table, and the number of countries per continent:

```
mysql> SELECT Continent, COUNT(*) FROM Country GROUP BY Continent;
```

Returns a list of all 7 continents and their corresponding country count;

```
+-----+-----+
| continent | count(*) |
+-----+-----+
| Asia      |      51 |
| Europe    |      46 |
| North America | 37 |
| Africa    |      58 |
| Oceania   |      28 |
| Antarctica |      5 |
| South America | 14 |
+-----+-----+
7 rows in set (#.## sec)
```

Step 2. Using GROUP BY and GROUP_CONCAT

1. Issue a **SELECT** statement with a **GROUP BY** clause to list the names of all the continents from the **Country** table, and use **GROUP_CONCAT()** to list their corresponding countries:

```
mysql> SELECT Continent, GROUP_CONCAT(Name) FROM Country
-> GROUP BY Continent\G
```

Returns a list of all 7 continents and their corresponding countries;

```
***** 1. row *****
continent: Asia
group_concat(name):Japan,Cyprus,Laos,Lebanon,Georgia,Macao,Philippines,Maldives,Malaysia,Mong...
```



Step 3. Using GROUP BY and WITH ROLLUP

- Issue a **SELECT** statement and use **GROUP BY** to list the names of all the continents from the **Country** table, calculating the sum of the continent's countries **GNP** and using **WITH ROLLUP** to calculate the total number of countries:

```
mysql> SELECT Continent, SUM(GNP) FROM Country
-> GROUP BY Continent WITH ROLLUP;
```

Returns a list of all 7 continents and their corresponding country **GNP** totals;

Continent	SUM(GNP)
Asia	7655392.00
Europe	9498865.00
...	
South America	1511874.00
NULL	29354907.90

8 rows in set (#.# sec)

Step 4. Using GROUP BY and HAVING

- Issue a **SELECT** statement and use **GROUP BY** to make groups of districts. Use **HAVING** to look for those districts that have at least 30 or more cities:

```
mysql> SELECT District, COUNT(*) AS NumCities FROM City GROUP BY District
-> HAVING NumCities >= 30;
```

Returns a list of 12 districts and their corresponding city counts;

District	NumCities
Andhra Pradesh	35
Buenos Aires	31
California	70
England	71
...	
Shandong	32
Southern Tagalog	31
Uttar Pradesh	43
West Bengali	46

12 rows in set (#.# sec)



102

4.4 Using UNION

The **UNION** keyword enables concatenation of the results from two or more **SELECT** statements. The syntax for using it is as follows:

```
SELECT ...
...
UNION
SELECT ...
...
UNION
SELECT ...
...
...
```

The result of such a statement consists of the rows retrieved by the first **SELECT**, followed by the rows retrieved by the second **SELECT**, and so on. Each **SELECT** must produce the same number of columns.

By default, **UNION** eliminates duplicate rows from the result set. To retain all rows, replace each instance of **UNION** with **UNION ALL**. (**UNION ALL** is more efficient for the server to process because it need not perform duplicate removal. However, returning the result set to the client involves more network traffic.)

UNION is useful under the following circumstances:

- There is similar information in multiple tables and it is necessary to retrieve rows from all of them at once.
- It is necessary to select several sets of rows from the same table, but the conditions that characterize each set aren't easy to write as a single **WHERE** clause. **UNION** allows retrieval of each set with a simpler **WHERE** clause in its own **SELECT** statement; the rows retrieved by each are combined and produced as the final query result.

103

Suppose that there are three mailing lists, each of which is managed using a different MySQL-based software package. Each package uses its own table to store names and email addresses, but they have slightly different conventions about how the tables are set up. The tables used by the list manager packages look like this:

```
mysql> CREATE TABLE list1 (
->     subscriber  CHAR(60),
->     email        CHAR(60)
-> );
mysql> CREATE TABLE list2 (
->     name         CHAR(96),
->     address      CHAR(128)
-> );
mysql> CREATE TABLE list3 (
->     email        CHAR(50),
->     real_name    CHAR(30)
-> );
```



Note that each table contains similar types of information (names and email addresses), but they do not use the same column names or types, and they do not store the columns in the same order. To write a query that produces the combined subscriber list, use **UNION**. It doesn't matter that the tables do not have exactly the same structure. To select their combined contents, name the columns from each table in the order that it is necessary to see them.

A query to retrieve names and addresses from the tables looks like this:

```
mysql> SELECT subscriber, email
      -> FROM list1
      -> UNION
      -> SELECT name, address
      -> FROM list2
      -> UNION
      -> SELECT real_name, email
      -> FROM list3;
```

The first column of the result contains names and the second column contains email addresses. The names of the columns resulting from a **UNION** are taken from the names of the columns in the first **SELECT** statement. This means that the result set column names are subscriber and email. If an alias is provided for columns in the first **SELECT**, the aliases are used as the output column names.

The data types of the output columns are determined by considering the values retrieved by all of the **SELECT** statements. For the query shown, the data types will be **CHAR(96)** and **CHAR(128)** because those are the smallest types that are guaranteed to be large enough to hold values from all three tables.

104

ORDER BY and **LIMIT** clauses can be used to sort or limit a **UNION** result set as a whole. To do this, surround each **SELECT** with parentheses and then add **ORDER BY** or **LIMIT** after the last parenthesis. Columns named in such an **ORDER BY** should refer to columns in the first **SELECT** of the statement. (This is a consequence of the fact that the first **SELECT** determines the result set column names.) The following statement sorts the result of the **UNION** by email address and returns the first 10 rows of the combined result:

```
mysql> (SELECT subscriber, email
      ->   FROM list1)
      -> UNION
      -> (SELECT name, address
      ->   FROM list2)
      -> UNION
      -> (SELECT real_name, email
      ->   FROM list3)
      -> ORDER BY email
      -> LIMIT 10;
```



ORDER BY and **LIMIT** clauses also can be applied to individual **SELECT** statements within a **UNION**. Surround each **SELECT** with parentheses and add **ORDER BY** or **LIMIT** to the end of the appropriate **SELECT**. In this case, an **ORDER BY** should refer to columns of the particular **SELECT** with which it's associated. (Also, although **LIMIT** may be used by itself in this context, **ORDER BY** has no effect unless combined with **LIMIT**. The optimizer ignores it otherwise.) The following query sorts the result of each **SELECT** by email address and returns the first five rows from each one:

```
mysql> (SELECT subscriber, email
->   FROM list1
->   ORDER BY email
->   LIMIT 5)
-> UNION
-> (SELECT name, address
->   FROM list2
->   ORDER BY address
->   LIMIT 5)
-> UNION
-> (SELECT real_name, email
->   FROM list3
->   ORDER BY email
->   LIMIT 5);
```





InLine Lab 4-G

In this exercise you will use the **UNION** statement using standard mathematical expressions and the **world** database. This will require a mysql command line client and access to a MySQL server.

Step 1. Using UNION

Action: Use a **UNION** to concatenate the output of two separate mathematical operations; $1+1$ and $2+2$.

```
mysql> SELECT 1+1 UNION SELECT 2+2;
```

Effect: The resultant table shows the output for each of the separate **SELECT** calculations.

Step 2. More UNION

Action: Combine the names of all countries from Europe with all countries from North America.

```
mysql> SELECT Name FROM Country WHERE Continent = 'Europe'  
-> UNION  
-> SELECT Name FROM Country WHERE Continent = 'North America';
```

Effect: Returns a list of all 83 country names.

NOTE: This query could be re-written to one **SELECT** with a combined **WHERE** clause. This would be more efficient.





Further Practice

In this exercise, you will use the Querying for Table Data information covered in this chapter to select and combine column information from the **world** database tables.

105

1. List all the distinct regions in the world.
2. List the first three countries in alphabetic order.
3. List all countries in the 'Baltic Countries' region.
4. List all the countries where the life expectancy is more than 79.
5. List the 5 largest cities in the world.
6. List the different countries (country codes) that have cities with more than 7 000 000 inhabitants. How many are there?
7. Find out the 5 most common government forms in the world (most common = the largest number of countries).
8. List the average surface area of the countries in each continent.
9. Calculate the average life expectancy in each continent (remember that each country have a different population).
10. List the 5 most densely populated countries in the world (only include countries with a surface area larger than 10000).
11. How many different districts are represented in the City table?
12. List all the languages that are spoken in more than 10 countries.



106

4.5 Chapter Summary

This chapter introduced the use of simple MySQL **SELECT** statements to query for database data. In this chapter, you learned to:

- Execute table data queries using the **SELECT** statement
- Use the **WHERE** clause to obtain only specific rows
- Use the **ORDER BY** clause to sort data into a particular order
- Use the **LIMIT** clause to obtain only a portion of the result set
- Use Aggregating functions to obtain summary values
- Use **GROUP BY** to group rows of data
- Use **HAVING** to specify criteria according to aggregated values
- Use **WITH ROLLUP** to obtain super aggregates
- Use the **UNION** keyword to concatenate results from multiple **SELECT** statements



5 HANDLING ERRORS AND WARNINGS

108

5.1 Learning Objectives

This chapter explains how to handle errors and warnings in MySQL. In this chapter, you will learn to:

- Set SQL Modes to effect error output
- Handle missing or invalid data values
- Interpret error messages
- Use the **SHOW WARNINGS** and **SHOW ERRORS** statements
- Invoke the perror utility program



109

5.2 SQL modes affecting syntax

SQL modes control aspects of server operation such as what SQL syntax MySQL should support and what kind of data validations it should perform. This makes it easier to use MySQL in different environments and to use MySQL together with other database servers.

The SQL mode exists on the global level and on the session level. By default, a new session uses the global sql_mode, but during the course of the session the sql_mode can be changed. The sql_mode can always be changed at the session level, and privileged users can also change the sql mode on the global level.

You can retrieve the current global SQL mode using the following statement:

```
mysql> SELECT @@global.sql_mode;
```

To obtain the SQL mode for the current session, the following statements can be used:

```
mysql> SELECT @@session.sql_mode;
```

This is equivalent to this shorthand notation:

```
mysql> SELECT @@sql_mode;
```

Example:

```
mysql> SELECT @@sql_mode;
+-----+
| @@sql_mode |
+-----+
| STRICT_ALL_TABLES,ERROR_FOR_DIVISION_BY_ZERO |
+-----+
```

110

5.2.1 Setting the SQL Mode

The SQL mode is controlled by means of the `sql_mode` system variable. You can set the default SQL mode by starting `mysqld` with the `--sql-mode="mode_value"` option. You can also change the mode at runtime by setting the `sql_mode` system variable with a `SET` statement:

```
SET [SESSION|GLOBAL] sql_mode='mode_value'
```

Setting the `GLOBAL` variable affects the operation of all clients that connect from that time on. Setting the `SESSION` variable affects only the current client. Any client can change its own session `sql_mode` value at any time. When neither `GLOBAL` or `SESSION` is specified, the default will be `SESSION`.

The value should be an empty string, or one or more mode names separated by commas. If the value is empty or contains more than one mode name, it must be quoted. If the value contains a single mode name, quoting is optional. SQL mode values are not case sensitive, although this training guide always writes them in uppercase. Here are some examples:



Clear the SQL mode:

```
mysql> SET sql_mode = '';
```

Set the SQL mode using a single mode value:

```
mysql> SET sql_mode = ANSI_QUOTES;
```

Set the SQL mode using multiple mode names:

```
mysql> SET sql_mode = 'IGNORE_SPACE,ANSI_QUOTES';
```

Some SQL mode values are composite modes that actually enable a set of modes. Values in this category include ANSI and TRADITIONAL. To see which mode values a composite mode consists of, retrieve the value after setting it:

111

```
mysql> SET sql_mode='TRADITIONAL';
Query OK, 0 rows affected (#.## sec)
mysql> SELECT @@sql_mode\G
***** 1. row *****
@@sql_mode: STRICT_TRANS_TABLES,STRICT_ALL_TABLES,NO_ZERO_IN_DATE,
NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,TRADITIONAL,NO_AUTO_CREATE_USER
1 row in set (#.## sec)
```



The most important SQL mode values and their functionality are listed below:

112

- ANSI - Changes syntax and behavior to better conform to standard SQL. It is a composite mode that is shorthand for a set of modes:
 - ANSI_QUOTES – Changes the meaning of the double quote character. When this mode is enabled, double quotes may be used to quote identifiers (like backticks). By default, double quotes are interpreted as string delimiters (like single quotes)
 - IGNORE_SPACE – allow spaces between a function name and the left parenthesis that starts the parameter list
 - PIPES_AS_CONCAT – treat || as a string concatenation operator (like CONCAT())
 - REAL_AS_FLOAT – treat REAL as a synonym for FLOAT
- ONLY_FULL_GROUP_BY – do not allow queries when the **SELECT** list refers to columns that are not in **GROUP BY** clause (used to be part of ANSI)
- ERROR_FOR_DIVISION_BY_ZERO - Attempts to enter data via **INSERT** or **UPDATE** statements produce an error if an expression includes division by zero. (With **ERROR_FOR_DIVISION_BY_ZERO** but not strict mode, division by zero results in a value of **NULL** and a warning, not an error.)
- STRICT_TRANS_TABLES, STRICT_ALL_TABLES - These values enable “strict mode”, which sets general input value restrictions. In strict mode, the server rejects values that are out of range, that have an incorrect data type, or that are missing for columns that have no default. If a value could not be inserted as given into a transactional table, abort the statement. For a non-transactional table, abort the statement if the value occurs in a single-row statement, or the first row of a multiple-row statement.
- NO_ZERO_DATE, NO_ZERO_IN_DATE - Several SQL mode values control how MySQL handles invalid date input. By default, MySQL requires that the month and day values correspond to an actual legal date, except that it allows “zero” dates ('0000-00-00') and dates that have zero parts ('2009-12-00', '2009-00-01'). Zero dates and dates with zero parts still allowed even in strict mode. To prohibit such dates, enable strict mode and the **NO_ZERO_DATE** and **NO_ZERO_IN_DATE** mode values.
- TRADITIONAL - Make MySQL behave more like a “traditional” SQL database system. A simple description of this mode is “give an error instead of a warning” when inserting an incorrect value into a column. It is a composite mode that is shorthand for a set of modes:
 - STRICT_TRANS_TABLES
 - STRICT_ALL_TABLES
 - NO_ZERO_DATE
 - NO_ZERO_IN_DATE
 - ERROR_FOR_DIVISION_BY_ZERO
 - NO_AUTO_CREATE_USER – do not allow auto creation of new users

MySQL Reference Manual

The MySQL reference manual, available online at the MySQL website, provides a complete list of all available SQL mode values along with in-depth documentation for most of the areas of MySQL use.



113

5.3 Handling Missing or Invalid Data Values

Many database servers spent a great deal of effort in validating data inserted into tables, usually generating errors for invalid input due to a mismatch with the column data types. Historically, MySQL has been non-traditional and more “forgiving” in its data handling:

The typical behavior for a MySQL server is to convert erroneous input values to the “closest” allowable values (as determined from column definitions) and continues on its way. For example, if you attempt to store a negative value into an **UNSIGNED** column, MySQL converts it to zero, which is considered to be the nearest legal value for the column's data type.

114

5.3.1 Handling Missing Values

In MySQL, **INSERT** statements may be incomplete in the sense of not specifying a value for every column in a table. Consider the following table definition:

```
mysql> CREATE TABLE t (
    ->     i INT DEFAULT NULL,
    ->     j INT NOT NULL,
    ->     k INT DEFAULT -1
    -> );
```

For this table, an **INSERT** statement is incomplete unless it specifies values for all three columns in the table. Each of the following statements is an example of a statement that is missing column values:

```
mysql> INSERT INTO t (i) VALUES (0);
mysql> INSERT INTO t (i,k) VALUES (1,2);
mysql> INSERT INTO t (i,k) VALUES (1,2), (3,4);
mysql> INSERT INTO t VALUES ();
```

In the last statement, the empty **VALUES** list means “use the default value for all columns.”



MySQL handles missing values as follows:

115

- If the column definition contains a **DEFAULT** clause, MySQL inserts the value specified by that clause. Note that MySQL adds a **DEFAULT NULL** clause to the definition if it has no explicit **DEFAULT** clause and the column can take **NULL** values. Thus, the definition of column i actually has **DEFAULT NULL** in its definition:

```
mysql> SHOW CREATE TABLE t\G
***** 1. row *****
Table: t
Create Table: CREATE TABLE `t` (
  `i` int(11) default NULL,
  `j` int(11) NOT NULL,
  `k` int(11) default '-1'
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

- If a column definition has no **DEFAULT** clause, missing-value handling depends on whether strict SQL mode is in effect and whether the table is transactional:
 - If strict mode is not in effect, MySQL inserts the implicit default value for the column data type and generates a warning.
 - If strict mode is in effect, an error occurs for transactional tables (and the statement rolls back). An error occurs for non-transactional tables as well, but a partial update might result: If the error occurs for the second or later row of a multiple-row insert, the earlier rows will already have been inserted.

The definition of column j has no **DEFAULT** clause, so **INSERT** statements that provide no value for j are handled according to these rules.

5.3.2 Handling invalid values in non-strict mode

116

In general, when operating in non-strict mode, MySQL performs type conversion based on the constraints implied by a column's definition. These constraints apply in several contexts:

- When you insert or update column values with statements such as **INSERT**, **REPLACE**, **UPDATE**, or **LOAD DATA INFILE**.
- When you change a column definition with **ALTER TABLE**.
- When you specify a default value using a **DEFAULT** value clause in a column definition. (For example, if you specify a default value of '43' for a numeric column, that string is converted to 43 when the default value is used.)

If MySQL is not operating in strict mode, it adjusts invalid input values to legal values when possible and generates warning messages. These messages can be displayed with the **SHOW WARNINGS** statement.



The following list discusses some of the conversions that MySQL performs. It isn't exhaustive, but is sufficiently representative to provide you with a good idea of how MySQL treats input values.

- **Conversion of out-of-range values to in-range values:** If you attempt to store a value that's smaller than the minimum value allowed by the range of a column's data type, MySQL stores the minimum value in the range. If you attempt to store a value that's larger than the maximum value in the range, MySQL stores the range's maximum value. For example, `TINYINT` has a range of `-128` to `127`. If you attempt to store values less than `-128` in a `TINYINT` column, MySQL stores `-128` instead. Similarly, MySQL stores values greater than `127` as `127`. If you insert a negative value into an `UNSIGNED` numeric column, MySQL converts the value to `0`.
- **String truncation:** String values that are too long are truncated to fit in the column. If you attempt to store '`Sakila`' into a `CHAR(4)` column, MySQL stores it as '`Saki`' and discards the remaining characters. (It is not considered an error to trim trailing spaces, so MySQL will insert '`Saki` ' into the column as '`Saki`' without generating a warning.)
- **Enumeration and set value conversion:** If a value that's assigned to an `ENUM` column isn't listed in the `ENUM` definition, MySQL converts it to `''` (the empty string). If a value that's assigned to a `SET` column contains elements that aren't listed in the `SET` definition, MySQL discards those elements, retaining only the legal elements.
- **Conversion to data type default:** If you attempt to store a value that cannot be converted to the column data type, MySQL stores the implicit default value for the type. For example, if you try to store the value '`Sakila`' in an `INT` column, MySQL stores the value `0`. The "zero" value is `'0000-00-00'` for date columns and `'00:00:00'` for `TIME` columns. The default for these types is an empty value (`''`). The empty string value is not equivalent to the `NULL` value.
- **Handling assignment of `NULL` to `NOT NULL` columns:** The effect of assigning `NULL` to a `NOT NULL` column depends on whether the assignment occurs in a single-row or multiple-row `INSERT` statement. For a single-row `INSERT`, an error occurs and the statement fails. For a multiple-row `INSERT`, MySQL assigns the column the implicit default value for its data type.

117

Using `ALTER TABLE` to change a column's data type maps existing values to new values according to the constraints imposed by the new data type. This might result in some values being changed. For example, if you change a `TINYINT` to an `INT`, no values are changed because all `TINYINT` values fit within the `INT` range. However, if you change an `INT` to a `TINYINT`, any values that lie outside the range of `TINYINT` are clipped to the nearest endpoint of the `TINYINT` range. Similar effects occur for other types of conversions, such as `TINYINT` to `TINYINT UNSIGNED` (negative values are converted to zero), and converting a long string column to a shorter one (values that are too long are truncated to fit the new size).

If a column is changed to `NOT NULL` using `ALTER TABLE`, MySQL converts `NULL` values to the implicit default value for the data type.



The following table shows how several types of string values are handled when converted to `DATE` or `INT` data types. It demonstrates several of the points just discussed. Note that only string values that look like dates or numbers convert properly without loss of information.

string value	converted date value	converted integer value
'2010-03-12'	'2010-03-12'	2010
'03-12-2010'	'0000-00-00'	3
'0017'	'0000-00-00'	17
'500 hats'	'0000-00-00'	500
'bartholomew'	'0000-00-00'	0

118

5.3.3 Handling Invalid Values in Strict Mode

Input values may be invalid for a number of reasons:

- For a numeric or temporal column, a value might be out of range.
- For a string column, a string might be too long.
- For an `ENUM` column, a value might be specified that is not a legal enumeration value or as part of a value. For a `SET` column, a value might contain an element that is not a set member.
- For a column that is defined as `NOT NULL`, a value of `NULL` might have been given.

Enabling strict mode turns on general input value restrictions. In strict mode, the server rejects values that are out of range, have an incorrect data type, or are missing for columns that have no default. Strict mode is enabled using the `STRICT_TRANS_TABLES` and `STRICT_ALL_TABLES` mode values.

`STRICT_TRANS_TABLES` enables strict behavior for errors that can be rolled back or canceled without changing the table into which data is being entered. If an error occurs for a transactional table, the statement aborts and rolls back. For a non-transactional table, the statement can be aborted without changing the table if an invalid value occurs in a single-row insert or the first row of a multiple-row insert. Otherwise, to avoid a partial update for a non-transactional table, MySQL adjusts any invalid value to a legal value, inserts it, and generates a warning. (Adjustment of `NULL` inserted into a `NOT NULL` column is done by inserting the implicit default value for the column data type.)

`STRICT_ALL_TABLES` is similar to `STRICT_TRANS_TABLES` but causes statements for non-transactional tables to abort even for errors in the second or later rows of a multiple-row insert. This means that a partial update might occur, because rows earlier in the statement will already have been inserted.



5.3.4 Additional Input Data Restrictions

119

Strict mode turns on general input value restrictions, but it is not as strict as you can tell the MySQL server to be. When strict mode is in effect, certain SQL mode values enable additional restrictions on input data values:

Division by zero can be treated as an error for data entry by enabling the `ERROR_FOR_DIVISION_BY_ZERO` mode value and strict mode. In this case, attempts to enter data via `INSERT` or `UPDATE` statements produce an error if an expression includes division by zero. (With `ERROR_FOR_DIVISION_BY_ZERO` but not strict mode, division by zero results in a value of `NULL` and a warning, not an error.)

```
mysql> SET sql_mode = 'STRICT_ALL_TABLES,ERROR_FOR_DIVISION_BY_ZERO';
```

By default, MySQL allows “zero” dates ('0000-00-00') and dates that have zero parts ('2009-12-00', '2009-00-01'). Such dates are allowed even if you enable strict mode, but if you want to prohibit them, you can enable the `NO_ZERO_DATE` and `NO_ZERO_IN_DATE` mode values:

```
mysql> SET sql_mode = 'STRICT_ALL_TABLES,NO_ZERO_DATE,NO_ZERO_IN_DATE';
```

The `TRADITIONAL` mode value is a composite mode that enables strict mode as well as the other restrictions just described. If you want your MySQL server to be as restrictive as possible about input data checking (and thus to act like other “traditional” database servers), the simplest way to achieve this is to enable `TRADITIONAL` mode rather than a list of individual more-specific modes:

```
mysql> SET sql_mode = 'TRADITIONAL';
```

Setting the SQL mode by using `TRADITIONAL` has the additional advantage that if future versions of MySQL implement other input data restrictions that become part of `TRADITIONAL` mode, you won't have to explicitly enable those modes to take advantage of them.



5.3.5 Interpreting Error Messages

120

If problems occur when you attempt to connect to MySQL Server with a client program or while the server attempts to execute the SQL statements that you send to it, MySQL produces diagnostic messages. Clients can display this information to assist you in troubleshooting and resolving problems.

Diagnostics might be error messages to indicate serious problems or warning messages to indicate less severe problems. MySQL provides diagnostic information in the following ways:

- An error message is returned for statements that fail:

```
mysql> SELECT * FROM no_such_table;
ERROR 1146 (42S02): Table 'test.no_such_table' doesn't exist
```

These messages typically have three components:

- A MySQL-specific error code.
 - An SQLSTATE error code. These codes are defined by standard SQL and ODBC.
 - A text message that describes the problem.
- An information string is returned by statements that affect multiple rows. This string provides a summary of the statement outcome:

```
mysql> INSERT INTO integers VALUES ('abc'), (-5), (NULL);
Query OK, 3 rows affected, 3 warnings (#.## sec)
Records: 3  Duplicates: 0  Warnings: 3
```

121

- An operating system-level error might occur:

```
mysql> CREATE TABLE CountryCopy SELECT * FROM Country;
ERROR 1 (HY000): Can't create/write to file './world/CountryCopy.frm'
(Errcode: 13)
```

For cases such as the preceding `SELECT` from a non-existent table, where all three error values are displayed, you can simply look at the information provided to see what the problem was. In other cases, all information might not be displayed. The information string for multiple-row statements is a summary, not a complete listing of diagnostics. An operating system error includes an Errcode number that might have a system-specific meaning.

- Error code number

The MySQL reference manual, available online at the MySQL website, provides a complete list of all descriptions of Error Codes and Messages along with in-depth documentation for most of the areas of MySQL use.



5.3.6 The SHOW WARNINGS Statement

122

MySQL Server generates warnings when it is not able to fully comply with a request or when an action has possibly unintended side effects. These warnings can be displayed with the **SHOW WARNINGS** statement.

The following example shows how warnings are generated for attempts to insert a character string, a negative integer, and **NULL** into a column that is defined as **INT UNSIGNED NOT NULL**:

```
mysql> CREATE TABLE integers (i INT UNSIGNED NOT NULL);
Query OK, 0 rows affected (#.# sec)
mysql> INSERT INTO integers VALUES ('abc'), (-5), (NULL);
Query OK, 3 rows affected, 3 warnings (#.# sec)
Records: 3  Duplicates: 0  Warnings: 3
```

The information returned by the server indicates that there were three instances in which it had to truncate or otherwise change the input to accept the data values that were passed in the **INSERT** statement. When a statement cannot be executed without some sort of problem occurring, the **SHOW WARNINGS** statement provides information to help you understand what went wrong. The following example shows the warnings generated by the preceding **INSERT** statement:

123

```
mysql> SHOW WARNINGS\G
*****
1. row ****
Level: Warning
Code: 1264
Message: Out of range value adjusted for column 'i' at row 1
*****
2. row ****
Level: Warning
Code: 1264
Message: Out of range value adjusted for column 'i' at row 2
*****
3. row ****
Level: Warning
Code: 1263
Message: Column set to default value; NULL supplied to NOT NULL
        column 'i' at row 3
3 rows in set (#.# sec)
```



124

You can combine **SHOW WARNINGS** with **LIMIT**, just as you're used to doing with **SELECT** statements, to "scroll" through the warnings a section at a time:

```
mysql> SHOW WARNINGS LIMIT 1,2\G
***** 1. row ****
Level: Warning
Code: 1264
Message: Out of range value adjusted for column 'i' at row 2
***** 2. row ****
Level: Warning
Code: 1263
Message: Column set to default value; NULL supplied to NOT NULL
        column 'i' at row 3
2 rows in set (#.## sec)
```

If you want to know only how many warnings there were, use **SHOW COUNT(*) WARNINGS**.

```
mysql> SHOW COUNT(*) WARNINGS;
+-----+
| @@session.warning_count |
+-----+
|                      3 |
+-----+
```

Warning Levels

125

Warnings generated by one statement are available from the server only for a limited time (until you issue another statement that can generate warnings). If you need to see warnings, you should always fetch them as soon as you detect that they were generated.

"Warnings" actually can occur at several levels of severity:

- **Error** messages indicate serious problems that prevent the server from completing a request.
- **Warning** messages indicate problems for which the server can continue processing the request.
- **Note** messages are informational only.



The following example shows messages that are generated at different levels. An error occurs for the SELECT statement, which cannot be executed successfully because the table does not exist. For the DROP statement, the message is only a note. The purpose of the statement is to ensure that the table does not exist. That is certainly true when the statement finishes, even though the statement did nothing.

```
mysql> SELECT * FROM no_such_table;
ERROR 1146 (42S02): Table 'test.no_such_table' doesn't exist
mysql> SHOW WARNINGS;
+-----+-----+
| Level | Code | Message           |
+-----+-----+
| Error | 1146 | Table 'test.no_such_table' doesn't exist |
+-----+-----+
1 row in set (#.## sec)

mysql> DROP TABLE IF EXISTS no_such_table;
Query OK, 0 rows affected, 1 warning (#.## sec)
mysql> SHOW WARNINGS;
+-----+-----+
| Level | Code | Message           |
+-----+-----+
| Note  | 1051 | Unknown table 'no_such_table' |
+-----+-----+
1 row in set (#.## sec)
```

126

To suppress generation of Note warnings, you can set the `sql_notes` system variable to zero:

```
mysql> SET sql_notes = 0;
```

127

5.3.7 The `SHOW ERRORS` Statement

The `SHOW ERRORS` statement is similar to `SHOW WARNINGS`, but displays only messages for error conditions. As such, it shows only messages having a higher severity and tends to produce less output than `SHOW WARNINGS`. `SHOW ERRORS`, like `SHOW WARNINGS`, supports a `LIMIT` clause to restrict the number of rows to return. It also can be used as `SHOW COUNT(*) ERRORS` to obtain a count of the error messages.



5.3.8 The perror Utility

128

perror is a command-line utility that is included with MySQL distributions. The purpose of the perror program is to show you information about the error codes used by MySQL when operating system-level errors occur. You can use perror in situations when a statement results in a message such as the following being returned to you:

```
mysql> CREATE TABLE CountryCopy SELECT * FROM Country;
ERROR 1 (HY000): Can't create/write to file './world/CountryCopy.frm'
(Errcode: 13)
```

This error message indicates that MySQL cannot write to the file CountryCopy.frm, but does not report the reason. It might be due to a full disk, a file system permissions problem, or some other error. To find out, run the perror program with an argument of the number given following Errcode in the preceding error message. perror displays a message indicating that the source of the problem is that someone has incorrectly set the file system permissions for the current database:

```
shell> perror 13
Error code 13: Permission denied
```

Instructor Note: It is not in the guide or slides, but show the students what the DROP command and SHOW WARNINGS look like after the sql_notes is set to zero;

```
mysql> DROP TABLE IF EXISTS no_such_table;
Query OK, 0 rows affected (#.# sec)
mysql> SHOW WARNINGS;
Empty set (#.# sec)
```





Quiz 5-A

In this exercise you will answer questions pertaining to SQL Modes.

1. Which of the following statements are true?
 - a) The SQL mode is set for the server, so it affects all clients that connect to the server.
 - b) If you want to set two SQL modes (for example, the `STRICT_ALL_TABLES` and `ERROR_FOR_DIVISION_BY_ZERO` modes), you must issue two `SET` statements.
 - c) Unless explicitly declared as global, setting SQL modes affects only the client that sets the modes.
 - d) SQL modes affect the behavior of the server; for example, they influence the way in which the server handles invalid input data.
 - e) SQL modes affect the features that the server provides for a client; for example, you could turn InnoDB support on and off using SQL modes.
- 2) What is the syntax for changing the SQL mode to '`STRICT_TRANS_TABLES`' and '`PIPES_AS_CONCAT`'?

- 3) If you attempt to store a negative value into an `UNSIGNED` column, MySQL converts it to the 'nearest' legal value for the column. Which value is that?

- 4) What type of values will be used for the following column `INSERT` statement?

```
INSERT INTO table1 VALUES();
```

-
5. With a particular setting for the SQL mode, MySQL adjusts invalid input values to legal values when possible and generates warning messages. The SQL mode can also be adjusted in order to issue an errors rather than warnings. What is the general term for SQL modes that enable this less forgiving behavior?

 6. MySQL Server generates warnings or errors when it is not able to fully comply with a request or when an action has possibly unintended side effects. These warnings and errors can be displayed with the following two statements:

and:



7. List the three levels of severity for warnings:

a) _____

b) _____

c) _____





Further Practice 5-B

In this exercise, you will use the information covered in this chapter to identify various errors and warnings while using the `mysql` client.

129

1. Find the pages in the reference manual that list the error codes (use the most current version 5.1 reference manual).
2. Using `perror`, find what the error code 130 represents.
3. Using the `test` database, attempt to list all the row content in a table called `test_table`. What is the type of message that you receive?
4. Find the exact description of this error code on the on-line reference manual page for this type of error code.
5. Show the current SQL mode setting. Change the SQL mode to `ANSI`. Confirm that the mode has been changed.
6. Attempt to remove the table named `test_table` (Use `DROP TABLE IF EXISTS`). What is the type of message that you receive?
7. Use the appropriate `SHOW` statement to view a description of the message.
8. View any current errors. Explain the result.



5.4 Chapter Summary

130

This chapter introduced handling of errors and warnings in MySQL. In this chapter, you learned to:

- Set SQL Modes to effect error output
- Handle missing or invalid data values
- Interpret error messages
- Use the **SHOW WARNINGS** and **SHOW ERRORS** statements
- **INVOKE THE perror UTILITY PROGRAM**





6 DATA TYPES

6.1 Learning Objectives

132 This chapter introduces the data types used with MySQL. In this chapter, you will learn to:

- Describe the three major categories of data types
- Understand character sets and collation
- Assign the appropriate data type to table entities
- Understand the meaning and use of **NULL** and **NOT NULL**



133

6.2 Data Type Overview

In MySQL the data types available can be broken down into four major categories:

- Numeric Numeric values (Integers, Floating-Point, Fixed-Point and Bit-field)
- Character Text strings
- Binary Binary data strings
- Temporal Time and dates

Within each category there are numerous specific data types that use varying amounts of memory and disk space, thus have varying effects on performance. Choosing the best data type for the column has a rather small effect on performance in an individual record, but as the database grows these small effects can lead to larger effects. This should be taken into account early in the design process, before they become performance issues.

6.2.1 The ABC's of Data Types

- A) Apt - the data needs to be represented in the type most fitting the thing it represents
- B) Brief - if you choose the data type that uses the least amount of storage space. This will save resources and increase performance
- C) Complete - the data type needs to allow for enough room to store the largest possible value for the particular item.

134

6.2.2 Creating Tables with Data Types

When you create a table, the declaration for each of its columns includes the column name, a data type that indicates what kind of values the column may hold, and possibly some attributes (options) that more specifically define how MySQL should handle the column. For example, the following statement creates a table named people, which contains an integer-valued numeric column named id and two 30-character string columns named `first_name` and `last_name`:

```
mysql> CREATE TABLE people (
    ->     id      INT,
    ->     first_name CHAR(30),
    ->     last_name  CHAR(30)
    -> );
```

The column definitions in that `CREATE TABLE` statement contain only names and data types. To more specifically control how MySQL handles a column, add attributes to the column definition. For example, to disallow negative values in the `id` column, add the `UNSIGNED` attribute.



135

6.3 Numeric Data Types

For storing numeric data, MySQL provides

- Integer data types for storing whole numbers
- Floating-point types that store approximate-value (real) numbers
- a Fixed-point type that stores exact-value (real) numbers
- a BIT type for bit-field values.

When you choose a numeric data type, consider the following factors:

- The range of values the data type represents
- The amount of storage space that column values require
- The column precision and scale for floating-point and fixed-point values

Precision and scale are terms that apply to floating-point and fixed-point values, which can have both an integer part and a fractional part. Precision is the number of significant digits. Scale is the number of digits to the right of the decimal point.

136

6.3.1 Integer Types

Integers are whole numbers. Integers do not have a fractional part, that is, a single integer value does not have any decimal places. The following list are the integer data types supported by MySQL. The types are listed in order of ascending precision (that is, each next integer data type can hold a larger range of integer values than the next one in the list)

- **TINYINT** – a very small integer data type
- **SMALLINT** – a small integer data type
- **MEDIUMINT** – a medium sized integer data type
- **INT** - a normal (average) size integer data type. **INTEGER** is a synonym for **INT**.
- **BIGINT** - a large integer data type

Integer data types may specify a *displaywidth*, which may be used by clients in rendering data from columns of the particular data type.:

```
INT [ (<width>) ]
```

Note that the displaywidth does not have any bearing on the range of values that may be stored in the integer column.

Integer example: the **Population** column from the **City** table in the **world** database:

```
Population INT(11)
```



All integer types can have an optional (non-standard) attribute **UNSIGNED**. By default, integer types are signed. A column that uses a signed integer type can be used to store both negative whole numbers as well as non-negative numbers. When you want to allow only non-negative numbers in a column, you can mark the data type as **UNSIGNED**, which will prevent any negative values from being stored in that column.

Marking an integer as **UNSIGNED** automatically makes 0 the smallest number that can be stored in the column, and effectively doubles the maximum number that may be stored in the column. That is because signed integers use one bit to represent the sign. When marking the integer as **UNSIGNED**, that bit is interpreted as part of the numerical value rather than to convey whether the number is positive or negative. So if an **INT** column is **UNSIGNED**, the size of the column's range is the same but its endpoints shift from -2147483648 and 2147483647 up to 0 and 4294967295.

When used in conjunction with the optional extension attribute **ZEROFILL**, the default padding of spaces is replaced with zeros. For example, for a column declared as **INT(5) ZEROFILL**, a value of 4 is retrieved as 00004. If you specify **ZEROFILL** for a numeric column, MySQL automatically adds the **UNSIGNED** attribute to the column.

The integer data types are summarized in the following table, which indicates the amount of storage per value that each type requires as well as its range.

137

Data Type	#bytes	Minimum Value	Maximum Value
TINYINT	1	-128	128
TINYINT UNSIGNED	1	0	255
SMALLINT	2	-32768	32767
SMALLINT UNSIGNED	2	0	65535
MEDIUMINT	3	-8388608	8388607
MEDIUMINT UNSIGNED	3	0	16777215
INT	4	-2147483648	2147483647
INT UNSIGNED	4	0	4294967295
BIGINT	8	-9223372036854775808	9223372036854775807
BIGINT UNSIGNED	8	0	18446744073709551615

138

6.3.2 Floating-Point Types

The floating-point data types include **FLOAT** and **DOUBLE**. Each of these types may be used to represent approximate-value numbers that have an integer part, a fractional part, or both. **FLOAT** and **DOUBLE** data types represent values in the native binary floating-point format (IEEE 754) used by the server host's CPU. This is a very efficient type for storage and computation, but values are subject to rounding error.

- **FLOAT**: a small, single-precision (4-byte) floating point number. A single precision floating point number is accurate to approximately 7 digits.
- **DOUBLE**: This is a normal, double-precision (8-byte) floating point number. A double precision floating point number is accurate to approximately 15 digits.



For both **FLOAT** and **DOUBLE**, The default value is **NULL** if column is nullable and 0 (numerical zero) if the column is not nullable.

The **FLOAT** data type may be specified using a number of different notations:

- Without any specification of the precision:

FLOAT

This specifies a data type that can hold single precision floating-point numbers. Each value stored occupies 4 bytes. This notation is defined by the SQL standard

- With a specification of the binary precision:

FLOAT (M)

In this case, *M* denotes the precision as the number of available bits to store the floating point value. This notation is also defined by the SQL standard. However, in the MySQL implementation, the value of *M* is not taken literally; rather it is used to decide whether 4 or 8 bytes will be used. If $0 \leq M \leq 23$, MySQL will use 4 bytes and if $24 \leq M \leq 53$, then MySQL will use 8 bytes.

- A specification of the number of digits for the number and its fractional part.

FLOAT (D, F)

In this notation, *D* denotes the maximum number of decimal digits and *F* is the number of digits following the decimal point. The maximum for *D* is 255 and the maximum for *F* is 30. Additionally, *F* must not be larger than *D*. One should realize that *D* does not indicate a precision in this case: a 4-byte **FLOAT** can accurately represent at most 6 digits, and beyond that, rounding can and will occur to fit the values into 4 bytes.

For **DOUBLE**, the notation using the binary precision is not available, leaving two available notations:

- Without any specification of the precision:

DOUBLE

This specifies a data type that can hold double precision floating-point numbers. Each value stored occupies 8 bytes. This notation is defined by the SQL standard

- A specification of the number of digits for the number and its fractional part.

DOUBLE (D, F)

In this notation, *D* denotes the maximum number of decimal digits and *F* is the number of digits following the decimal point. This MySQL specific notation is not defined by the SQL standard. Like for **FLOAT**, the maximum for *D* is 255 and the maximum for *F* 30. One should realize that *D* does not indicate a precision in this case: an 8-byte **DOUBLE** can accurately represent at most 15 digits, and beyond that, rounding can and will occur to fit the values into 8 bytes.



Floating-Point Example:

The GNP column of the **Country** table in the **world** database:

```
GNP  FLOAT (10, 2)
```

Largest value uses only 7 digits, although 10 are allowed; 2 to right of decimal point

8510700.00

139

Floating-Point Type Comparison

Data Type	#bytes	Range
FLOAT	4	-3.402823466E+38 to -1.175494351E-38
FLOAT UNSIGNED	4	1.175494351E-38 to 3.402823466E+38
DOUBLE	8	-1.7976931348623157E+308 to -2.2250738585072014E-308
DOUBLE UNSIGNED	8	2.2250738585072014E-308 to 1.7976931348623157E+308

6.3.3 Fixed-Point Types

140

The fixed-point data type is **DECIMAL**. It is used to represent exact-value numbers that have an integer part, a fractional part, or both.

DECIMAL uses a fixed-decimal storage format: All values in a **DECIMAL** column have the same number of decimal places and are stored exactly as given. **DECIMAL** values are not processed quite as efficiently as **FLOAT** or **DOUBLE** values (which normally use the processor's native binary format), but **DECIMAL** values are not subject to rounding errors, so they are more accurate. In other words, there is an accuracy versus speed trade-off in choosing which type to use. For example, the **DECIMAL** data type is a popular choice for financial applications involving currency calculations, because accuracy is most important.

The **DECIMAL** data type is assigned using the following syntax:

```
DECIMAL (P, S)
```

P is the maximum number of significant digits (the precision) and S is the number of digits following the decimal point (the scale). P can be 65 at the maximum. S can be 30 at the maximum. Additionally, S cannot be larger than P.

Fixed –Point Example

If you want to represent values such as dollar-and-cents currency figures, you can use a two-digit scale:

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

Example value

650.88



The precision and scale can be omitted, or just the scale. The defaults for omitted precision and scale are 10 and 0, respectively, so the following declarations are equivalent:

```
total DECIMAL  
total DECIMAL(10)  
total DECIMAL(10,0)
```

The amount of storage required for **DECIMAL** column values depends on the precision and scale. Approximately 4 bytes are required per 9 digits on each side of the decimal point. The maximum range is the same as for **DOUBLE**; the effective range for a given **DECIMAL** column is determined by precision and scale.

The **NUMERIC** data type in MySQL is a synonym for **DECIMAL**. (If you declare a column as **NUMERIC**, MySQL uses **DECIMAL** in the definition.) In the SQL Standard, **DECIMAL** and **NUMERIC** are different types, but in MySQL they are the same. In standard SQL, the precision for **NUMERIC** must be exactly the number of digits given in the column definition. The precision for **DECIMAL** must be at least that many digits but is allowed to be more. In MySQL, the precision is exactly as given, for both types.

141

6.3.4 **BIT** Types

The **BIT** data type represents bit-field values. **BIT** column specifications take a width indicating the number of bits per value, from 1 to 64 bits. The following columns store 4 and 20 bits per value, respectively:

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

For a **BIT(n)** column, the range of values is 0 to $2^n - 1$, and the storage requirement is approximately $(n+7) \text{ div } 8$ bytes per value. So, **BIT(n)** requires 1 byte if $1 \leq n \leq 8$, 2 bytes if $9 \leq n \leq 16$ etc.

BIT columns can be assigned values using numeric expressions. To write literal **BIT** values in binary format, the literal-value notation `b'val'` can be used, where 'val' indicates a so-called *bitstring*: a string consisting of the characters '0' and '1' which represent individual bit values. For example, `b'1111'` equals 15 and `b'1000000'` equals 64.





InLine Lab 6-A

This lab requires you to use the MySQL command line client in order to view and set various numeric data types, as specified.

Step 1. Choose the current database and set the SQL_MODE

1. From the mysql prompt type;

```
mysql> USE test;
mysql> SET SQL_MODE := 'TRADITIONAL';
```

Sets the current database to be use to the **test** database.

Step 2. Create and test an integer data type

1. Type the following to create a table with a single column that will hold a small integer data type:

```
mysql> CREATE TABLE integers(
    ->     n SMALLINT UNSIGNED
    -> );
```

Creates a new table called **integers** in the **test** database with the column **n** set as a **SMALLINT UNSIGNED** data type.

2. Type the following to add values to the **integers** table just created:

```
mysql> INSERT INTO integers VALUES (5);
```

Inserts a value to the column **n** in the **integers** table of 5.

3. Type the following to verify that the value from the last step was added to the **integers** table.

```
mysql> SELECT * FROM integers;
```

Shows a tabular output of the current values in the **integers** table. Confirms the value that was added in the previous step

n
5



4. Type the following to add a negative value to the **integers** table:

```
mysql> INSERT INTO integers VALUES (-5);
```

Due the fact that the **n** column was designated as unsigned, the -5 value causes an error to be reported, which states that the value is out of range:

```
ERROR 1264 (22003): Out of range value for column 'n' at row 1
```

5. Type the following to add another positive value to the **integers** table:

```
mysql> INSERT INTO integers VALUES (70000);
```

Due the fact that the upper limit of the **SMALLINT** range is below the 70,000 value inserted, the insert causes an error to be reported, which states that the value is out of range.

```
ERROR 1264 (22003): Out of range value for column 'n' at row 1
```

Step 3. Create and test a float data types

1. Type the following to create a table with a single column that will hold a float data type:

```
mysql> CREATE TABLE floating (n FLOAT);
```

Creates a new table called **floating** in the **test** database with the column **n** defined to use the **FLOAT** data type.

2. Type the following to add a decimal value to the **floating** table:

```
mysql> INSERT INTO floating VALUES (.99);
```

Inserts a value to the column **n** in the floating table of .99.

3. Type the following to verify that the value was entered correctly:

```
mysql> SELECT * FROM floating;
```

Shows a tabular output of the current values in the **floating** table. Confirms the value that was added in the previous step:

+-----+
n
+-----+
0.99
+-----+



4. Type the following to compare the value stored:

```
mysql> SELECT * FROM floating WHERE n=.99;
```

The result set is empty, showing that the value is not exactly equal to .99.

5. Type the following to test the range of a float data type::

```
mysql> SELECT * FROM floating WHERE
-> n BETWEEN .99-.00001 AND .99+.00001;
```

The result set is now showing the value of .99, which fits within the range given in the query:

n
0.99

Step 4. Create and test a decimal data type

1. Type the following to create a table that uses a decimal data type that allows for three digits with one of those digits in the decimal place;

```
mysql> CREATE TABLE fixed (n DECIMAL (3,1));
```

Creates a new table called **fixed** in the test database with the column **n** set as a **DECIMAL** data type.

2. Type the following to add a decimal value to the **fixed** table;

```
mysql> INSERT INTO fixed VALUES (42.1);
```

Inserts a value to the column **n** in the **fixed** table of 42.1.

3. Type the following to verify that the value inserted is correctly formatted;

```
mysql> SELECT * FROM fixed;
```

Shows a tabular output of the current values in the **fixed** table. Confirms the value that was added in the previous step:

n
42.1



4. Type the following to determine if a value that exceeds the columns allowed digits will execute;

```
mysql> INSERT INTO fixed VALUES (142.1);
```

Due the fact that the limit of digits allowed by the **DECIMAL** data type was exceeded, the insert causes an error to be reported, which states that the value is out of range.

```
ERROR 1264 (22003) : Out of range value for column 'n' at row 1
```

Step 5. Create and test a BIT data type

1. Type the following to create a table that contains a single column storing bit values;

```
mysql> CREATE TABLE bits (b BIT(10));
```

Creates a new table called **bits** in the test database with the column **b** set as a BIT data type

2. Type the following to insert a bit value of 5:

```
mysql> INSERT INTO bits VALUES (b'101');
```

Inserts the value '101' into the column **b** of the **bits** table.

3. Type the following to view the bit value as it appears in the table:

```
mysql> SELECT b FROM bits;
```

Shows a tabular output of the current value in the **bits** table. Shows the value of the character that **b** represents;

b
♣

4. Type the following to retrieve the actual value of the bit stored;

```
mysql> SELECT b+0 FROM bits;
```

Shows a tabular output of the current value in the **bits** table. Shows the value that **b+0** represents. Adding 0 to the number converts it from base 2 to base 10;

b+0
5



142

6.4 Character String Data Types

A character string data type is a sequence of alphanumeric characters that belong to a given character set. Character strings are such an important and useful data type that they are implemented in nearly every programming language.

MySQL supports a number of data types for storing character strings. These types differ in a number of things, such as whether data is stored in a fixed or variable length format, the maximum length that can be stored and whether the type supports unstructured string values.

The following table lists the true character string data types provided in MySQL. All these types can be used for storing unstructured freely formatted character strings (provided their length fits the defined amount of space):

- **CHAR** Fixed-length character string
- **VARCHAR** Variable-length character string
- **TEXT** Variable-length character string

Apart from these unstructured string-types, MySQL also supports two structured string types. These are called structured, because the values that can be stored in columns of these types must be constructed out of a list of values supplied by the user defining the data type:

- **ENUM** Enumeration consisting of a fixed set of legal values
- **SET** Set consisting of a fixed set of legal values

When you choose a character string data type, consider the following factors:

- Is character set and collation important?
- The maximum length of values you need to store.
- Whether to use a fixed or variable amount of storage.
- How trailing spaces are handled for comparison, storage, and retrieval.
- The number of distinct values required; **ENUM** or **SET** may be useful if the set of values is fixed.
- Distribution of lengths (minimum versus maximum)

143

6.4.1 Unstructured Character String Data Types

The CHAR type

The **CHAR** data type is a fixed-length string. When defining a column of the **CHAR** data type, a length is specified. All in-memory representations of values for that column are always stored in a buffer that has the size specified by the length, even if the value could be stored in a smaller amount of memory. For this reason, the **CHAR** type is referred to as a fixed type.

(Internally values of the **CHAR** type are right padded with spaces to fit the specified length. Upon retrieval the spaces are removed again.)



The syntax for assigning a column to the **CHAR** data type is as follows:

```
CHAR (L)
```

The *L* stands for the maximum number of characters the **CHAR** value will accept. Internally each value for the **CHAR** column will occupy *L* characters even if the value proper does not require the entire available length. That means if a column **CHAR** type is defined to have a length of 10, but an actual column value is only 3 characters long, then in the internal representation, 7 characters remain unused.

For the **CHAR** data type, the maximum value for *L* is 255 characters. The required size is adjusted to fit the largest possible string with respect to the character set for that **CHAR** type. So, a **CHAR(255)** `utf8` column will occupy $3 * 255 = 765$ bytes, because MySQL `utf8` characters can be composed of at most 3 bytes.

The **VARCHAR** type

This data type is used for storage of variable-length character strings. The syntax for assigning a column to the **VARCHAR** data type is ...

```
VARCHAR (L)
```

The *L* stands for the maximum number of characters for values stored in this **VARCHAR** column. The theoretical maximum for *L* is 65533. However, the practical maximum is subject to a number of limitations:

- Internally, the size of a row cannot exceed 65565 bytes (by row size is meant: the maximum amount of bytes required for each column, except those of the **TEXT** and **BLOB** types). This limitation is actually not specific to **VARCHAR**, but is easily reached by creating a single **VARCHAR(65563)** column. A **VARCHAR(65563)** requires exactly 65565 bytes as 2 bytes are required to store the length of the character string value.
- *L* specifies a number of *characters*. However, depending on the character set, one character maybe represented by a one or more *bytes*. When defining a **VARCHAR** column, MySQL assumes worst case and computes that *L* times the maximum number of bytes per character will at most be needed for the value. So, for a `utf8` column, the maximum practical value for *L* will be:

65533 bytes / 3 ~ 21844 characters

We have to divide by 3 because MySQL `utf8` values can at most occupy 3 bytes.

When storing only single-byte `utf8` characters in **VARCHAR** `utf8` columns, MySQL will still respect the maximum length defined for the data type. So a **VARCHAR(21844)** `utf8` column can only store strings that are at maximum 21844 characters long. Even if all of those 21844 characters happen to be single-byte characters, then MySQL will still respect the length specified when the column was defined, although such a column theoretically has enough room to store $3 * 21844 = 65532$ single byte `utf8` characters or 32766 2-byte `utf8` characters.

- The maximum length of a value is subject to the maximum allowed packet size. This is configurable through the `max_allowed_packet` server variable



Note that the length for **CHAR** columns results in fixed length in-memory representation.

The specified length may or may not be used when storing the values on disk. For example, compressed MyISAM tables or ARCHIVE tables will use less disk space if they can.

In the internal representation, MySQL will require all space required to store the characters plus an additional 1 or 2 bytes to store the length of the string. This means generally speaking, **VARCHAR** uses less space (and memory) than **CHAR**.

Example:

The **Language** column from the **CountryLanguage** table in the **world** database:

Language **CHAR** (30)

An example value (from the **CountryLanguage** table) uses only 25 characters, although 30 are allowed:

'Southern Slavic Languages'

144

The text types

The text types comprises a family of unstructured, variable-length character string types that are most suitable for storing infrequently accessed, largish character strings

- **TINYTEXT** Stores character strings with a maximum length of 255 characters.
- **TEXT** Stores character strings with a maximum size of 65,535 bytes.
- **MEDIUMTEXT** Stores character strings with a maximum size of 16,777,215 bytes.
- **LONGTEXT** Stores character strings column with a maximum size of 4,294,967,295 bytes.

All data types in the text family except **TINYTEXT** define a maximum *size* as a number of *bytes*. **TINYTEXT** specifies a maximum *length* as a number of *characters*. When using single-byte character sets, such as latin1 or ASCII the size and the length will amount to the same figure.

When using multi-byte character sets like utf8 or ucs2, the length will generally be less than the size. We witnessed a similar issue when we discussed the **VARCHAR** data type. However, an important difference with the **VARCHAR** type is that MySQL does not exactly take the length of the character string value into account when determining whether a value can be stored in a text column. Rather, it will look at the size of the character string value, meaning that a utf8 **TEXT** column will be able to store character strings that are at a maximum 65535 bytes in size. Character strings can mix single byte and multi-byte characters provided the total size of the string value is less than or equal to the maximum for the data type.



145

6.4.2 Text Type Comparison

Data Type	Max. #bytes	#single-byte characters	#two-byte characters	#three-byte characters
CHAR (<i>L</i>) <i>L</i> <= 255	765	255	255	255
VARCHAR (<i>L</i>) <= 65533	65533	65533	32766	21844
TINYTEXT	765	255	255	255
TEXT	65536	65536	32767	21845
MEDIUMTEXT	16777216	16777216	8388608	5592405
LONGTEXT	4294967296	4294967296	2147483648	1431655765

The chart shows a comparison of the various unstructured character string types supported by MySQL.

The column “Max. #bytes” shows the maximum size a value for the type require. Note that for **CHAR** and **TINYTEXT**, the maximum size for a particular value can be calculated by taking the maximum length (*L* for **CHAR** columns, 255 for **TINYTEXT** columns) and multiplying that to the maximum number of characters for the character set that applies to that column. For the other types, the maximum size is fixed.

The remaining columns denote the length in terms of the maximum number of single, two and three byte characters that could be stored in a column of that type. Note that for **CHAR** and **TINYTEXT**, the length is equal to the maximum number of bytes..Because the size for the other types is fixed, the maximum number of characters that can be stored depends on the number of bytes that are occupied by a character.

The maximum *L* for the **VARCHAR** type is subject to the character set: at DDL time, MySQL assumes the worst and does not allow a specification for *L* that, in combination with the chosen character set, could imply a string of which the size exceeds 65533 bytes. For the remaining types, MySQL will simply fit as much characters in the available space as possible.

146

6.4.3 Structured Character String types

Apart from the unstructured character string types, MySQL also supports to structured string types. These are called structured because they do not simply allow an arbitrary character string to be entered. Rather, the possible values for these data types are derived from a statically defined list of values which is part of the definition of the data type.

The ENUM type

This **ENUM** data type is an enumeration data type. When defining a column with an **ENUM** data type, a list of values is provided specifying all available values for that column. A maximum of 65,535 distinct values can be supplied that way. When inserting or updating an **ENUM** column, exactly one value must be chosen from all available values. If the column-value is not one of the predefined values, it is truncated to the empty string.

The syntax for defining an **ENUM** data type is as follows:

```
ENUM('value1', ..., 'valueN')
```



ENUM Example:

The **Continent** column of the **Country** table in the **world** database:

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

An **ENUM** data type ensures that only valid values for the **Continent** column may be entered.

The order of the values specified for the **ENUM** data type is significant. Internally, an **ENUM** value is represented by an integer that identifies the literal character string value. This has the side effect that sorting and comparisons on an **ENUM** column will be done according to the internal integer value rather than the alphabetic value of the character string.

The SET data type

Like the **ENUM** data type, a list of values must be supplied when defining a column of the **SET** type. Unlike the **ENUM** type the **SET** type allows multiple values to be taken from the list and combined into a valid value. The character string representation of **SET** values thus takes the form of a comma-separated list of character strings, mimicking the mathematical concept of a set.

The syntax for defining a column of the **SET** type is given below:

```
SET ('value1', ..., 'valueN')
```

SET Example:

The **special_features** column of the **film** table in the **sakila** database:

```
special_features SET('Trailers','Commentaries','Deleted Scenes','Behind the
Scenes')
```

Example value:

```
'Deleted Scenes,Behind the Scenes'
```



147

6.4.4 Character Set and Collation Support

A character set is a named character repertoire that includes an encoding scheme which defines how the characters are encoded as numbers. It defines the allowable characters for the data type.

All character strings are governed by the rules of collation during comparison. A collation is a named collating sequence, which defines the character sort order and governs how individual characters and character strings can be compared to one another.

- Character strings have the following characteristics:
- A sequence of characters that belong to a specific character set.
- Multi-byte character sets may require a fixed or variable number of bytes per character.
- Comparisons are based on the collation (sorting order) of the character set associated with the string.
- Multi-byte character comparisons are performed in character units, not in byte units.
- The collation determines whether uppercase and lowercase versions of a given character are equivalent.
- The collation also determines whether to treat instances of a given character with different accent marks as equivalent.
- A collation can be a binary collation. In this case, comparisons are based on numeric character values.

148

MySQL has an extensive list of character sets from which to choose. Proper choice can have a big impact on performance. To view the available character sets use the following statement:

```
mysql> SHOW CHARACTER SET;
+-----+-----+-----+-----+
| Charset | Description | Default collation | Maxlen |
+-----+-----+-----+-----+
| big5    | Big5 Traditional Chinese | big5_chinese_ci | 2 |
| dec8    | DEC West European | dec8_swedish_ci | 1 |
| cp850   | DOS West European | cp850_general_ci | 1 |
| hp8     | HP West European | hp8_english_ci | 1 |
| koi8r   | KOI8-R Relcom Russian | koi8r_general_ci | 1 |
| latin1  | cp1252 West European | latin1_swedish_ci | 1 |
| latin2  | ISO 8859-2 Central European | latin2_general_ci | 1 |
| swe7    | 7bit Swedish | swe7_swedish_ci | 1 |
| ascii   | US ASCII | ascii_general_ci | 1 |
| ujis    | EUC-JP Japanese | ujis_japanese_ci | 3 |
...
```

The **Default collation** column contains the default collation that is used if no collation is explicitly specified. The **Maxlen** column denotes the maximum number of bytes that may be used to compose a single character.



149

A given character set may have several collations to choose from. This enables you to select different sort orders for the same character set. For example, with the latin1 character set, you can choose from any of the following collations, many of which correspond to the sorting order rules of specific languages:

```
mysql> SHOW COLLATION LIKE 'latin1%';
+-----+-----+-----+-----+-----+
| Collation      | Charset | Id | Default | Compiled | Sortlen |
+-----+-----+-----+-----+-----+
| latin1_german1_ci | latin1 | 5 |       | Yes     | 0 |
| latin1_swedish_ci | latin1 | 8 | Yes   | Yes    | 1 |
| latin1_danish_ci | latin1 | 15 |       | Yes    | 0 |
| latin1_german2_ci | latin1 | 31 |       | Yes    | 2 |
| latin1_bin        | latin1 | 47 |       | Yes    | 1 |
| latin1_general_ci | latin1 | 48 |       |        | 0 |
| latin1_general_cs | latin1 | 49 |       |        | 0 |
| latin1_spanish_ci | latin1 | 94 |       |        | 0 |
+-----+-----+-----+-----+-----+
```

Each collation name ends with _ci, _cs, or _bin, indicating whether the collation is case insensitive, case sensitive, or binary.





InLine Lab 6-B

This lab requires you to use the `mysql` command line client to practice using various string data types.

NOTE: The SQL syntax used in this lab will be covered in detail later in the course.

Step 1. Make the test database the default database

1. From the `mysql` prompt type;

```
mysql> USE test;
```

Sets the current database to be used to the `test` database.

Step 2. Create and test a table with a VARCHAR column

1. Type the following create a table that uses the `VARCHAR` data type:

```
mysql> CREATE TABLE chars (c VARCHAR(5));
```

Creates a new table called `chars` in the `test` database, having a column `c` with the `VARCHAR` data type. The column can hold character strings of at most 5 characters in length.

2. Type the following to insert a literal string into the `chars` table:

```
mysql> INSERT INTO chars VALUES ('abc');
```

Adds a new row to the `chars` table having the value 'abc' for column `c`. Note that the value needs to be enclosed in single quotes.

3. Type the following to verify the insertion was accomplished successfully:

```
mysql> SELECT * FROM chars;
```

Returns the rows in the `chars` table.

+-----+
c
+-----+
abc
+-----+

Confirms the value that was added in the previous step, and note that the single quotes used to denote the string value are not returned, as they are simply a notational device, and not themselves considered to be part of the value proper.



4. Type the following to test the maximum length specification for the column in the **chars** table:

```
mysql> INSERT INTO chars VALUES ('abcdef');
```

Due to the fact that the limit of characters allowed by the **VARCHAR** data type was exceeded, the insert causes an error to be reported, which states that the data was too long for the column.

If you do not get this error, it is due to the **SQL_MODE** setting. The output of an error message is based upon the assumption that the current SQL mode is set as '**TRADITIONAL**'. When you do not get an error, you will get a warning instead. In this case, may use the **SHOW WARNINGS** command to see the warning.

Step 3. Configure and verify a more relaxed SQL mode

1. Type the following to clear the SQL mode:

```
mysql> SET SQL_MODE := '';
```

Allows a mode relaxed **SQL_MODE** for the current session.

1. Type the following to ensure that the SQL mode has been cleared:

```
mysql> SELECT @@SQL_MODE;
```

Shows that the **SQL_MODE** settings have been cleared.

```
+-----+  
| @@SQL_MODE |  
+-----+  
|           |  
+-----+
```

Step 4. Test and verify the maximum length specification again

1. Resubmit the previous **INSERT** statement against the **c** column in the **chars** table to determine if the change in **SQL_MODE** makes a difference:

```
mysql> INSERT INTO chars VALUES ('abcdef');
```

With the newly cleared **SQL_MODE**, the insert will be completed with the result noting 1 warning.



2. Type the following to determine if the value that exceeded the maximum length specification was added:

```
mysql> SELECT * FROM chars;
```

Returns two rows. Confirms that the previous **INSERT** statement succeeded, although the value for the column **c** was truncated to fit the maximum length specified for the column.

```
+-----+
| c    |
+-----+
| abc  |
| abcde |
+-----+
```

Step 5. Create and test a table that is using the **SET** data type

1. Type the following to create a table that will use the **SET** data type:

```
mysql> CREATE TABLE sets (
->     name VARCHAR(60),
->     colors SET('red', 'blue', 'green')
-> );
```

Creates a new table called **sets**, with a column **colors** of the **SET** data type, allowing values that represent a set of one or more color names.

2. Type the following to insert a row into the **sets** table:

```
mysql> INSERT INTO sets VALUES ('Sarah', 'red, blue');
Query OK, 1 row affected, 1 warning (#.## sec)
```

(Note the space that is entered immediately following the comma in the set value)

Inserts a row into the **sets** table. Note that a warning is issued, indicating there was a problem inserting the **SET** value for the **colors** column:

```
mysql> SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message          |
+-----+-----+-----+
| Warning | 1265 | Data truncated for column 'colors' at row 1 |
+-----+-----+-----+
```



3. Type the following to verify that the row of data was added to the **sets** table:

```
mysql> SELECT * FROM sets;
```

Shows the rows in the **sets** table. Note that only the value 'red' from the inserted **SET** value was inserted.

name	colors
Sarah	red

4. Type the following to attempt to insert the values again, but this time without the space after the comma separating the colors red and blue:

```
mysql> INSERT INTO sets VALUES ('Sarah', 'red,blue');
```

(Note that there is no space between the two values in the set literal value).

Inserts values into the column **name** and the set **colors** in the **sets** table. Note that this time, no warning is issued.

5. Type the following to verify that the additional column was added correctly:

```
mysql> SELECT * FROM sets;
```

Returns the rows that are currently in the table. Confirm that the value that was inserted in the previous step is now in the table.

name	colors
Sarah	red
Sarah	red,blue

Note that the set literal 'red,blue' was now inserted as expected.

Step 6. Restore the more restrictive SQL_MODE again

1. Type:

```
mysql>SET SQL_MODE='TRADITIONAL';
```

Sets the SQL_MODE back to a more restrictive mode.



Step 7. Enter a value incompatible with the SET data type

1. Type the following to insert a value that is not contained in the **SET** definition:

```
mysql> INSERT INTO sets VALUES ('Sarah', 'orange');
```

Results in an error indicating that the value has been truncated. This is to be expected as the value 'orange' is not included in the list of values used for the definition of the **SET** data type for the **colors** column.

```
ERROR 1265 (01000): Data truncated for column 'colors' at row 1
```



150

6.5 Binary String Data Types

Binary values are usually thought of as being a sequence of bytes, which means the binary digits (bits) are grouped in eights (octets). In a way, these binary types are strings types in the sense that they are also a sequence of smaller units. Unlike characters strings, the bytes that make up a such a binary string value do not represent characters. Consequently, binary strings do not have attached character semantics and thus lack character set and collation information that is present for the character string types.

Examples of items that could be stored as binary values include things like images, sounds, movies or even executable program files.

6.5.1 Binary Types

The following is the list of binary string data types:

- **BINARY** – This data type is similar to the **CHAR** (fixed-length) type, but stores binary byte strings rather than non-binary character strings.
- **VARBINARY** – This data type is similar to the **VARCHAR** (variable-length) type, but stores binary byte strings rather than non-binary character strings.

Apart from these types, MySQL also supports a number of BLOB types

What exactly is a BLOB?

A *BLOB* is a variable-length unstructured collection of binary data stored as a single value in a database management system. *BLOBs* are typically images, audio or other multimedia objects, though sometimes binary code is stored as a blob. A commonly used explanation for the term *BLOB* is that it is an acronym for *Binary Large Object*.

In MySQL, *BLOBs* are quite similar to the **TEXT** types without attached character set and collation.

These are the *BLOB* types supported by MySQL:

- **TINYBLOB** – This data type is a *BLOB* column with a maximum length of 255 bytes.
- **BLOB** – This data type is a *BLOB* column with a maximum length of 65,535 bytes.
- **MEDIUMBLOB** – This data type is a *BLOB* column with a maximum length of 16,777,215 bytes.
- **LONGBLOB** – This data type is a *BLOB* column with a maximum length of 4,294,967,295 bytes.



151

6.5.2 Binary String Type Comparison

Note that for all types, the practical length is subject to the maximum row size and maximum packed size like discussed for the character string types.

For the storage requirement values, L represents the maximum length of a column. M represents the actual length of a given value, which may be 0 to M .

Name	Maximum #bytes	Maximum size	Required Storage
BINARY (L)	$0 \leq L \leq 255$	255B	$L + 1$
VARBINARY (L)	$0 \leq L \leq 65532$	$\sim 64kB$	$0 \leq L \leq 255: L + 1$ $256 \leq L \leq 65532: L + 2$
TINYBLOB	255	255B	$M + 1$
BLOB	65535	64kB	$M + 2$
MEDIUMBLOB	16777215	16MB	$M + 3$
LONGBLOB	4294967295	4GB	$M + 4$

152

6.6 Temporal Data Types

6.6.1 Temporal Data Types

Date and time data types are referred to as *temporal* data types. MySQL provides data types for storing different kinds of temporal information. In the following descriptions, the terms YYYY, MM, DD, hh, mm, and ss stand for a year, month, day of month, hour, minute, and second value, respectively.

The TIME type

This data type can be used for storing a time part. **TIME** values can be used to represent a time of day or an elapsed period of time. Values of the **TIME** type can range from -838:59:59 to 838:59:59. **TIME** values are retrieved as string literals in the 'HH:MM:SS' format. Literals of the **TIME** type can be denoted using either character strings or integers.

MySQL uses 3 bytes to store values of the **TIME** type.

The YEAR type

The **YEAR** data type is assigned using the following syntax:

```
YEAR [ (2|4) ]
```

The **YEAR** data type can be used to represent years in either a two-digit or a four-digit format. The default is the four-digit format.



For a four-digit **YEAR** type, allowable values are 1901 to 2155, and 0000. For a two-digit **YEAR** type, the allowable values are 70 to 69, representing years from 1970 to 2069. MySQL displays **YEAR** values in YYYY format, but allows **YEAR** columns to be assigned using either strings or numbers.

Values of the **YEAR** type require one byte of storage.

The DATE type

The **DATE** type may be used to store full calendar dates. The values for the **DATE** type range from '1000-01-01' to '9999-12-31'. MySQL retrieves **DATE** values in the 'YYYY-MM-DD' format. Literals of the **DATE** type columns can be denoted using either character strings or numbers. For example, the date January 9, 1998 would look like this;

```
'1998-01-09'
```

Values of the **DATE** type require 3 bytes of storage.

The DATETIME type

This data type stores both a date part as well as a time part.

Values for the **DATETIME** type range from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. MySQL retrieves **DATETIME** values in 'YYYY-MM-DD HH:MM:SS' format. Literal values for the **DATETIME** type can be assigned using either strings or numbers.

The **DATETIME** type requires 8 bytes of storage.

The TIMESTAMP type

The **TIMESTAMP** type stores both a date and a time part like the **DATETIME** type. The values for the **TIMESTAMP** type range from '1970-01-01 00:00:00' to partway through the year 2037. **TIMESTAMP** values are retrieved as strings in the format 'YYYY-MM-DD HH:MM:SS' with a display width fixed at 19 characters.

TIMESTAMP columns differ in a number of ways from **DATETIME** columns:

- **TIMESTAMP** columns can be defined to be automatically record the current date time whenever the row is **INSERTED** or **UPDATED**
- When storing a **TIMESTAMP** value, the value is first converted to universal coordinated time. Upon retrieval, the value is converted to local time again. The stored **TIMESTAMP** value thus unambiguously represents one point in time.
- **TIMESTAMP** values require only 4 bytes of storage



The automatic assignment of the current date and time to a **TIMESTAMP** column in response to **INSERT** and **UPDATE** events is controlled using two column attributes that are specifically available only to the **TIMESTAMP** type:

- The **DEFAULT CURRENT_TIMESTAMP** attribute causes the column to be initialized with the current date and time when the row is **INSERTED**.
- The **ON UPDATE CURRENT_TIMESTAMP** attribute causes the column to be updated with the current date and time when the value of another column in the row is **UPDATED**.
- Omitting both **DEFAULT CURRENT_TIMESTAMP** as well as **ON UPDATE CURRENT_TIMESTAMP** implicitly results in a **TIMESTAMP** column that has both **DEFAULT CURRENT_TIMESTAMP** as well as **ON UPDATE CURRENT_TIMESTAMP**.
- In order to prevent automatic value generation for a **TIMESTAMP** column you can define a **DEFAULT** clause to set the **TIMESTAMP** column to a constant value.

In a given table, only one **TIMESTAMP** column can have either or both **DEFAULT CURRENT_TIMESTAMP** and **ON UPDATE CURRENT_TIMESTAMP**. Assigning **DEFAULT CURRENT_TIMESTAMP** to one column and **ON UPDATE CURRENT_TIMESTAMP** to another is not allowed and results in an error:

ERROR 1293 (HY000) : Incorrect table definition; there can be only one TIMESTAMP column with CURRENT_TIMESTAMP in DEFAULT or ON UPDATE clause

If you need complex logic for assigning timestamps or other values, it is advisable to look into using triggers for that purpose.

153

Temporal Data Type Summary

Type	#bytes	Minimum Value	Maximum Value
DATE	3	1000-01-01	9999-12-31
TIME	3	-838:59:59	838:59:59
DATETIME	8	1000-01-01 00:00:00	9999-12-31 23:59:59
TIMESTAMP	4	1970-01-01 00:00:00	
YEAR (2)	1	(19) 70	(20) 69
YEAR (4)	1	1901	2155





InLine Lab 6-C

This lab requires you to use the mysql command line client in order to retrieve and set various temporal values data types.

NOTE: SQL statements used in this lab will be covered in detail later in the course.

Step 1. Set the default database

1. From the mysql prompt type:

```
mysql> USE test;
```

Sets the current database to be use to the **test** database.

Step 2. Set the TRADITIONAL sql_mode

1. Type:

```
mysql> SET SQL_MODE='Traditional';
```

Sets the SQL_MODE settings to the traditional settings.

Step 3. Create and test a table that is using the DATE data type

1. Type the following to create a table that uses the **DATE** data type:

```
mysql> CREATE TABLE birthdates (name VARCHAR(60), bdate DATE);
```

Creates a new table called **birthdates** in the **test** database with the column **bdate** using a **DATE** data type.

2. Type the following to enter a record into the **birthdates** table:

```
mysql> INSERT INTO birthdates VALUES ('joe', '1950-02-15');
```

Inserts values in to the columns **name** and **bdate** of the **birthdates** table.



3. Type the following to verify that the record was added to the **birthdates** table:

```
mysql> SELECT * FROM birthdates;
```

Shows a tabular output of the current values in the **birthdates** table. Confirm the values inserted in the previous step:

name	bdate
joe	1950-02-15

4. Type the following to enter an invalid date into the **birthdates** table:

```
mysql> INSERT INTO birthdates VALUES ('jane', '1950-02-30');
```

An error will result from this insert since there is no 30th of February date.

```
ERROR 1292 (22007): Incorrect date value: '1950-02-30' for column 'bdate'
at row 1
```

Step 4. Create and test a table that is using the **TIMESTAMP** data type

1. Type the following to create a table that utilizes a **TIMESTAMP** data type:

```
mysql> CREATE TABLE students (name VARCHAR(60), modtime TIMESTAMP);
```

Creates a new table called **students** in the **test** database with the column **modtime** having the **TIMESTAMP** data type.

2. Type the following record of data into the **students** table:

```
mysql> INSERT INTO students (name) VALUES ('Peter');
```

Inserts values in to the columns **name** and **modtime** in the **birthdates** table. Note that there is no need to give a value for the **modtime** column, since it is automatically filled in with the current date and time.

3. Type the following to enter in a second record into the **students** table:

```
mysql> INSERT INTO students (name) VALUES ('Tariq');
```

Inserts values in to the columns **name** and **modtime** in the **birthdates** table. Note that there is no need to give a value for the **modtime** column, since it is automatically filled in with the now current date and time. As some time has passed since the previous insert, we should be able to see this reflected in the values stored in the **modtime** column.



4. Type the following to verify that the values were added along with reviewing the values assigned to the timestamp column:

```
mysql> SELECT * FROM students;
```

Returns the rows in the **students** table.

NOTE: Your dates and times will differ according to the date/time that this lab is completed.

name	modtime
Peter	2010-10-14 20:02:21
Tariq	2010-10-14 20:02:29



154

6.7 NULLs

6.7.1 The Meaning of **NULL**

In SQL, expressions can evaluate to the *null-value*. The *null-value* is a special value that has the purpose to represent the fact that a value cannot be computed or is not known. There is also a **NULL** keyword that denotes the *null-value* when used in an expression. The **NULL** keyword can also appear in table column definitions to denote whether the column is *nullable* or not: a nullable column may contain the *null-value*, whereas an error occurs when a *null-value* is inserted into a column that is not nullable.

For nullable columns, different meanings may be attached to the occurrence of a *null-value*, such as; “no value”, “unknown value”, “missing value”, “out of range”, “not applicable”, “none of the above”, and so on. However, all the reasons for using **NULL** can be split into two categories:

- **Unknown:** There is a value, but the precise value is unknown at this time. The *null-value* can then be interpreted as: “secret”, “figure not available”, “to be determined”, “impossible to calculate”, and “partially unknown”
- **Not applicable:** If a value would be specified it would not mean anything.: “undefined”, “moot”, “irrelevant”, “none” and “n/a”

Is NULL a value?

The term *null-value* is defined in the SQL standard. However, some people dislike the notion of a *null-value*. To them, **NULL** primarily denotes the absence of a value, in which case it would be strange to talk about **NULL** as if it were a value.

Talking about a *null-value* becomes less strange when you realize that in SQL statements, a **NULL** may be inserted in most places where a value can appear. So in that limited, syntactical sense, **NULL** is indeed a *value-expression*

155

6.7.2 When to Use NULL

In the beginning stages of database design, when you are making a list of the data that will be included, it becomes clear that some data may not be available for all columns. These are the cases to scrutinize and determine whether *null-values* should be allowed. Also, this can be changed for an existing table if a problem is detected due to occurrences of *null-values* in the column.

For instance, the **world** database has a **Country** table that contains a column for life expectancy:

```
`LifeExpectancy` FLOAT(3,1) DEFAULT NULL
```

For the countries that have a population of zero (0), there will be no value for this column. Therefore, it has been set to **DEFAULT NULL**. The **DEFAULT NULL** simply means that if no value is supplied for the **LifeExpectancy** column, the column will automatically have the *null-value*.

In order to allow the column to store *null-values*, it must be nullable. In MySQL, columns are nullable by default (except **TIMESTAMP** columns) unless something extra is done to make them not nullable.



156

6.7.3 When NOT to Use NULL

There are times when you will not want to allow *null-values* in a column. The most common case is when it is a primary key. Also for any column that must have a value in order for the database design to make sense.

Columns that are in the primary key are made not-nullable by default. **TIMESTAMP** columns are also not-nullable by default. It is possible to explicitly specify that a column is not-nullable by using **NOT NULL**:

Here's an example of the **Name** column in the **City** table in the **world** database:

```
Name CHAR(35) NOT NULL
```

The **NOT NULL** ensures that an attempt to store a **NULL** value in the **Name** column results in an error.

Another example in the **world** is the **CountryLanguage** table which contains a column that indicates (for each language) whether it is the “official” language or not:

```
`IsOfficial` ENUM('T', 'F') NOT NULL DEFAULT 'F'
```

Not only will this column not allow **NULL** values, but it has a default value set to '**F**'. This ensures that the default will be inserted whenever there is no explicit value given for the **IsOfficial** column





Quiz

In this exercise you will answer questions pertaining to MySQL Data Types.

1. If you want to store monetary values (for example, values representing U.S. dollar-and-cent amounts such as \$48.99), which data type should you use to avoid rounding errors?

2. Which data type is more space-efficient: **CHAR**(100) or **VARCHAR**(100)?

3. In a table population, you want to store the number of inhabitants of cities. Storage space is at a premium. You expect the maximum population to be 15,000,000 for a city. Which data type (and desired column attributes) would you use? What's the storage requirement for this data type?

4. You perform the following **INSERT** operation on table datetest, which has a single **DATE** column called d with a default value of **NULL**:

```
INSERT INTO datetest VALUES ('12:00:00');
```

What data value will actually be stored in the table? Provide a short explanation.

5. Explain whether the following statements are true or false: “The character set of a column is determined by the character set of its table. The same holds for its collation”:

6. Each character set has exactly one collation. (True or False)



7. Here's the structure of a table continent that has only one column (**Name**, which stores names of continents). Assume that the SQL mode has no input data restrictions enabled.

```
mysql> DESCRIBE continent\G
***** 1. row ****
Field: name
Type:
enum('Africa','America','Antarctica','Asia','Australia','Europe')
Null: YES
Key:
Default: NULL
Extra:
```

What string value will be stored by the following **INSERT** operation? What integer value will be stored internally?

```
INSERT INTO continent VALUES ('Africa');
```

8. The following **CREATE TABLE** statement shows the definition for table defaults;

```
mysql> CREATE TABLE defaults (
->     id INT UNSIGNED NOT NULL UNIQUE,
->     col1 INT NULL,
->     col2 INT NOT NULL,
->     col3 INT DEFAULT 42,
->     col4 CHAR(5) NULL,
->     col5 CHAR(5) NOT NULL,
->     col6 CHAR(5) DEFAULT 'yoo',
->     col7 TEXT NULL,
->     col8 TEXT NOT NULL,
->     col9 TIME NOT NULL,
->     col10 DATE NULL
-> );
```

What's the effect on the columns with a **TEXT** data type if you issue this **INSERT** statement? Why?

```
mysql> INSERT INTO defaults (id) VALUES (1);
```



157

6.8 Chapter Summary

This chapter introduced the data types supported by MySQL. In this chapter, you learned to:

- Describe the three major categories of data types
- Understand character sets and collation
- Assign the appropriate data type to table entities
- Understand the meaning and use of **NULL** and **NOT NULL**



7 SQL EXPRESSIONS

159

7.1 Learning Objectives

This chapter introduces the use of SQL Expressions in MySQL. In this chapter, you will learn to:

- Use components of expressions
- Use numeric, string, and temporal values in expressions
- Properties of **NULL** values
- Types of functions that can be used in expressions
- Write comments in SQL statements



7.2 SQL Comparisons

This chapter discusses how to use expressions in SQL statements. Expressions can appear in many places as part of SQL statements, such as the **SELECT** list, the **ORDER BY** and **GROUP BY** clauses. The conditions that appear in the **WHERE** and **HAVING** clause are also special cases of (boolean) SQL expressions.

160

7.2.1 Components of SQL Expressions

Expressions are a common element of SQL statements, and they occur in many contexts. For example, expressions often occur in the **WHERE** clause of **SELECT**, **DELETE**, or **UPDATE** statements to identify which records to retrieve, delete, or update. But expressions may be used in many other places; for example, in the output column list of a **SELECT** statement, or in **ORDER BY** or **GROUP BY** clauses.

Terms of expressions consist of literals (numbers, strings, dates, and times), built-in constants like **NULL**, **TRUE** and **FALSE**, references to table columns, and function calls. Terms may be combined using operators into more complex expressions. Many types of operators are available, such as those for arithmetic, comparison, logical, and pattern-matching operations.

Here are some examples of expressions:

- The following statement refers to table columns to select country names and populations from the **Country** table:

```
mysql> SELECT Name, Population FROM Country;
```

- You can work directly with literal data values that aren't stored in a table. The following statement refers to several literal values: an integer, an exact-value decimal value, an approximate-value floating-point value in scientific notation, and a string value:

```
mysql> SELECT 14, -312.82, 4.32E-03, 'I am a string';
```

- Another way to produce data values is by invoking functions. This statement calls functions that return the current date and a server version string:

```
mysql> SELECT CURDATE(), VERSION();
```



All these different types of expressions can be combined into more complex expressions to produce other values of interest. The following statement demonstrates this:

162

```
mysql> SELECT Name,
->   TRUNCATE(Population/SurfaceArea,2) AS `people/sq. Km`,
->   IF(GNP > GNPOld,'Increasing','Not increasing') AS `GNP Trend`
->   FROM Country
->   ORDER BY Name LIMIT 10;
```

Name	people/sq. km	GNP Trend
Afghanistan	34.84	Not increasing
Albania	118.31	Increasing
Algeria	13.21	Increasing
American Samoa	341.70	Not increasing
Andorra	166.66	Not increasing
Angola	10.32	Not increasing
Anguilla	83.33	Not increasing
Antarctica	0.00	Not increasing
Antigua and Barbuda	153.84	Increasing
Argentina	13.31	Increasing

The expressions in the preceding statement use these types of values:

- References to table columns: Name, Population, SurfaceArea, GNP, and GNPOld. (“GNP” means “gross national product.”)
- Literal values, such as the string literals 'Increasing', 'Not increasing', and integer literals 2 and 10.
- Function calls: The numeric function `TRUNCATE()` is used here to format the population/area ratio to two decimal places, and the conditional function `IF()` returns its second argument when the condition that is passed as first argument is true or the third argument otherwise.



7.2.2 Numeric Expressions

163

Numbers can be exact-value literals or approximate-value literals. Exact-value literals are used just as given in SQL statements when possible and thus are not subject to the inexactness produced by rounding error. On the other hand, approximate-value literals are subject to rounding error and may not necessarily be used exactly as given.

Exact-value literals are written with no exponent. Approximate-value literals are written in scientific notation with an exponent. For example, the numeric values `-43`, `368.93`, and `.00214` are exact values, whereas `-4.3E1`, `3.6893E2`, and `2.14E-3` are approximate values. Even though the two sets of numbers look like they have the same values, internally they are represented in different ways:

- Exact-value numbers are either integer values or decimal values. Decimal values have a fractional part (decimal places), whereas integer values are whole numbers and do not have a fractional part. Internally, these are represented using the `INT` or `DECIMAL` data type respectively. Operations on integers are performed with the precision of `BIGINT` values (that is, 64 bits). Operations on decimal values have a precision of up to 64 decimal digits (that is, a `DECIMAL` can have at most 64 digits, including decimal places). Currently, the scale for decimal values allows up to 30 decimal digits after the decimal point.
- Approximate-value literals are represented as floating-point numbers (like the `DOUBLE` data type) and have a mantissa and exponent. The mantissa allows up to 53 bits of precision, which is about 15 decimal digits.
- With a few exceptions, almost all numerical expression that include a part that evaluates to `NULL`, will itself evaluate to `NULL`.

When numbers are used in an arithmetic or comparison operation, the result of the operation may depend on whether it involves exact or approximate values. Consider the following two comparisons:

```
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 |
+-----+-----+
```

In the first expression, exact values are used, so the comparison involves exact calculations. In the second expression, approximate values are used and rounding error is possible. This illustrates that if you use approximate values in comparisons, you cannot expect exact-value precision. The internal representation of floating-point numbers inherently allows for the possibility of rounding error.

164

If you mix numbers with strings in numeric context, MySQL converts the strings to numbers and performs a numeric operation:

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



7.2.3 String Expressions

Literal strings in expressions are written as quoted values. By default, MySQL allows either single quotes or double quotes. The notation with single quotes is defined by the SQL standard and is the preferred notation (at least in this guide). The meaning of double-quoted strings is dependent upon the SQL mode: if the `ANSI_QUOTES` SQL mode is enabled, double quotes are interpreted as identifier-quoting characters (according to the SQL standard). For this reason, it is best to reserve single quotes to denote string literals as this notation is not only portable across RDBMSes but also independent of the SQL mode.

165

What are server SQL Modes?

Server SQL modes define what SQL syntax MySQL should support and what kind of data validation checks it should perform. This makes it easier to use MySQL in different environments and to use MySQL together with other database servers. The MySQL Server applies these modes individually to different clients.

The data types for representing strings in tables include `CHAR`, `VARCHAR` and the `TEXT` and `BLOB` types. You choose which type to use depending on factors such as the maximum length of values and whether you require fixed-length or variable-length values.

Direct use of strings in expressions occurs often in comparison operations. Otherwise, most string operations are performed by using functions.

The usual comparison operators like `=`, `<>`, `<`, `BETWEEN . . . AND`, and so forth can be applied to string values:

- `<` Less than
- `<=` Less than or equal to
- `=` Equal to
- `<=>` Equal to (works even for NULL values)
- `<>` or `!=` Not equal to
- `>=` Greater than or equal to
- `>` Greater than
- `BETWEEN . . . AND` test if a string falls into a range of values

In MySQL, string concatenation is usually achieved using the `CONCAT()` function:

```
mysql> SELECT CONCAT('abc', 'def', REPEAT('X', 3));
+-----+
| CONCAT('abc', 'def', REPEAT('X', 3)) |
+-----+
| abcdefXXX |
+-----+
```

166



By default, the `||` operator is treated as the logical `OR` operator. This behavior is not according to the SQL standard. In standard SQL, the `||` operator is used for string concatenation, and this interpretation maybe enabled by including `PIPES_AS_CONCAT` in the SQL mode:

```
mysql> SELECT 'abc' || 'def';
+-----+
| 'abc' || 'def' |
+-----+
|          0 |
+-----+
1 row in set, 2 warnings (#.## sec)
```

In this `SELECT` statement, `||` performs a logical `OR` operation. This is a boolean operation, and because MySQL uses numbers for boolean operations, MySQL converts the strings in the expression to numbers first. Neither can be interpreted as a number, so MySQL converts them to zero (which is interpreted as FALSE). This explains why there is a warning count of two. The resulting operands for the operation are zero, so the result also is zero.

After `PIPES_AS_CONCAT` is enabled, `||` produces a string concatenation instead:

167

```
mysql> SET sql_mode = 'PIPES_AS_CONCAT';
Query OK, 0 rows affected (#.## sec)

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

168 Case Sensitivity in String Comparisons

String comparisons are somewhat more complex than numeric or temporal comparisons. That is because depending on the purpose of the comparison, letters may be compared literally (binary comparison), or take the meaning of a letter as it appears in human readable text into account (for example, for some purposes it is convenient to disregard differences in letter case or accentuation)

String expressions contain characters from a particular character set, which is associated with one of the collations (sorting orders) available for the character set. Characters may consist of single or multiple bytes. A collation can be case insensitive (lettercase is not significant) or case sensitive (lettercase is significant).

The rules that govern string comparison apply in several ways. They determine the result of comparisons performed explicitly with operators such as `=` and `<`, and comparisons performed implicitly by `ORDER BY`, `GROUP BY`, and `DISTINCT` operations, and even the behavior of `UNIQUE` indexes



The default character set and collation for literal strings depend on the values of the `character_set_connection` and `collation_connection` system variables. The default character set is `latin1`. The default collation is `latin1_swedish_ci`, which is case insensitive as indicated by the “`_ci`” at the end of the collation name. Assuming these connection settings, literal strings are not case sensitive by default. You can see this by comparing strings that differ only in lettercase:

```
mysql> SELECT 'Hello' = 'hello';
+-----+
| 'Hello' = 'hello' |
+-----+
|           1 |
+-----+
```

A given collation might cause certain accented characters to compare the same as other characters. For example, ‘`ü`’ and ‘`ue`’ are different in the default `latin1_swedish_ci` collation, but with the `latin1_german2_ci` collation (“German phone-book” collation), they have the same sort value and thus compare as equal:

```
169
mysql> SELECT 'Müller' = 'Mueller';
+-----+
| 'Müller' = 'Mueller' |
+-----+
|           0 |
+-----+
mysql> SET collation_connection = latin1_german2_ci;
mysql> SELECT 'Müller' = 'Mueller';
+-----+
| 'Müller' = 'Mueller' |
+-----+
|           1 |
+-----+
```

169



170

Using LIKE for Pattern Matching

Operators such as = and != are useful for finding values that are equal to or not equal to a specific exact comparison value. When it's necessary to find values based on similarity instead, a pattern match is useful. To perform a pattern match, use

```
expression LIKE 'pattern'
```

where *expression* is the expression of which you want to test the value and '*pattern*' is a string literal that denotes a pattern that describes the general form of values that you want to match. In string literals that are used as patterns for the **LIKE** pattern-matching operator, the characters _ and % have a special meaning ("metacharacters"), denoting a *wildcard* rather than the literal occurrence of the characters themselves:

- The '%' character matches any sequence of *zero or more characters*. For example,
 - the pattern 'a%' matches any string that begins with 'a',
 - '%b' matches any string that ends with 'b', and
 - '%c%' matches any string that contains a 'c'.

So, the pattern '%' matches any string, including empty strings.

- The '_' (underscore) character matches *any single character*. For example,
 - 'd_g' matches strings such as 'dig', 'dog', and 'd@g'.
 - Because '_' matches any single character, it matches itself and the *pattern* 'd_g' also matches the *string* 'd_g'.

171

A pattern can use these metacharacters in combination. For example, '_%' matches any string containing at least one character. For example, to list all countries with names that start with 'United';

```
mysql> SELECT Name FROM Country
      -> WHERE Name LIKE 'United%';
+-----+
| Name           |
+-----+
| United Arab Emirates   |
| United Kingdom    |
| United States Minor Outlying Islands |
| United States     |
+-----+
4 rows in set (#.## sec)
```



To invert the results of the pattern match, use **NOT LIKE** rather than **LIKE**:

```
mysql> SELECT Name FROM Country
-> WHERE Name NOT LIKE 'United%';
+-----+
| Name
+-----+
| Aruba
| Afghanistan
| Angola
...
| South Africa
| Zambia
| Zimbabwe
+-----+
235 rows in set (#.# sec)
```

NOTE: The **NOT LIKE** results in a list which is 4 short of the total Country list (239), since the specified pattern matched items have been left out.

172 Pattern matching with regular expressions

The **LIKE** operator offers a powerful way to compare string expressions using a pattern. However, there is an even more powerful pattern matching operator: **REGEXP** or **RLIKE**. As is the case with **LIKE**, **RLIKE** (or **REGEXP**) takes on two string operands, the left hand operand being a string operand to match and the right operand being a pattern. Again similar to **LIKE**, **RLIKE** will evaluate to **TRUE** if the left hand operand matches the pattern and to **FALSE** otherwise.

The difference between **LIKE** and **RLIKE** is that the pattern for **RLIKE** can be a regular expression. Where the expressiveness of the **LIKE** patterns is limited to the occurrence of catch-all wildcard characters % and _, regular expressions offer many possibilities to precisely define specific patterns of particular character sequences as well as how many times the pattern can occur in order to match. With regular expressions you can test for complex patterns such as:

- URLs
- IP addresses
- E-mail addresses
- Phone numbers
- Postal codes



- 173 From the MySQL point of view, the regular expression in an **R_LIKE** expression is just a character string, but it has its own syntax to allow a declarative denotation of quite complex patterns. The following chart describes the most common syntax elements in MySQL regular expressions:

Syntax Element	Notation	Description
Literal text	As is	Each literal character matches an occurrence of that character as is.
Character Class	[spec]	Matches one character specified by the character class specification given by spec. Character class specifications allow efficient definition of a group of characters, each of which is considered to match. The character class specification – the part inbetween the square brackets has its own mini-syntax and is discussed later
Wildcard (period, dot)	.	Matches any character (including carriage return and newline)
Anchors: match a specific position in the string – not the occurrence of a particular character		
Start of input anchor	^	Matches the position right at the start of the left hand operand. Note that the ^ can appear inside square brackets too, where it has an entirely different meaning
End of input anchor	\$	Matches the position right at the end of the left hand operand
Start of word anchor	[[:< :]]	Matches the position right before the start of a word
End of word anchor	[[:> :]]	Matches the position right at the end of a word
Occurrence Indication: specify the number of times the preceding unit may occur.		
Optional Occurrence	?	Specifies that an occurrence of the preceding unit is optional
Repeated Occurrence	+	Specifies that the occurrence of the preceding unit must appear at least once, but may be repeated any number of times
Optional Repeated Occurrence	*	Specifies that the preceding unit is optional and may be repeated any number of times
Specific occurrence indication	{ n }	The n denotes an integer that indicates that the preceding unit is to appear exactly n times
Minimum and Maximum Occurrence specification	{ m, n }	The m and the n denote integers, indicating that the preceding unit must appear at least m and at most n times



Operators, delimiters, escaping

Syntax Element	Notation	Description
Parenthesis	(and)	Parenthesis are used to group a sequence of elements. This is useful if you want to apply an operator or occurrence indication to a sequence of units
Alternation (Choice)		Indicates that a match with the unit appearing on either the left hand or the right hand side of the pipe. The pipe has a lower precedence than a sequence of characters.
Escape	\	In order to specify a match to a character that has a special meaning in the regular expression syntax (so-called metacharacters), such a character needs to be escaped. This is done by preceding that character with a backslash. However, because MySQL regular expressions are denoted as a character string the backslash itself has to be escaped using another backslash. We will illustrate this in the examples later in this section

174 The following chart described the syntax for character class specifications, that is, the notation inside the square brackets [and]. Almost all (negation is an exception) of the following elements may occur multiple times inside square brackets, and maybe freely mixed with other elements of the character class syntax. Also, the characters that are metacharacters inside character classes are generally not metacharacters outside that context and vice versa (the ^ is an exception).

A few examples

175

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

In the previous example, all city names that contain the string 'nat' are matched. The actual matching is collation-aware and this means that by default the comparison is done in a case insensitive manner.



An important thing to note is that the `RLIKE` comparison is considered `TRUE` if at least one match is found of the right-hand pattern in the left-hand operand. If the pattern is intended to completely and exactly describe the entire left-hand operand, be sure to use the start and end anchors `^` and `$` respectively. For example, to find all cities that start with 'New' and end with 'rk', you could use something like this:

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

Note that the anchors do not match a character – they match a particular position of the text.

Except for the anchors, the previous example introduced another powerful feature of regular expressions: *occurrence indicators*. The sequence `.*` denotes “any character, any number of times” and is thus equivalent to `%` in the normal `LIKE` patterns. A standalone dot (period) matches the occurrence of any character and is thus equivalent to the `_` in normal `LIKE` patterns. The `*` (asterisk) denotes the period may be matched any number of times, including zero times. More fine grained control is possible with the other occurrence indicators like `?` (question mark, may occur once) and `+` (plus-sign, must occur at least once) or the exact `{m, n}` notation.

Sometimes you need to know match one entry from a list of alternatives. The *choice* or *alternation* operator `|` is useful in these cases. For example, to find all cities that have either 'Los' or 'Las' in the city name, you can do something like this:

176

```
mysql> SELECT Name FROM City
      -> WHERE Name RLIKE ' Los | Las ';
+-----+
| Name   |
+-----+
| Las Heras   |
| ..... |
| East Los Angelos |
| Las Heras   |
+-----+
```

Note that a sequence of characters has a higher precedence than the alternation operator. So in this case, a choice is made for either the sequence 'Las' or the sequence 'Los', not for the characters immediately left and right of the operator. This is worth noting as the occurrence indicators have a higher precedence than a character sequence.



The previous query could also have been written using a character class like this:

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

Note that the character class [ao] matches exactly one character: either 'a' or 'o'.

177 Character classes offer convenient features to match an entire range of characters. The following example illustrates this feature by matching Argentinean postal codes:

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

Here's the explanation for this moderately complex pattern:

- Argentinian postal codes consist of one letter to indicate the province, 4 digits for the municipality and for the larger cities a 3 letter code to denote the side of the street block.
- Not all letters are valid province letters, and the regular expression uses three ranges to denote only valid province letters: A–H, J–N, and P–Z.
- To match four digits, the range [0–9] is used, and in order to demand exactly 4 occurrences, the curly braces are used to denote the exact number of occurrences: {4}. Note that the occurrence indicator ‘binds’ to the element that immediately precedes it, in this case the [0–9] character class.
- Finally yet another range is used to test for a 3 letter sequence: [A–Z] {3}.
- However, because the trailing three letters do not appear in every postal code, an appropriate occurrence indicator must be applied to that entire part of the pattern. To do that, we must enclose that part of the pattern in parenthesis and add the optional occurrence indicator ? directly after the closing right parenthesis.

178 If you want to match characters that are part of the regular expression syntax themselves, they need to be escaped. The escape character is the backslash character. Escaping is not as straightforward as it may seem because the regular expression is itself a string, and the backslash is already used as the escape character for special characters in ordinary MySQL strings. So, the following example might not be intuitive for many people:

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

In order to understand what is happening exactly it is good to realize that MySQL will first unescape the string that makes up the regular expression just like any ordinary string. Any backslashes that are then present in the pattern are interpreted in the context of the regular expression syntax. So, the following example does work:

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



What happens here is that the pattern '\\\\' is first interpreted as a MySQL string. Because the backslash character is an escape character in that context, MySQL will take away the first backslash, and use the value (not the semantics) of the character immediately following that backslash. So the character value of the first \\ pair is a single backslash: \\. The same goes for the second pair of backslashes. This means that after the ordinary MySQL string unescaping, the string representation of the regular expression '\\\\' actually denotes the regular expression \\\. This is then subject to further interpretation, but now in the context of the regular expression syntax. In the regular expression syntax, the backslash is also an escape character, and unescaped in the same way we just described.

To complicate things further, the backslash is *not* an escape character inside a character class definition. So, in order to match a literal backslash in a character set definition, no ‘extra’ escaping is possible on top of the regular MySQL string escaping:

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

One might wonder how one can then escape characters inside a character class. There is no escape character for escaping character class meta characters, but one can use named characters using the [.name .] notation.

179

7.2.4 Temporal Expressions

Temporal values include dates, times, and date/time values that have both a date and time. More specialized temporal types are timestamp (commonly used for recording “current date and time”) and year (for temporal values that require a resolution only to year units).

Direct use of temporal values in expressions occurs primarily in comparison operations, or in arithmetic operations that add an interval to or subtract an interval from a temporal value. Otherwise, most temporal value operations are performed by using functions.

Temporal data may be generated via any of the following means:

- Copying data from an existing **DATE**, **DATETIME**, or **TIME** column
- Executing a built-in function that returns a **DATE**, **DATETIME**, or **TIME** column
- Building a string representation of the temporal data to be evaluated by the server



Date Components:

- **DATE** YYYY-MM-DD
- **TIME** HH:MM:SS
- **DATETIME** YYYY-MM-DD HH:MI:SS
- **TIMESTAMP** YYYY-MM-DD HH:MI:SS
- **DAY** DD
- **MONTH** MM
- **QUARTER** Q
- **YEAR** YYYY

180 The usual comparison operators may be applied to temporal values ($=$, $<$, $>$, **BETWEEN . . . AND**, and so forth).

To perform interval arithmetic, use the **INTERVAL** keyword and a unit value:

```
mysql> SELECT '2010-01-01' + INTERVAL 10 DAY, INTERVAL 10 DAY + '2010-01-01';
+-----+-----+
| '2010-01-01' + INTERVAL 10 DAY | INTERVAL 10 DAY + '2010-01-01' |
+-----+-----+
| 2010-01-11                   | 2010-01-11                   |
+-----+-----+
```

For addition of temporal and interval values, you can write the operands in either order, as just shown. To subtract an interval from a temporal value, the interval value must be second (it doesn't make sense to subtract a temporal value from an interval):

```
mysql> SELECT '2010-01-01' - INTERVAL 10 DAY;
+-----+
| '2010-01-01' - INTERVAL 10 DAY |
+-----+
| 2009-12-22                     |
+-----+
```





InLine Lab 7-A

In this exercise you will use SQL comparisons. This will require a `mysql` command line client and access to a MySQL server.

Step 1. SQL numeric expressions

1. Use a `SELECT` statement to add the following together; 2, 3, and `NULL`:

```
mysql> SELECT 2+3+NULL;
```

The result shows a value of `NULL`:

```
+-----+
| 2+3+NULL |
+-----+
|      NULL |
+-----+
1 row in set (#.# sec)
```

Step 2. Automatic Conversion

1. Issue the following `SELECT` statement:

```
mysql> SELECT 2+3+'6';
```

The string ‘6’ is converted into a number:

```
+-----+
| 2+3+'6' |
+-----+
|      11 |
+-----+
1 row in set (#.# sec)
```



Step 3. Greater than

1. Issue a **SELECT** statement that will find the populations of the three least populated countries. Hint: The population must be greater than zero (0):

```
SELECT Population
FROM Country
WHERE Population > 0
ORDER BY Population ASC LIMIT 3;
```

3 rows are returned:

```
+-----+
| Population |
+-----+
|      50 |
|     600 |
|   1000 |
+-----+
3 rows in set (#.# sec)
```

Step 4. Using Parenthesis

1. Use a numeric expression to find the average of the values you found in the previous step three populations.

```
SELECT (50+600+1000)/3;
```

Returns the average numeric value;

```
+-----+
| (50+600+1000)/3 |
+-----+
|    550.0000 |
+-----+
1 row in set (#.# sec)
```



Step 5. SQL string expressions

1. Use a **SELECT** statement to concatenate the Name and corresponding District from the City table with the words ' is in ' between the name and the district, limiting the output to the first five rows found:

```
SELECT CONCAT(Name, ' is in ', District)
FROM City
LIMIT 5;
```

Returns a list of five cities and their districts with string;

```
+-----+
| CONCAT(Name, ' is in ', District) |
+-----+
| Kabul is in Kabul
| Qandahar is in Qandahar
| Herat is in Herat
| Mazar-e-Sharif is in Balkh
| Amsterdam is in Noord-Holland
+-----+
5 rows in set (#.# sec)
```

Step 6. Looking for a piece of text

1. Issue a query for all cities that include the word 'City' in their name:

```
SELECT Name FROM City WHERE Name LIKE '%city%';
```

Returns a list of 12 city names.

Step 7. More advanced LIKE patterns

1. Issue a query for all cities that include 'City' in the name, and have 'a' as the second letter:

```
SELECT Name FROM City WHERE Name LIKE '_a%city%';
```

Returns a list consisting of 4 cities, which fit the query criteria;

```
+-----+
| Name
+-----+
| Kansas City
| Salt Lake City
| Kansas City
| Daly City
+-----+
4 rows in set (#.# sec)
```



Step 8. SQL temporal expressions

1. Show the current date (default format) with the column title ‘TODAY’:

```
SELECT CURRENT_DATE() AS TODAY;
```

Returns a single column with the current date as you specified.

Step 9. Using INTERVAL calculation

1. Show the current date and time (default format) as it will be ten years from now:

```
SELECT NOW() + INTERVAL 10 YEAR;
```

Returns a single column with 10 years added to the current date and time.



181 7.3 Functions in SQL Expressions

Functions can be invoked within expressions and return a value that is used in place of the function call when the expression is evaluated. When you invoke a function, there must be no space after the function name and before the opening parenthesis. It's possible to change this default behavior by enabling the `IGNORE_SPACE` SQL mode to cause spaces after the function name to be ignored:

```
mysql> SELECT PI ();
ERROR 1305 (42000): FUNCTION world.PI does not exist
mysql> SET sql_mode = 'IGNORE_SPACE';
Query OK, 0 rows affected (#.## sec)
mysql> SELECT PI ();
+-----+
| PI ()      |
+-----+
| 3.141593   |
+-----+
1 row in set (#.## sec)
```

182 7.3.1 Comparison Functions

Comparison functions enable you to test relative values or membership of one value within a set of values.

`LEAST()` and `GREATEST()` take a set of values as arguments and return the one that is smallest or largest, respectively:

```
mysql> SELECT LEAST(4,3,8,-1,5), LEAST('cdef','ab','ghi');
+-----+-----+
| LEAST(4,3,8,-1,5) | LEAST('cdef','ab','ghi') |
+-----+-----+
|          -1 | ab           |
+-----+-----+
```

```
mysql> SELECT GREATEST(4,3,8,-1,5), GREATEST('cdef','ab','ghi');
+-----+-----+
| GREATEST(4,3,8,-1,5) | GREATEST('cdef','ab','ghi') |
+-----+-----+
|             8 | ghi           |
+-----+-----+
```



- 183 **INTERVAL()** takes only integer expressions as arguments. The value of the first argument is compared to the value of the subsequent arguments, and returns the index of the last argument that has a value that is equal to or less than the first argument:

```
mysql> SELECT INTERVAL(10, 1, 2, 4, 8, 16);
+-----+
| INTERVAL(10, 1, 2, 4, 8, 16) |
+-----+
|                               4 |
+-----+
1 row in set (#.## sec)
```

7.3.2 Control Flow Functions

Control flow functions enable you to choose between different values based on the result of an expression.

- 184 **IF()** tests the expression in its first argument and returns its second or third argument depending on whether the expression is true or false:

```
mysql> SELECT IF(1 > 0, 'yes','no');
+-----+
| IF(1 > 0, 'yes','no') |
+-----+
| yes                  |
+-----+
```

- 185 This example uses the **IF()** construct to specify the order of the list of countries from the `Country` table in the `world` database using the code value to place the USA in the first row of the results:

```
mysql> SELECT name FROM country
    -> ORDER BY IF(code='USA',1,2), name
    -> LIMIT 3;
+-----+
| name           |
+-----+
| United States |
| Afghanistan   |
| Albania        |
+-----+
```



186

Suppose that you have a *Sales* table containing rows as shown below:

division	sale	syear
A	100	1999
A	140	2000
A	10	2001
A	122	2003
B	12	2003
B	100	2002
B	80	2001
C	200	2002
C	230	2003
C	20	2001
C	120	2000
A	20	2002

A query can be constructed using **IF()** that lists the sales data by division and year:

```
mysql> SELECT division,
-> SUM(IF(syear=1999,sale,0)) AS year99,
-> SUM(IF(syear=2000,sale,0)) AS year00,
-> SUM(IF(syear=2001,sale,0)) AS year01,
-> SUM(IF(syear=2002,sale,0)) AS year02,
-> SUM(IF(syear=2003,sale,0)) AS year03,
-> SUM(IF(syear=2004,sale,0)) AS year04
-> FROM sales
-> GROUP BY division;
+-----+-----+-----+-----+-----+-----+-----+
| division | year99 | year00 | year01 | year02 | year03 | year04 |
+-----+-----+-----+-----+-----+-----+-----+
| A        |    100 |    140 |     10 |     20 |    122 |      0 |
| B        |      0 |      0 |     80 |    100 |     12 |      0 |
| C        |      0 |    120 |     20 |    200 |    230 |      0 |
+-----+-----+-----+-----+-----+-----+-----+
```



If there are NULL's in the IF expression

If either side of the expression tested is **NULL**, the result type of the **IF()** function is the type of the non- **NULL** expression.

187

```
mysql> SELECT IF(1 > NULL, 'yes','no');
+-----+
| IF(1 > NULL, 'yes','no') |
+-----+
| no                         |
+-----+  
  
mysql> SELECT IF(NULL = NULL, 'yes','no');
+-----+
| IF(NULL = NULL, 'yes','no') |
+-----+
| no                         |
+-----+  
  
mysql> SELECT IF(NULL <> NULL, 'yes','no');
+-----+
| IF(NULL <> NULL, 'yes','no') |
+-----+
| no                         |
+-----+
```



188

The **CASE** construct is an operator rather than a function, but it too provides flow control. It has two forms of syntax, the *simple case expression* and the *searched case expression*. The syntax model for the simple case expression is shown below:

```
CASE case_expr
    WHEN when_expr THEN result_expr
    [WHEN when_expr THEN result_expr] ...
    [ELSE result_expr]
END
```

For the simple **CASE** expression, the expression *case_expr* is evaluated and compared to the *when_expr* of each of the subsequent **WHEN** clauses. As soon as a **WHEN** clause is found that has a *when_expr* that equals the *case_expr*, the *result_expr* of that **WHEN** clause is returned. If no **WHEN** clause has a *when_expr* equal to *case_expr*, and there is an **ELSE** clause, the expression in the **ELSE** clause becomes the **CASE** result. If there is no **ELSE** clause, the result is **NULL**.

189

In this example, a simple **CASE** expression is used to specify the order of the first three rows in the list of countries from the *Country* table (world database) using the code values:

```
mysql> SELECT Name
-> FROM Country
-> ORDER BY
-> CASE code
->   WHEN 'USA' THEN 1
->   WHEN 'CAN' THEN 2
->   WHEN 'MEX' THEN 3
->   ELSE 4 END, Name;
-> LIMIT 3
+-----+
| name      |
+-----+
| United States |
| Canada     |
| Mexico     |
+-----+
4 rows in set (#.# sec)
```



190 The syntax model for the searched **CASE** expression looks like this:

```
CASE
  WHEN when_cond THEN result_expr
  [WHEN when_cond THEN result_expr] ...
  [ELSE result_expr]
END
```

For this syntax, the conditional expression in each **WHEN** clause is evaluated until one is found to be true, and then its corresponding **THEN** expression becomes the result of the **CASE**. If none of the **WHEN** clauses are true and there is an **ELSE** clause, its expression becomes the **CASE** result. If there is no **ELSE** clause, the result is **NULL**.

In the following example, we group the countries into three groups, USA, Europe and the rest of the world. Note that the criteria is based on different columns in the two rows:

```
mysql> SELECT
->      CASE
->        WHEN Code = 'USA' THEN 'United States'
->        WHEN Continent = 'Europe' THEN 'Europe'
->        ELSE 'Rest of the world'
->      END AS Area,
->      SUM(GNP),
->      SUM(Population)
-> FROM Country
-> GROUP BY Area;
+-----+-----+
| Area           | SUM(GNP)    | SUM(Population) |
+-----+-----+
| Europe         | 9498865.00 | 730074600      |
| Rest of the world | 11345342.90 | 5070317850     |
| United States   | 8510700.00 | 278357000      |
+-----+-----+
3 rows in set (#.## sec)
```



191

7.3.3 Numeric Functions

Numeric functions perform several types of mathematical operations, such as rounding, truncation, trigonometric calculations, or generating random numbers.

192

The `ROUND()` function performs rounding of its argument. The rounding method applied to the fractional part of a number depends on whether the number is an exact or approximate value:

- For positive exact values, `ROUND()` rounds up to the next integer if the fractional part is .5 or greater, and down to the next integer otherwise. For negative exact values, `ROUND()` rounds down to the next integer if the fractional part is .5 or greater, and up to the next integer otherwise. Another way to state this is that a fraction of .5 or greater rounds away from zero and a fraction less than .5 rounds toward zero:

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

- For approximate values, `ROUND()` uses the rounding method provided in the C library used by the MySQL server. This can vary from system to system, but typically rounds to the nearest even integer:

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



- 193 The **FLOOR()** function returns the largest integer not greater than its argument, and the **CEILING()** function returns the smallest integer not less than its argument:

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

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

NOTE: **TRUNCATE(<number>, <decimals>)** is similar to **FLOOR()**. However, for truncate one can specify the number of decimals to truncate to.

- 194 The **ABS()** and **SIGN()** functions extract the absolute value and sign of numeric values:

```
mysql> SELECT ABS(-14.7), ABS(14.7);
+-----+-----+
| ABS(-14.7) | ABS(14.7) |
+-----+-----+
|      14.7 |      14.7 |
+-----+-----+

mysql> SELECT SIGN(-14.7), SIGN(14.7), SIGN(0);
+-----+-----+-----+
| SIGN(-14.7) | SIGN(14.7) | SIGN(0) |
+-----+-----+-----+
|          -1 |           1 |          0 |
+-----+-----+-----+
```



195 A family of functions performs 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 |
+-----+-----+-----+
```

196 **7.3.4 String Functions**

String functions calculate string lengths, extract pieces of strings, search for substrings or replace them, perform lowercase conversion, and more.

197 The functions **INSTR()**, **LOCATE()** and **POSITION()** are used to find the position of the occurrence of a string inside another string. All these can accept two arguments with functionally equivalent results:

```
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
```

Note that all these functions return the position of the beginning of the first occurrence of the substring 'and' in the string 'Alice and Bob', where 1 is considered to be the position of the first character (and 0 would be returned if the substring was not found). These functions differ in how the arguments are passed: **INSTR()** needs the string as first argument and the substring that is to be searched as second argument, whereas it is the other way around for **LOCATE()** and **POSITION()**. In addition, **POSITION()** requires the **IN** keyword rather than a comma to separate the arguments.

One might wonder why there are so many different ways to support the same functionality and the answer is compatibility and standards compliance: **INSTR()** and **LOCATE()** appear in other popular RDBMS products and **POSITION()** is defined by the SQL standard.



LOCATE() can optionally accept a third argument to specify the offset from where the search should start:

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

So in the previous example, we find that looking from the 7th position of the string 'Alice and Bob', the first occurrence of the substring ' ' is found at position 10.

Note that **INSTR()**, **LOCATE()** and **POSITION()** take the collation of the argument strings into account. This means that for searching the substring, the collation of the argument is used in the string comparison – not the binary value of the characters or bytes that make up the string. An explicit **BINARY** keyword can be added before one of the argument strings to specify a binary search. Alternatively, the **COLLATE** keyword can be used to specify a specific collation that is to be used for searching the substring.

- 198 The **LENGTH()** (or the synonym **OCTET_LENGTH()**) and **CHAR_LENGTH()** functions determine string lengths in byte and character units, respectively. The values returned by the two functions will differ for strings that contain multi-byte characters. The following example shows this, using the latin1 single-byte character set and the ucs2 multi-byte character set:

```
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 |
+-----+
```

- 199 **CONCAT()** and **CONCAT_WS()** concatenate strings – that is, they take on a number of string arguments and ‘glue’ them together into a new string. **CONCAT()** concatenates all of its arguments, whereas **CONCAT_WS()** interprets its first argument as a separator to place between the following arguments:

```
mysql> SELECT CONCAT('See', 'spot', 'run');
+-----+
| CONCAT('See', 'spot', 'run') |
+-----+
| Seespotrun |
+-----+
```



```
mysql> SELECT CONCAT_WS(' ', 'See', 'spot', 'run');
+-----+
| CONCAT_WS(' ', 'See', 'spot', 'run') |
+-----+
| See spot run                         |
+-----+
```

200 **SUBSTRING()** and **SUBSTRING_INDEX()** are used to obtain a part of a string (a substring). **SUBSTRING()** takes on a string argument from which a part is to be taken, an integer argument to indicate the offset from where to take a part of the string argument. The first character of the string argument is considered to have offset 1. Optionally, **SUBSTRING()** can take on a third integer argument to indicate the length of the part that is to be returned:

```
mysql> SELECT SUBSTRING('Alice and Bob', 1, 5);
+-----+
| SUBSTRING('Alice and Bob', 1, 5) |
+-----+
| Alice                            |
+-----+
```

So the previous example requires 5 characters to be returned from the string 'Alice and Bob', starting from the start of that string. The last argument is optional. If it is omitted, the string part that is returned extend from the offset argument unto the end of the string argument. The three argument version of **SUBSTRING()** is equivalent to the **MID()** function.

SUBSTRING_INDEX() returns a part of the first string argument up to a particular occurrence of the second string argument in the first string argument. The particular number of occurrences of the second string argument is specified by the (mandatory) third integer argument:

```
mysql> SELECT SUBSTRING_INDEX('Alice and Bob', 'and', 1);
+-----+
| SUBSTRING_INDEX('Alice and Bob', 'and', 1) |
+-----+
| Alice                                         |
+-----+
```

Normally the number of occurrences of the second string argument are specified when reading the first string argument from left to right. By using a negative number for the third argument, the number of occurrences of the second string argument are counted from right to left:

```
mysql> SELECT SUBSTRING_INDEX('Alice and Bob', 'and', -1);
+-----+
| SUBSTRING_INDEX('Alice and Bob', 'and', -1) |
+-----+
| Bob                                           |
+-----+
```



Note that **SUBSTRING_INDEX()** uses a binary comparison when searching the substring. This means that it does not take the collation of the string arguments into account.

- 201 **LEFT()** and **RIGHT()** are like **SUBSTRING()** in that they return a part of the first string argument. **LEFT()** takes the number of characters specified by the second integer argument from the start of the first string argument:

```
mysql> SELECT LEFT('Alice and Bob', 5);
+-----+
| LEFT('Alice and Bob', 5) |
+-----+
| Alice                      |
+-----+
```

RIGHT() can be thought of as the opposite of **LEFT()**. **RIGHT()** takes the number of characters specified by the second integer argument from the end of the first string argument:

```
mysql> SELECT RIGHT('Alice and Bob', 3);
+-----+
| RIGHT('Alice and Bob', 3) |
+-----+
| Bob                         |
+-----+
```

- 202 The **LTRIM()**, **RTRIM()** and **TRIM()** functions can be used to remove space characters appearing at the extremities of an argument string. **LTRIM()** removes leading spaces (that is, spaces occurring on the left side of the argument string), **RTRIM()** removes trailing spaces (that is, spaces occurring on the right side of the argument string), and **TRIM()** removes spaces from both ends of the argument string:

```
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>
```



For the **TRIM()** function there is another variant available. This optionally allows the usage of the keywords **TRAILING**, **LEADING** or **BOTH** (the default) to specify at which end(s) the characters are to be removed, and thus effectively unites **LTRIM()** and **RTRIM()** to the single **TRIM()** function. This variant is even more flexible in that it optionally allows you to specify the substring that is to be trimmed off:

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

- 203 The functions **INSERT()** and **REPLACE()** (not to be confused with the **REPLACE** and **INSERT** statement) can be used to replace pieces of an existing string. **REPLACE()** globally replaces occurrences of a particular substring with another string. The first argument to **REPLACE()** is the string wherein pieces will be replaced. The second argument is the string that appears in the first string argument that is to be replaced. The third argument is the string that will be used to replace each occurrence of the second argument in the first argument:

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

Note that **REPLACE()** uses a binary search for replacing occurrences of the second argument. That is, it does not take the collation into account.

The **INSERT** function can be used for a positional replacement of a piece of an existing string with another string. The first argument to **INSERT** is the string wherein the replacement is to take place. The second argument is the offset within the first argument where the newly inserted string will appear. The third argument is the number of characters of the first string that must be overwritten. Finally, the last argument is the string that will be placed into the first string argument at the specified position:

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



- 204 The **CHARSET()** and **COLLATE()** functions respectively return information about the character set and collation of a string argument:

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

CONVERT() provides a way to convert data between different character sets. The syntax is:

```
mysql> SELECT CONVERT(_latin1'Müller' USING utf8);  
+-----+  
| CONVERT(_latin1'Müller' USING utf8) |  
+-----+  
| Müller |  
+-----+
```

You may also use **CAST()** to convert a string to a different character set.

```
mysql> SELECT CAST(_latin1'test' AS CHAR CHARACTER SET utf8);  
+-----+  
| CAST(_latin1'test' AS CHAR CHARACTER SET utf8) |  
+-----+  
| test |  
+-----+
```



205 The **STRCMP()** function compares two strings and returns -1 , 0 , or 1 if the first string is less than, equal to, or greater than the second string, respectively:

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

If you just want to find out if two strings are equal and are not interested to know if one of the strings is ‘bigger’ than the other one, it may be better to use the equals operator $=$:

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

Note that both the equals operator as well as the **STRCMP()** function take the collation of the strings into account. In particular, if you need a case sensitive comparison you typically need to add a **COLLATE** clause and provide a case sensitive collation to ensure a case sensitive comparison.



7.3.5 Temporal Functions

206 Temporal functions perform operations such as extracting parts of dates and times, reformatting values, or converting values to seconds or days. In many cases, a temporal function that takes a date or time argument also can be given a date-type argument and will ignore the irrelevant part of the datetime value.

Functions for retrieving temporal data:

- **NOW()** - Current date and time as set on the client host (in **DATETIME** format)
- **CURDATE()** - Current date as set on the client host (in **DATE** format)
- **CURTIME()** - Current time as set on the client host (in **TIME** format)
- **YEAR()** - Year in **YEAR** format, per value indicated (can use **NOW()** function within parenthesis to get current year per client)
- **MONTH()** - Month of the year in integer format, per value indicated (can use **NOW()** as above)
- **DAY()** - Day of the month in integer format, per value indicated (can use **NOW()** as above)
- **DAYOFMONTH()** - Synonym for **DAY()**
- **DAYNAME()** - (English) Day of the week in string format, per value indicated (can use **NOW()** as above)
- **HOUR()** - Hour of the Day in integer format, per value indicated (can use **NOW()** as above)
- **MINUTE()** - Minute of the Day in integer format, per value indicated (can use **NOW()** as above)
- **SECOND()** - Second of the Minute in integer format, per value indicated (can use **NOW()** as above)
- **GET_FORMAT()** - Returns a date format string, per values indicated for date-type and international format.



207 Obtain the current date and time according to the server:

```
mysql> SELECT NOW();
+-----+
| NOW()           |
+-----+
| 2004-04-30 11:59:15 |
+-----+
1 row in set (#.# sec)
```

Obtain the date format for a specified date-type and international standard:

```
mysql> SELECT GET_FORMAT(DATE, 'EUR');
+-----+
| GET_FORMAT(DATE, 'EUR') |
+-----+
| %d.%m.%Y           |
+-----+
1 row in set (#.# sec)
```

There are several choices for the two arguments;

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

208 The functions YEAR(), MONTH(), DAYOFMONTH() and DAYOFYEAR() may be used for extracting parts of DATE or DATETIME values:

```
mysql> SELECT YEAR('2010-04-15'), MONTH('2010-04-15') ,
      -> DAYOFMONTH('2010-04-15');
+-----+-----+-----+
| YEAR('2010-04-15') | MONTH('2010-04-15') | DAYOFMONTH('2010-04-15') |
+-----+-----+-----+
|          2010 |                 4 |            15 |
+-----+-----+-----+

mysql> SELECT DAYOFYEAR('2010-04-15');
+-----+
| DAYOFYEAR('2010-04-15') |
+-----+
|             105 |
+-----+
```



Likewise, the functions HOUR, MINUTE and SECOND may be used to extract time parts from TIME or DATETIME values:

```
mysql> SELECT HOUR('09:23:57'), MINUTE('09:23:57'), SECOND('09:23:57');
+-----+-----+-----+
| HOUR('09:23:57') | MINUTE('09:23:57') | SECOND('09:23:57') |
+-----+-----+-----+
|         9 |          23 |         57 |
+-----+-----+-----+
```

NOTE: The functions YEAR(), MONTH(), DAY(), HOUR(), MINUTE(), SECOND() should not be confused with the similar keywords used in conjunction with the INTERVAL syntax

- 209 **MAKEDATE()** and **MAKETIME()** compose dates and times from component values. **MAKEDATE()** produces a date from year and day of year arguments:

```
mysql> SELECT MAKEDATE(2010,105);
+-----+
| MAKEDATE(2010,105) |
+-----+
| 2010-04-15          |
+-----+
```

MAKETIME() produces a time from hour, minute, and second arguments.

```
mysql> SELECT MAKETIME(9,23,57);
+-----+
| MAKETIME(9,23,57) |
+-----+
| 09:23:57          |
+-----+
```

- 210 If you need to determine the current date use **CURRENT_DATE()** or the synonym **CURDATE()**. If you need to obtain the current time, use **CURRENT_TIME()** or the synonym **CURTIME()**. To obtain the current datetime, use **CURRENT_TIMESTAMP()** or **NOW()**:

```
mysql> SELECT CURRENT_DATE(),
->      CURRENT_TIME(),
->      CURRENT_TIMESTAMP();
+-----+-----+-----+
| CURRENT_DATE() | CURRENT_TIME() | CURRENT_TIMESTAMP() |
+-----+-----+-----+
| 2005-05-31     | 21:40:18     | 2005-05-31 21:40:18 |
+-----+-----+-----+
```



The functions `CURRENT_DATE()`, `CURRENT_TIME()` and `CURRENT_TIMESTAMP()` return their result with respect to the local timezone (of the server). In contrast, the functions `UTC_DATE()`, `UTC_TIME()` and `UTC_TIMESTAMP()` respectively return a result with respect to the coordinated universal time.

NOTE: These functions can be invoked with or without parentheses following the function name. However, we will always write the parenthesis to avoid confusion.

211

7.3.6 `NULL`-Related Functions

Functions intended specifically for use with `NULL` values include `ISNULL()` and `IFNULL()`. `ISNULL()` is true if its argument is `NULL` and false otherwise:

```
mysql> SELECT ISNULL(NULL), ISNULL(0), ISNULL(1);
+-----+-----+-----+
| ISNULL(NULL) | ISNULL(0) | ISNULL(1) |
+-----+-----+-----+
|          1 |        0 |        0 |
+-----+-----+-----+
```

`IFNULL()` takes two arguments. If the first argument is not `NULL`, that argument is returned; otherwise, the function returns its second argument:

```
mysql> SELECT IFNULL(NULL, 'a'), IFNULL(0, 'b');
+-----+-----+
| IFNULL(NULL, 'a') | IFNULL(0, 'b') |
+-----+-----+
|      a           |      0       |
+-----+-----+
```

The `IFNULL()` function is MySQL specific. MySQL also supports the `COALESCE()` function which is defined by the SQL standard and has a similar purpose as `IFNULL()`. `COALESCE()` is actually more flexible in that it can take on multiple arguments and return the value of the first argument that is not `NULL`.

212

Other functions handle `NULL` values in various ways, so you have to know how a given function behaves. In many cases, passing a `NULL` value to a function results in a `NULL` return value. For example, any `NULL` argument passed to `CONCAT()` causes it to return `NULL`:

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



But not all functions behave that way. `CONCAT_WS()` (concatenate with separator) simply ignores `NULL` arguments entirely, that is, it does not output a separator for a `NULL` argument:

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

The fact that `CONCAT_WS()` ignores `NULL` turns out to be very convenient when pretty printing addresses and names, as any optional parts do not need special handling. Consider for example a table that contains international addresses that has a mandatory `address_line1` column and a nullable `address_line2` column. `CONCAT_WS()` allows us to obtain the full address without any issues:

```
mysql> SELECT CONCAT_WS('\n', address_line1, address_line2) AS address  
-> FROM Addresses
```

If `CONCAT_WS()` would behave like `CONCAT()` does, we would have to take extra care to prevent a `NULL` `address_line2` to render the entire address to `NULL`. However, this would lead to other complications, because if we would change `NULL` arguments to say, an empty string, then we would end up with extra empty lines due to the separator being output for each argument.





InLine Lab 7-B

In this exercise you will use SQL functions. This will require a `mysql` command line client and access to a MySQL server.

Step 1. SQL Control Flow Functions

1. Use the `IFNULL()` function to produce an alternative to producing a NULL output with the following SQL statement:

```
mysql> SELECT IFNULL(headOfState, 'Anarchy') FROM Country  
-> WHERE Name = 'San Marino'
```

Because the country of San Marino does not have an authority figure identified as the head of state, the resulting output will identify that that country is facing an anarchy situation versus producing a NULL value:

```
+-----+  
| IFNULL(headOfState, 'Anarchy') |  
+-----+  
| Anarchy |  
+-----+  
1 row in set (#.# sec)
```

Step 2. IF in ORDER BY

1. Use the `IF()` function to specify the order of the list of country codes and languages from the `CountryLanguage` table, using the code value to place languages of Norway in the first rows of the results. Limit the results to the first 10:

```
mysql> SELECT CountryCode, Language FROM CountryLanguage  
-> ORDER BY IF(CountryCode='NOR',1,2), Language LIMIT 10;
```

Returns a list of 10 country codes and corresponding languages, listing all languages that are spoken in Norway first:

```
+-----+-----+  
| CountryCode | Language |  
+-----+-----+  
| NOR | Danish |  
| NOR | English |  
| NOR | Norwegian |  
...  
| ERI | Afar |  
+-----+-----+  
10 rows in set (#.# sec)
```



Step 3. SQL numeric functions

1. Round the number 3.75 to the nearest integer.

```
mysql> SELECT ROUND(3.75);
```

The result shows a value of 4.

Step 4. Using SIGN

1. Show the sign of the following numbers; -80, -(-15.4), 0.

```
mysql> SELECT SIGN(-80), SIGN(-(-15.4)), SIGN(0);
```

Returns a value of -1, 1, 0 respectively to indicate whether the numbers are negative or positive;

SIGN(-80)	SIGN(-(-15.4))	SIGN(0)
-1	1	0

Step 5. SQL string functions

1. Use a **SELECT** statement to show the string length of the city called 'Dallas':

```
mysql> SELECT CHAR_LENGTH(Name) FROM City WHERE Name = 'Dallas';
```

Returns a length value of 6.

Step 6. Using CONCAT

1. Concatenate the following individual words; 'This ', 'is ', 'a ', 'great ', 'class! '.

```
mysql> SELECT CONCAT('This ', 'is ', 'a ', 'great ', 'class!');
```

Returns a continuous string connecting the specified words;

This is a great class!	
------------------------	--

Step 7. SQL temporal functions

1. Use a **SELECT** statement to show the current date.

```
mysql> SELECT CURDATE();
```

Returns a table with the current date



Step 8. CURDATE and date calculation

1. Use a **SELECT** statement to show the date, 21 days from today.

```
mysql> SELECT CURDATE() + INTERVAL 21 DAY
```

Returns a table with the date in 21 days.

Step 9. NOW and datetime calculation

1. Use a **SELECT** statement to show the current date/time, and the date/time in exactly 5 hours from now.

```
mysql> SELECT NOW(), NOW() + INTERVAL 5 HOUR
```

Returns a table with the current date/time and the date/time in 5 days.

213

7.3.7 Comments in SQL Statements

MySQL supports three forms of comment syntax. One of those forms has variants that allow special instructions to be passed through to the MySQL server.

- A /* sequence begins a comment that ends with a */ sequence. This style is the same as that used for writing comments in the C programming language. A C-style comment may occur on a single line or span multiple lines:

```
/* this is a comment */  
/*  
   this is a comment,  
   spanning multiple lines  
*/
```

- A -- (double dash) sequence followed by a space (or control character) begins a comment that extends to the end of the line. This syntax requires a space and thus differs from standard SQL syntax, which allows comments to be introduced by -- without the space. MySQL disallows a double dash without a space as a comment because it's ambiguous. (For example, does 1--3 mean “one minus negative three” or “one followed by a comment”?)
- A # character begins a comment that extends to the end of the line. This commenting style is like that used by several other programs, such as Perl, Awk, and several Unix shells. It is not defined by the SQL standard but a MySQL specific convenience feature.



214 For the C-style comments, there is a MySQL specific variant of the comment syntax that allows them to be interpreted as part of the SQL statement, but ignored by other database products. This is an aid to writing more portable SQL because it enables you to write comments that are treated as part of the surrounding statement if executed by MySQL and ignored if executed by other database servers. There are two ways to write embedded SQL in a C-style comment:

- If the comment begins with /* ! rather than with /*, MySQL executes the body of the comment as part of the surrounding query. The following statement creates a table named t, but for MySQL creates it specifically as a MEMORY table:

```
CREATE TABLE t (i INT) /*! ENGINE = MEMORY */;
```

- If the comment begins with /* ! followed by an integer, the number will be interpreted as a version number, allowing embedded SQL to be executed only on servers of a particular version. The server executes the body of the comment as part of the surrounding query if its version is at least as recent as that specified in the query. Otherwise, it ignores the comment. For example, the FULL keyword for SHOW TABLES was added in MySQL 5.0.2. To write a comment that's understood only by servers from MySQL 5.0.2 and up and ignored by older servers, write it as follows:

```
SHOW /*!50002 FULL */ TABLES;
```

215 **7.3.8 Comments on database objects**

Apart from the code comments discussed in the previous section, MySQL also has (limited) support for comments attached to database objects. These database object comments apply to tables, columns, and other objects like stored routines and are specified using the MySQL specific COMMENT clause that may be part of the DDL statements that create or change these database objects.

The comments can be used to provide a short description of the purpose of the object, allowing database users a quick inspection. Comments like this can be retrieved using the SHOW CREATE TABLE syntax, and are also available through the information_schema database.

Table Comments

Comments can be added to the CREATE TABLE statement with the COMMENT keyword (up to 60 characters of free form text). For example:

```
CREATE TABLE `CountryLanguage` (
  ...
) ENGINE=MYISAM COMMENT 'Lists Languages Spoken'
```



Column Comments

Column comments can be included in **CREATE TABLE** statements too:

```
CREATE TABLE `CountryLanguage` (
    CountryCode CHAR(3) NOT NULL
        COMMENT 'The code that identifies the Country',
    Language CHAR(30) NOT NULL
        COMMENT 'The name of the language spoken in the Country',
    ...
)
```



Further Practice

In this exercise, you will exercise the SQL expressions syntax covered in this chapter using the `world` database.

216

1. What weekday will it be 1 year and 2 months from now?
2. What are the lengths of the longest and shortest city names in the `City` table?
3. List the first 5 result rows of populations from the `Country` table, rounded up to 6 places to the left of the decimal point.
4. List all the country names where the life expectancy cannot be calculated.
5. Which is the most common two-character combination at the start of a country's name?
6. Select the current date and set the output format to result in the following format; `month_name date, year(4 digit)`.
7. List the countries in Asia with their region listed after the name within parenthesis (i.e.; "Cyprus (Middle East)").
8. List the name of the countries that have the letters a, c and s contained in their names.
9. List the average life expectancy in each continent, round to the closest integer.
10. Create the following result by querying the `Country` table:

WorldPop	EuropePop
6078749450	730074600

11. In the records for each country, there is a field that holds the type of government form that is used by the leadership to lead the country (`GovernmentForm`). Using this field, find the distribution of the 5 most common government forms for each continent. The following is an example of the resulting output for the top government form used by countries per continent:

GovernmentForm	Eu	Af	As	NA	SA	Oc	An	Total
Republic	25	46	26	10	9	6	0	122



217

7.4 Chapter Summary

This chapter introduced the use of SQL Expressions in MySQL. In this chapter, you've learned to:

- Use Components of expressions
- Use numeric, string, and temporal values in expressions
- Properties of **NULL** values
- Types of functions that can be used in expressions
- Write comments in SQL statements





8 OBTAINING METADATA

8.1 Learning Objectives

219

This chapter explains how to obtain *metadata* using the MySQL database server. In this chapter, you will learn to:

- List the various metadata access methods available
- Recognize the structure of the INFORMATION_SCHEMA database/schema
- Use the available commands to view metadata
- The differences between **SHOW** statements and INFORMATION_SCHEMA tables
- Use the mysqlshow client program



220

8.2 Metadata Access Methods

A databases is a structured collection of data. Data that describes the structure of the database is referred to as *metadata*: metadata is “data about data”. This chapter discusses the various means by which MySQL provides access to metadata for databases, tables, and other objects managed by the database server. The following topics will be covered:

- Using the `INFORMATION_SCHEMA` database to access metadata
- Using `SHOW` and `DESCRIBE` statements to access metadata
- Using the `mysqlshow` program to access metadata

MySQL provides metadata for several aspects of database structure. To name a few, you can obtain the names of databases, tables, columns, table indexes and stored routine definitions.

One method by which MySQL makes metadata available is through a family of `SHOW` statements, each of which displays one kind of information. For example, `SHOW DATABASES` and `SHOW TABLES` return lists of database and table names, and `SHOW COLUMNS` produces information about definitions of columns in a table.

A client program, `mysqlshow`, acts as a command-line front end to a few of the `SHOW` statements. When invoked, it examines its arguments to determine what information to display, issues the appropriate `SHOW` statement, and displays the results that the statement returns.

`SHOW` and `mysqlshow` have been available since very early releases of MySQL. As of MySQL 5, metadata access is enhanced through the `INFORMATION_SCHEMA`.

From the database user's point of view, the `INFORMATION_SCHEMA` is a database (schema) containing a number of objects that appear to be tables. The tables in the `INFORMATION_SCHEMA` contain data about all objects managed by the database server that are available to the current user. The `INFORMATION_SCHEMA` and its contents are automatically present and available to all users.

The `INFORMATION_SCHEMA` is defined in the SQL standard. MySQL implements a subset of all the `INFORMATION_SCHEMA` features described in the standard, and offers a number of extensions to the standard as well. Therefore, it provides better compliance with standard SQL because `INFORMATION_SCHEMA` is standard, not a MySQL-specific extension like `SHOW`.



221

8.3 The INFORMATION_SCHEMA database

The INFORMATION_SCHEMA database serves as a central repository for database metadata. It is a “virtual database” in the sense that it is not stored on disk anywhere, but it contains tables like any other database, and the contents of its tables can be accessed using `SELECT` like any other tables. Furthermore, you can use `SELECT` to obtain information about INFORMATION_SCHEMA itself. For example, to list the names of its tables, use the following statement:

222

```
mysql> SELECT TABLE_NAME
-> FROM INFORMATION_SCHEMA.TABLES
-> WHERE TABLE_SCHEMA = 'INFORMATION_SCHEMA'
-> ORDER BY TABLE_NAME;
+-----+
| Tables_in_information_schema |
+-----+
| CHARACTER_SETS                |
| COLLATIONS                     |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMNS                         |
| COLUMN_PRIVILEGES              |
| ENGINES                         |
| EVENTS                          |
| FILES                           |
| KEY_COLUMN_USAGE               |
| PARTITIONS                      |
| PLUGINS                         |
| PROCESSLIST                     |
| REFERENTIAL_CONSTRAINTS        |
| ROUTINES                        |
| SCHEMATA                        |
| SCHEMA_PRIVILEGES              |
| STATISTICS                      |
| TABLES                          |
| TABLE_CONSTRAINTS              |
| TABLE_PRIVILEGES               |
| TRIGGERS                        |
| USER_PRIVILEGES                |
| VIEWS                           |
+-----+
```

NOTE: More tables are also available in the 5.1.12 (and later) versions. Please refer to the latest version of the online MySQL Reference Manual.



223

8.3.1 INFORMATION_SCHEMA Tables

The tables shown in that list contain the following types of information:

- CHARACTER_SETS - Information about available character sets
- COLLATIONS - Information about collations for each character set
- COLLATION_CHARACTER_SET_APPLICABILITY - Information about which collations are applicable to a particular character set
- COLUMNS - Information about columns in tables and views
- COLUMN_PRIVILEGES - Information about column privileges held by MySQL user accounts
- ENGINES - Information about storage engines
- EVENTS - Information about scheduled events
- FILES - Information about the files in which MySQL NDB Disk Data tables are stored
- KEY_COLUMN_USAGE - Information about constraints on key columns
- PARTITIONS - Information about table partitions

224

- PLUGINS - Information about server plugins
- PROCESSLIST - Information about which threads are running
- REFERENTIAL_CONSTRAINTS -- Information about foreign keys
- ROUTINES - Information about stored procedures and functions
- SCHEMATA - Information about databases
- SCHEMA_PRIVILEGES - Information about database privileges held by MySQL user accounts
- STATISTICS - Information about table indexes
- TABLES - Information about tables in databases
- TABLE_CONSTRAINTS - Information about constraints on tables
- TABLE_PRIVILEGES - Information about table privileges held by MySQL user accounts
- TRIGGERS - Information about triggers in databases
- USER_PRIVILEGES - Information about global privileges held by MySQL user accounts
- VIEWS - Information about views in databases

225



226

8.3.2 Displaying INFORMATION_SCHEMA Tables

To display the names of the columns in a given INFORMATION_SCHEMA table, use a statement of the following form, where the TABLE_NAME comparison value names the table in which you're interested:

```
mysql> SELECT COLUMN_NAME
    -> FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_SCHEMA = 'INFORMATION_SCHEMA'
    -> AND TABLE_NAME = 'VIEWS';
+-----+
| COLUMN_NAME      |
+-----+
| TABLE_CATALOG   |
| TABLE_SCHEMA    |
| TABLE_NAME      |
| VIEW_DEFINITION |
| CHECK_OPTION    |
| IS_UPDATABLE    |
+-----+
```

The names of the INFORMATION_SCHEMA database, its tables, and columns are not case sensitive:

```
mysql> SELECT column_name
    -> FROM information_schema.columns
    -> WHERE table_schema = 'information_schema'
    -> AND table_name = 'views';
+-----+
| column_name      |
+-----+
| TABLE_CATALOG   |
| TABLE_SCHEMA    |
| TABLE_NAME      |
| VIEW_DEFINITION |
| CHECK_OPTION    |
| IS_UPDATABLE    |
+-----+
```



227

When you retrieve metadata from INFORMATION_SCHEMA by using **SELECT** statements, you have the freedom to use any of the usual **SELECT** features:

- You can specify in the select list which columns to retrieve.
- You can restrict which rows to retrieve by specifying conditions in a **WHERE** clause.
- You can group or sort the results with **GROUP BY** or **ORDER BY**.
- You can use **JOINS**, **UNIONS**, and subqueries.
- You can retrieve the result of an INFORMATION_SCHEMA query into another table with **CREATE TABLE...SELECT** or **INSERT...SELECT**. This enables you to save the result and use it in other statements later.
- You can create views on top of INFORMATION_SCHEMA tables

228

The following examples demonstrate how to exploit various features of **SELECT** to pull out information in different ways from INFORMATION_SCHEMA:

Display the storage engines used for the tables in a given database:

```
mysql> SELECT TABLE_NAME, ENGINE  
-> FROM INFORMATION_SCHEMA.TABLES  
-> WHERE TABLE_SCHEMA = 'world';
```

Find all the tables that contain **SET** columns:

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME  
-> FROM INFORMATION_SCHEMA.COLUMNS  
-> WHERE DATA_TYPE = 'set';
```

Display the default collation for each character set:

```
mysql> SELECT CHARACTER_SET_NAME, COLLATION_NAME  
-> FROM INFORMATION_SCHEMA.COLLATIONS  
-> WHERE IS_DEFAULT = 'Yes';
```

Display the number of tables in each database:

```
mysql> SELECT TABLE_SCHEMA, COUNT(*)  
-> FROM INFORMATION_SCHEMA.TABLES  
-> GROUP BY TABLE_SCHEMA;
```



The INFORMATION_SCHEMA is read-only. Its tables cannot be modified with statements such as **INSERT**, **DELETE**, or **UPDATE**. Execute these types of statements in an attempt to change the data in the INFORMATION_SCHEMA tables results in an error:

```
mysql> DELETE FROM INFORMATION_SCHEMA.VIEWS;  
ERROR 1288 (HY000): The target table VIEWS of the DELETE is not updatable
```





InLine Lab 8-A

In this exercise you will use the methods covered in this chapter for obtaining metadata. This will require a mysql command line client and access to a MySQL server.

Step 1. The SCHEMATA table

1. Use the **SELECT** statement to obtain schemata information about the test schema:

```
mysql> SELECT *  
-> FROM INFORMATION_SCHEMA.SCHEMATA  
-> WHERE SCHEMA_NAME= 'test'\G
```

Returns the properties for the test database:

```
***** 1. row *****  
 CATALOG_NAME: NULL  
 SCHEMA_NAME: test  
 DEFAULT_CHARACTER_SET_NAME: latin1  
 DEFAULT_COLLATION_NAME: latin1_swedish_ci  
 SQL_PATH: NULL
```

Step 2. Change the default database

1. Change the database to the INFORMATION_SCHEMA database:

```
mysql> USE INFORMATION_SCHEMA;
```

Shows that database has changed.

Step 3. The TABLES table

1. Select the name and engine of the tables in the world schema:

```
mysql> SELECT TABLE_NAME, ENGINE  
-> FROM TABLES  
-> WHERE TABLE_SCHEMA = 'world';
```

Lists all the tables in the world database, by name and engine;

table_name	engine
City	InnoDB
Country	MyISAM
CountryLanguage	MyISAM



Step 4. Using GROUP BY

1. List for each schema (database) the number of tables per storage engine:

```
mysql> SELECT TABLE_SCHEMA, ENGINE, COUNT(*)
-> FROM TABLES
-> GROUP BY TABLE_SCHEMA, ENGINE
```

Lists the number of tables per storage engine for each schema:

table_schema	engine	count(*)
information_schema	MEMORY	12
information_schema	MyISAM	4
mysql	MyISAM	17
test	InnoDB	11
world	NULL	3
world	InnoDB	12
world	MyISAM	2

Step 5. Table data length

1. List the data length of the City table in the world database? :

```
mysql> SELECT TABLE_NAME, DATA_LENGTH
-> FROM tables
-> WHERE TABLE_NAME = 'City';
```

Returns a row with the table name and data length:

table_name	data_length
city	376832



Step 6. The COLUMNS table / Data types

- How many table columns in the world database are using the CHAR or the VARCHAR data types?

```
mysql> SELECT DATA_TYPE, COUNT(*)
-> FROM INFORMATION_SCHEMA.COLUMNS
-> WHERE TABLE_SCHEMA = 'world'
-> AND DATA_TYPE IN ('CHAR', 'VARCHAR')
-> GROUP BY DATA_TYPE;
```

Returns a row with the data type name and the number of times that the data type was used when defining a table column;

DATA_TYPE	COUNT(*)
char	20

1 row in set (0.00 sec)

NOTE: There are no VARCHAR data types in the world database, therefore it does not show up in the output..



229

8.4 Using `SHOW` and `DESCRIBE`

The contents of the `INFORMATION_SCHEMA` tables are accessed using ordinary `SELECT` statements. In addition, MySQL also supports the `SHOW` and `DESCRIBE` syntax which form an alternative means of accessing metadata. The `SHOW` and `DESCRIBE` syntax are not as flexible as using `INFORMATION_SCHEMA` queries, but for many purposes the `SHOW` and `DESCRIBE` syntax is sufficient. In those cases, it is often quicker and easier to use this MySQL specific syntax.

8.4.1 SHOW Statements

MySQL supports a set of `SHOW` statements that each return one kind of metadata. This section describes a few of them;

- `SHOW DATABASES` – provides a list of the available databases.
- `SHOW [FULL] TABLES` – provides a list of the available tables in a particular database
- `SHOW [FULL] COLUMNS` – provides a list of the available columns in a particular table.
- `SHOW INDEX` – provides a list of all index columns for a particular table
- `SHOW CHARACTER SET` – provides information about character sets
- `SHOW COLLATION` provides information about character sets.

230

`SHOW DATABASES` lists the names of the available databases:

```
mysql> SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| menagerie      |
| mysql          |
| test           |
| world          |
+-----+
```

231

Without a `FROM` clause, `SHOW TABLES` lists the tables in the default database:

```
mysql> SHOW TABLES;
+-----+
| Tables_in_world |
+-----+
| City            |
| Country         |
| CountryLanguage |
+-----+
```



In the **SHOW TABLES** statement, the **FROM** clause is used to specify a particular database. The statement then shows a list of the tables from the specified database.

```
mysql> SHOW TABLES FROM mysql;
+-----+
| Tables_in_mysql |
+-----+
| columns_priv   |
| db             |
| func           |
| ...
| time_zone      |
| time_zone_leap_second |
| time_zone_name |
| time_zone_transition |
| time_zone_transition_type |
| user           |
+-----+
23 rows in set (#.## sec)
```

232

SHOW COLUMNS displays column structure information for the table named in the **FROM** clause:

```
mysql> SHOW COLUMNS FROM CountryLanguage;
+-----+-----+-----+-----+-----+-----+
| Field | Type    | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| CountryCode | char(3) | NO  | PRI |          |       |
| Language | char(30) | NO  | PRI |          |       |
| IsOfficial | enum('T', 'F') | NO  |      | F        |       |
| Percentage | float(4,1) | NO  |      | 0.0     |       |
+-----+-----+-----+-----+-----+-----+
```

233

SHOW COLUMNS (SHOW FIELDS) takes an optional **FULL** keyword that causes additional information to be displayed (collation, privileges, and comment) – Only the first column is displayed:

```
mysql> SHOW FULL COLUMNS FROM CountryLanguage\G
***** 1. row *****
Field: CountryCode
Type: char(3)
Collation: latin1_swedish_ci
Null: NO
Key: PRI
Default:
Extra:
Privileges: select,insert,update,references
Comment:
```



234

For some **SHOW** statements, you can give a **LIKE** clause to perform a pattern-match operation that determines which rows to display. **SHOW DATABASES**, **SHOW TABLES**, and **SHOW COLUMNS** support this feature. For example:

```
mysql> SHOW DATABASES LIKE 'm%';
+-----+
| Database (m%) |
+-----+
| menagerie      |
| mysql          |
+-----+
```

Several **SHOW** also supports the use of a **WHERE** clause. As with the **LIKE** clause, **WHERE** determines which rows to display, but **WHERE** is more flexible because you can use any kind of test, not just a pattern match:

```
mysql> SHOW COLUMNS FROM Country WHERE `Default` IS NULL;
+-----+-----+-----+-----+-----+-----+
| Field        | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| IndepYear    | smallint(6) | YES  |     | NULL    |       |
| LifeExpectancy | float(3,1) | YES  |     | NULL    |       |
| GNP          | float(10,2) | YES  |     | NULL    |       |
| GNPOld       | float(10,2) | YES  |     | NULL    |       |
| HeadOfState   | char(60)    | YES  |     | NULL    |       |
| Capital       | int(11)     | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
```

NOTE: In the preceding statement, the column name (`Default`) must be given as a quoted identifier because it is a reserved word.

235

SHOW INDEX (SHOW KEYS) displays information about the indexes (actually, indexes and also index columns) that a table has:

```
mysql> SHOW INDEX FROM City\G
***** 1. row *****
      Table: City
Non_unique: 0
      Key_name: PRIMARY
Seq_in_index: 1
Column_name: ID
      Collation: A
Cardinality: 4079
      Sub_part: NULL
      Packed: NULL
      Null:
Index_type: BTREE
      Comment:
```



236

SHOW statements are available for metadata other than for databases, tables, and columns. For example, **SHOW CHARACTER SET** displays the available character sets along with their default collations:

```
mysql> SHOW CHARACTER SET;
+-----+-----+-----+-----+
| Charset | Description          | Default collation | Maxlen |
+-----+-----+-----+-----+
| big5    | Big5 Traditional Chinese | big5_chinese_ci   |      2 |
| dec8    | DEC West European       | dec8_swedish_ci   |      1 |
| cp850   | DOS West European        | cp850_general_ci |      1 |
| hp8     | HP West European         | hp8_english_ci   |      1 |
| koi8r   | KOI8-R Relcom Russian   | koi8r_general_ci |      1 |
| latin1  | ISO 8859-1 West European | latin1_swedish_ci |      1 |
...
```

SHOW COLLATION displays the collations for each character set:

```
mysql> SHOW COLLATION;
+-----+-----+-----+-----+-----+-----+
| Collation      | Charset | Id | Default | Compiled | Sortlen |
+-----+-----+-----+-----+-----+-----+
| big5_chinese_ci | big5   |  1 | Yes     | Yes     |      1 |
| big5_bin        | big5   | 84 |          | Yes     |      1 |
| dec8_swedish_ci | dec8   |  3 | Yes     |          |      0 |
| dec8_bin        | dec8   | 69 |          |          |      0 |
| cp850_general_ci | cp850 |  4 | Yes     |          |      0 |
| cp850_bin        | cp850 | 80 |          |          |      0 |
| hp8_english_ci   | hp8    |  6 | Yes     |          |      0 |
...
```



237

8.4.2 DESCRIBE Statements

DESCRIBE, another metadata-display statement, is equivalent to **SHOW COLUMNS**. **DESCRIBE** can be abbreviated as **DESC**. So,

```
DESCRIBE table_name;
```

and

```
DESC table_name;
```

and

```
SHOW COLUMNS FROM table_name;
```

are equivalent and display the same information. However, whereas **SHOW COLUMNS** supports an optional **LIKE** and **WHERE** clause, **DESCRIBE** does not.

DESCRIBE shows the column definitions for any **INFORMATION_SCHEMA** table:

```
mysql> DESCRIBE INFORMATION_SCHEMA.CHARACTER_SETS;
+-----+-----+-----+-----+-----+-----+
| Field          | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| CHARACTER_SET_NAME | varchar(64) | NO   |   |   |   |
| DEFAULT_COLLATE_NAME | varchar(64) | NO   |   |   |   |
| DESCRIPTION      | varchar(60)  | NO   |   |   |   |
| MAXLEN           | bigint(3)   | NO   |   |   | 0  |
+-----+-----+-----+-----+-----+-----+
```





InLine Lab 8-B

In this exercise you will practice the **SHOW** and **DESCRIBE** syntax. This will require a mysql command line client and access to a MySQL server and use of the world database.

Step 1. Using SHOW DATABASES

1. Show all the available databases.

```
mysql> SHOW DATABASES;
```

A list of the available databases is returned

Step 2. Using SHOW TABLES

1. Show all available tables in the current database

```
mysql> SHOW TABLES;
```

Show all the tables in the world database:

Step 3. Using LIKE

1. Show all databases that have the letter 'o' in their name:

```
mysql> SHOW DATABASES LIKE '%o%';
```

Will list all databases fitting the criteria:

+-----+
Database (%o%)
+-----+
information_schema
world
+-----+
2 rows in set (#.# sec)

Step 4. Using SHOW FULL COLUMNS

1. Show detailed information about the columns in the City table:

```
mysql> SHOW FULL COLUMNS FROM City\G
```

Results in a vertical list of several column details.



Step 5. Using SHOW INDEX

1. Show the index information for the City table:

```
mysql> SHOW INDEX FROM City\G
```

Shows a list of Index information for City:

```
***** 1. row *****
   Table: City
Non_unique: 0
   Key_name: PRIMARY
Seq_in_index: 1
Column_name: ID
   Collation:
Cardinality: 4079
Sub_part: NULL
   Packed: NULL
      Null:Index_type: BTREE
      Comment:
1 row in set (#.## sec)
```

NOTE: The INFORMATION_SCHEMA tables TABLE_CONSTRAINTS and STATISTICS also contain index metadata.

Step 6. Using DESCRIBE

1. Show the structure of the CountryLanguage table:

```
mysql> DESCRIBE CountryLanguage;
```

Returns the columns and attributes contained by the CountryLanguage table:

Field	Type	Null	Key	Default	Extra
CountryCode	char(3)	NO	PRI		
Language	char(30)	NO	PRI		
IsOfficial	enum('T','F')	NO		F	
Percentage	float(4,1)	NO		0.0	



Step 7. Using SHOW CHARACTER SET

1. List all character sets available:

```
mysql> SHOW CHARACTER SET;
```

A full list of character sets on the current system will result.

Step 8. Using SHOW COLLATION

1. List all collations available:

```
mysql> SHOW COLLATION;
```

A full list of collations on the current system will result.



238

8.5 The mysqlshow Client Program

The `mysqlshow` client program produces information about the structure of your databases and tables. It provides a command-line interface to various forms of the `SHOW` statement that list the names of your databases, tables within a database, or information about table columns or indexes. The `mysqlshow` command has this syntax:

```
mysqlshow [options] [db_name [table_name [column_name]]]
```

The options part of the `mysqlshow` command may include any of the standard connection parameter options, such as `--host` or `--user`. You'll need to supply these options if the default connection parameters aren't appropriate. `mysqlshow` also understands options specific to its own operation. Invoke `mysqlshow` with the `--help` option to see a complete list of its options.

The action performed by `mysqlshow` depends on the number of non-option arguments you provide:

239

With no arguments, `mysqlshow` displays a result similar to that of `SHOW DATABASES`:

```
shell> mysqlshow
+-----+
|   Databases   |
+-----+
| information_schema |
| menagerie        |
| mysql            |
| test             |
| world            |
+-----+
```

With a single argument, `mysqlshow` interprets it as a database name and displays a result similar to that of `SHOW TABLES` for the database:

```
shell> mysqlshow world
Database: world
+-----+
|   Tables    |
+-----+
| City       |
| Country    |
| CountryLanguage |
+-----+
```



240

With two arguments, mysqlshow interprets them as a database and table name and displays a result similar to that of **SHOW FULL COLUMNS** for the table:

```
shell> mysqlshow world City
```

(Output omitted.)

With three arguments, the output is the same as for two arguments except that mysqlshow takes the third argument as a column name and displays **SHOW FULL COLUMNS** output only for that column:

```
shell> mysqlshow world City CountryCode
```

(Output omitted.)

When mysqlshow is used to display table structure, the --keys option may be given to display index structure as well. This information is similar to the output of **SHOW INDEX** (or **SHOW KEYS**) for the table.

If the final argument on the command line contains special characters, mysqlshow interprets the argument as a pattern and displays only the names that match the pattern. The special characters are '%' or '*' to match any sequence of characters, and '_' or '?' to match any single character. For example, the following command shows only those databases with a name that begins with 'w':

```
shell> mysqlshow -u <user_name> -p "w%"  
Enter Password: <password>  
Wildcard: w%  
+-----+  
| Databases |  
+-----+  
| world     |  
+-----+
```

NOTE: The above example demonstrates the use of user and password parameters as part of command execution.

The pattern characters might be treated as special by your command interpreter. An argument that contains any such characters should be quoted, as shown in the preceding example. Alternatively, use a character that your command interpreter doesn't treat specially. For example, '*' can be used without quoting on Windows and '%' without quoting on Unix.



InLine Lab 8-C



In this exercise you will use the methods covered in this chapter for SHOW/DESCRIBE. This will require a MySQL command line client.

Step 1. Showing databases

- From the shell prompt, list the available databases:

```
shell> mysqlshow -uroot -p  
Enter password: *****
```

Outputs all available databases managed by the MySQL server:

Databases
information_schema
mysql
test
world

Step 2. Showing tables

- Add the database name 'world' to the command line:

```
shell> mysqlshow world -uroot -p  
Enter password: *****
```

Shows list of all tables in the existing world database:

Tables
City
Country
CountryLanguage

Step 3. Showing table columns

- Add the the table name 'CountryLanguage' to the command line:

```
shell> mysqlshow world CountryLanguage -uroot -p  
Enter password: *****
```

Shows the table structure of the CountryLanguage table;



241

8.6 Chapter Summary

This chapter introduced several methods of obtaining metadata. In this chapter, you learned to:

- List the various metadata access methods available
- Recognize the structure of the `INFORMATION_SCHEMA` database
- Use the available commands to view metadata
- The differences between using `SHOW` statements and `INFORMATION_SCHEMA` queries
- Use the `mysqlshow` client program





9 DATABASES

9.1 Learning Objectives

243

This chapter introduces the use of *databases* in MySQL. In this chapter, you will learn to:

- Understand the purpose of mysql's data directory
- Employ good practices when designing a database structure
- Choose proper identifiers for databases
- Create a database
- Alter a database
- Remove a database
- Obtain database metadata



244

9.2 Database Properties

MySQL Server manages data by performing storage, retrieval, and manipulation of rows of data. Rows are contained within tables, and tables are contained within databases.

MySQL uses a directory on the file system to represent a database. Such a directory is called a *database directory*. MySQL uses a database directory to store a number of files to store information that is specific to that particular database.

For a particular MySQL server, all of its database directories have a common parent directory known as the *data directory*. Apart from being a container for all the database directories, MySQL uses the data directory to store a number of files to store information that applies to the server as a whole.

Databases and the data directory have the following features:

- A database directory has the same name as the database that it represents. For example, a database named world corresponds to a database directory named world under the data directory.
- MySQL uses the database directory to manage the components of the database such as its tables. A database may be empty or have one or more tables. A database directory may also contain files for other database objects such as triggers.
- Each database directory has a default character set and collation. You can specify these properties for a database when you create it. The properties are stored in a file named db.opt in the database directory.
- Databases cannot be nested: one database cannot contain another.
- The objects that “belong” to a database (database objects) include:
 - Table data and record of relationships
 - Stored procedures
 - Triggers
 - Views
 - Events

245

The preceding description of data directory organization indicates that MySQL Server can manage multiple databases, each of which may contain multiple tables. MySQL does not place any limits on the number of databases, although your operating system or file system might: If the file system on which the data directory resides has a limit on the number of subdirectories a directory may contain, MySQL can create no more than that number of database directories, and hence, databases.

Database or Schema?

In MySQL, the terms “database” and “schema.” are synonymous. As of MySQL 5.0, statements that use the **DATABASE** keyword can be written with **SCHEMA** instead. For example, **CREATE SCHEMA** is the same as **CREATE DATABASE**, and **SHOW SCHEMAS** is the same as **SHOW DATABASES**.



9.3 Design Practices

246

When designing a database, two important things need to be taken into account:

- Information to be stored: What (kinds of) things do we need to store information about?
- Questions to ask: Queries to be issued against the database

We must keep in mind the rules of the business that we are trying to model. In other words, the things that we need to store data about and what the relationships between those things. Along with these questions, we need to design our database in such a way that it ensures consistency and integrity and avoids structural problems such as redundancy.

You can employ various techniques with MySQL in order to practice good design:

- **Keys** - assists in identifying unique data
- **Normalization** – process of refining database tables for efficiency
- **Modeling** – graphical representation of database structure

247

9.3.1 Keys

Informally, a superkey is a set of columns within a table whose values can be used to uniquely identify a row within a table. A candidate key is a minimal set of columns necessary to identify a row. The primary key is a candidate key that has been designated (or created) as the candidate key that best defines exactly one unique row. An index is also created when a primary key is created.

Indexes

An index is a feature in a database that allows quick access to the rows in a table. Indexes will be covered in detail later in this course in the chapter on Tables.

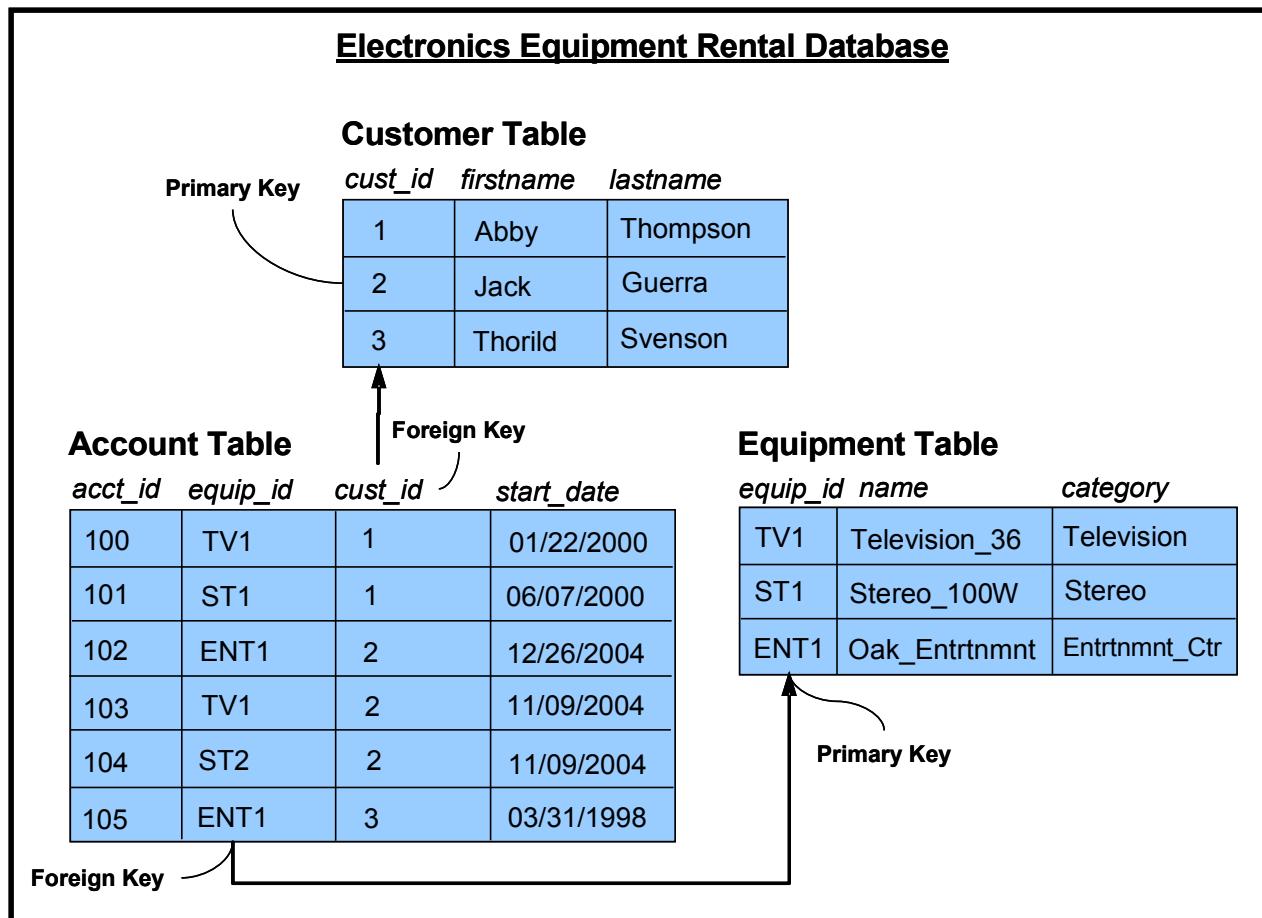
Relational databases have the capability to split data across several tables in a structured way so that data is rarely repeated. The primary key in one table can be referenced by a key in another table, thereby connecting the two tables. The referencing key is called a foreign key.

Benefits to Using Keys

- Decrease lookup time
- Enforce uniqueness identification of each row
- A primary key cannot contain a NULL (unknown or empty value)



248



Primary Key

A primary key can be some natural value or a derived value (i.e., *equip_id*+*Television*). A primary can also be an "unnatural" value (i.e., *cust_id*).

Key Constraints

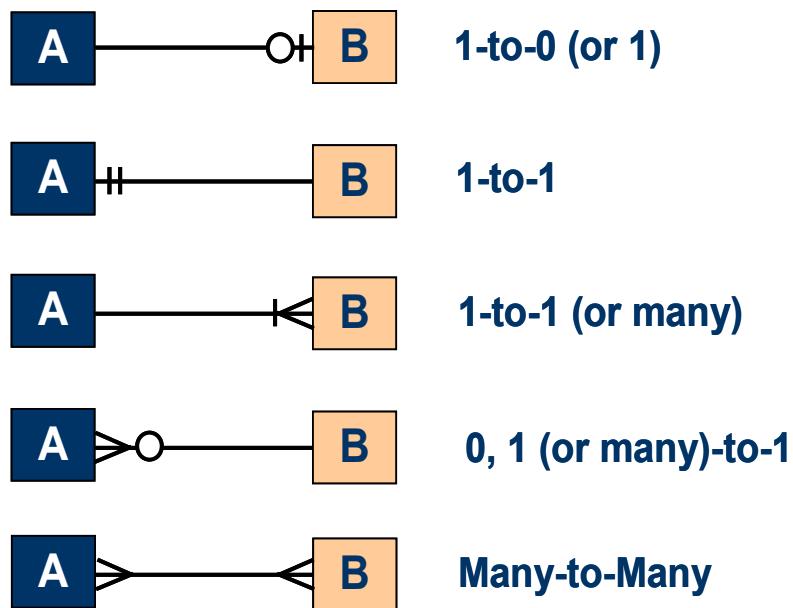
A key constraint is a restriction placed on one or more columns of a table. Without constraints, a database may not be consistent. Table constraints are covered in detail in the Tables chapter of this training.



249 9.3.2 Common Diagramming Systems**ERD**

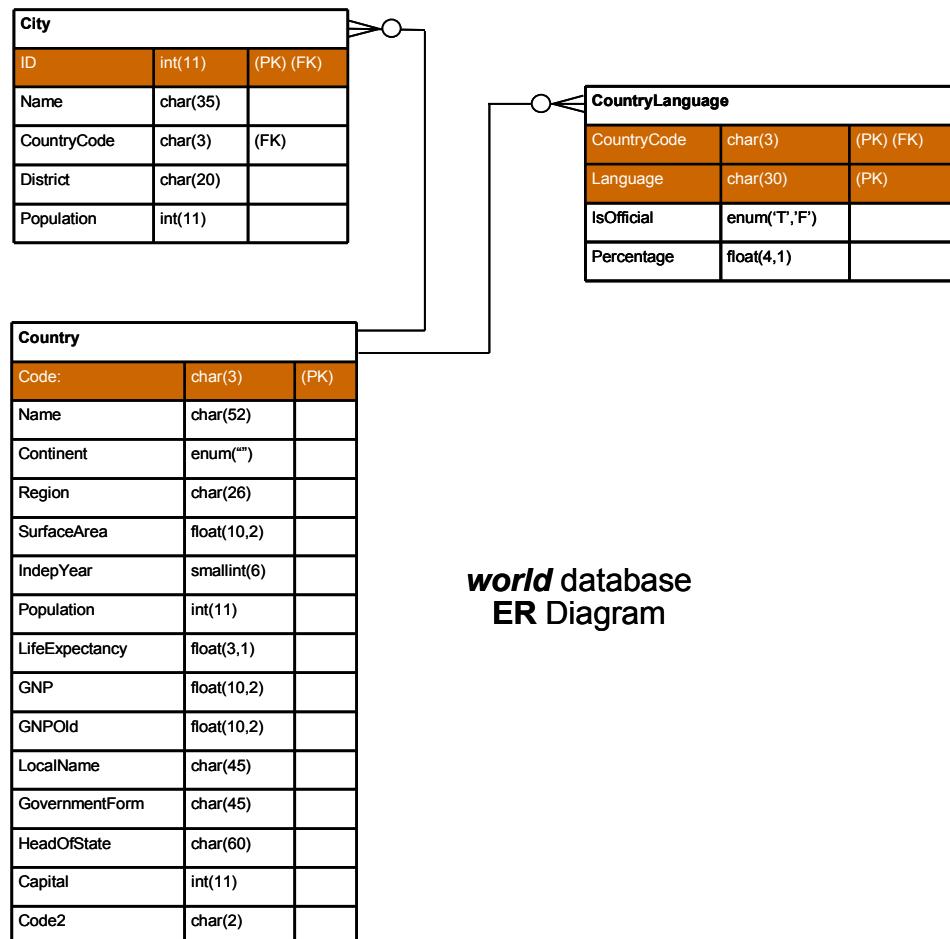
The entity-relationship model or entity-relationship diagram (ERD) is a more common data model or diagram for high-level descriptions of conceptual data models, and it provides a graphical notation for representing such data models in the form of entity-relationship diagrams. Such models are typically used in the first stage of information-system design; they are used, for example, to describe information needs and/or the type of information that is to be stored in the database during the requirements analysis. The data modeling technique, however, can be used to describe any Ontology (i.e. an overview and classifications of used terms and their relationships) for a certain universe of discourse (i.e. area of interest). In the case of the design of an information system that is based on a database, the conceptual data model is, at a later stage (usually called logical design), mapped to a logical data model, such as the relational model; this in turn is mapped to a physical model during physical design.

There are a number of conventions for entity-relationship diagrams (ERDs). There are a range of notations more typically employed in logical and physical database design, including information engineering, IDEF1x and dimensional modeling. Some basic ERD relationship notation is shown below:



250

The example ER diagram below is for the world database which will be used for examples and exercises during this course.



9.3.3 Normalization

251

Normalization is a technique that is widely used as a guide in designing relational databases. It is the process of refining a database design to ensure that each independent piece of information is in only one place (except for foreign keys). Normalizing your tables removes redundant data, makes it possible to access data more flexibly, and eliminates the possibility that inappropriate modifications will take place that make the data inconsistent. Normalization of a complex table often amounts to taking it through a process of decomposition into a set of smaller tables. This process removes redundant groups within rows and then duplicate data within columns.

Why Normalize?

- **Eliminate Redundant Data** – Removing entries of redundant data is vital to the success of normalization and ensures integrity by keeping individual data in one location making it easier to update and manage.
- **Eliminate Columns Not Dependent On Key** – Columns that are not dependent on a key can get disconnected from data and cause data corruption. By having all data connected to a key identifier, data integrity is ensured.
- **Isolate Independent Multiple Relationships** – With relationships of data across multiple tables, normalization provides a cleaner recognition of these relationships and ensures that all relationships are identified and applied.

252

Advantages of Normalizing

Data normalization can be an expensive process in time and resources; however, besides the advantages derived from the purposes of normalization, there are other advantages that need to be taken into account:

- **ER Diagrams** – Developing ERD's are much easier when the data is normalized and the relationships are easy to identify. (More about ERD's later in the course.)
- **Compact** – By having the data normalized it is easier to modify a single object property. If an attribute of a table had student genders labeled as boy or girl and this was identified as incorrect labeling of students, it would be easy to change boy to male and girl to female when there is a normalized table called gender.
- **Joins** – Joins can be very expensive. However, proper normalization can improve relationships between tables and minimize the amount of data having to be searched.
- **Optimizer** – Due to the better joins, the optimizer has fewer indexes per table, so it doesn't have to consider as many execution plans. This helps with retrievals and updates.
- **Updates** - Easier to update in one location versus multiple locations.

253

Disadvantages of Normalizing

Even with the numerous advantages associated with normalization, there are some disadvantages that have to be addressed:

- **Numerous Tables** - Multiple tables must be accessed for user reports and other user data.
- **Maintenance** - Maintenance may be in some conflict with your business processes. You have to make sure that you do not make changes that go against your organizational standards.
- **Slower Read Requests** – with Joins



254

Example of *world* Database Normalization

Viewing the ERD diagram (above) of the world database, you can see the columns being used for each table. The following demonstrates a portion of the normalization process:

- Notice that there is a table called CountryLanguage. Why are the languages not included in the Country table? As you can see below there can be a big difference in the number of languages spoken in various countries; (Turkey...)

```
+-----+-----+
| CountryCode | Language |
+-----+-----+
| TUR         | Arabic    |
| TUR         | Kurdish   |
| TUR         | Turkish   |
+-----+-----+
3 rows in set (#.# sec)
```

(versus Canada...)

```
+-----+-----+
| CountryCode | Language      |
+-----+-----+
| CAN         | Chinese       |
| CAN         | Dutch         |
| CAN         | English       |
| CAN         | Eskimo Languages |
| CAN         | French        |
| CAN         | German        |
| CAN         | Italian       |
| CAN         | Polish        |
| CAN         | Portuguese   |
| CAN         | Punjabi       |
| CAN         | Spanish       |
| CAN         | Ukrainian    |
+-----+-----+
12 rows in set (#.# sec)
```



255

- It would be inefficient to add more columns to the Country table structure to allow for the extra languages for some countries, while others will not need as many (or any at all, in the case of Antarctica). Therefore, there would be many empty spaces in the table.
- Also, there are details which pertain only to the languages themselves which would be extraneous to the other country information (e.g., official country language, percentage of population speaking each language).

CountryCode	Language	IsOfficial	Percentage
TUR	Arabic	F	1.4
TUR	Kurdish	F	10.6
TUR	Turkish	T	87.6

3 rows in set (#.# sec)

- The goal of normalization is met by creating this table, since the information that it contains is in only one place and connected to the other tables via primary keys.

256

Normal Forms

Normalization has many levels with each successive level providing stronger guarantees about the data modification anomalies being eliminated. Each level, or form, depends on the previous level and it is best practice to work through each form individually to achieve the best results. There are numerous levels to normalization; however, the first three levels are the most common and are explained in detail below:

- First Normal Form (1NF)** – A table that is in the first level of normalization contains no repeating groups within rows.
- Second Normal Form (2NF)** – A table that is in this level of normalization has been normalized at the first level (1NF) and every non-key (supporting) value is dependent on the primary key value. In addition, the latter constraint means that a non-key value must depend on every column in a composite primary key.
- Third Normal Form (3NF)** – A table that is in this level of normalization has been normalized so that every non-key (supporting) value depends solely on the primary key and not on some other non-key (supporting) value.

257

1 to Many Relationships

One of the most positive results of normalizing a database is the resulting relationships between tables. It becomes several *1 to Many* relationships. This relationship ensures an efficient approach to updates, deletes and alterations to existing data. The world database has examples of this relationship:

- 1* Country to *many* Languages
- 1* Country to *many* Cities



Instructor Note: Further discussion of normalization using the world database... Show the database structures, then ask questions...Why do we have three tables instead of just one? Why not just two? Are there ever any repeats? Are there ever any needless blank entries?

Specifically, show the City table and ask/demo these questions: Could we store the name of the country? Do country names change? What happens then? Do we have two places to change data? What about when Rhodesia becomes Zimbabwe? What happens then? What if we had a numeric key for the countries? The key could stay the same, the name and the code could change in ONE PLACE.



Quiz 9-A

In this exercise you will answer questions pertaining to Normalization.

1. Take a look at the tables in the world database using **DESCRIBE**. Based upon your examination of the tables, what would you say is the purpose of the world database?

2. Explain the reasons for the choice of tables, entities and identifiers. How was normalization used?

3. Explain the relationships between the entities in the tables which will have a relationship to each other.

4. List some examples within the world database where normalization was not used.



258

9.4 Identifiers

An identifier is simply the name of an alias, a database, a table, a column or an index. When you write SQL statements, you use names to refer to databases and objects contained in databases such as tables, stored routines, functions and triggers. Some of these objects have components with their own names. For example, tables have columns and indexes. It's also possible to create aliases, which act as synonyms for table and column names.

9.4.1 Identifier Syntax

Identifiers may be unquoted or quoted. If unquoted, an identifier must follow these rules:

- An identifier may contain all alphanumeric characters, the underline character ('_'), and the dollar sign ('\$').
- An identifier may begin with any of the legal characters, even a digit. However, it's best to avoid identifiers that might be misinterpreted as constants.
- An identifier cannot consist entirely of digits.
- An identifier may be quoted, in which case it can contain characters such as spaces or dashes that aren't otherwise legal. To quote an identifier, you may enclose it within backtick (`) characters. If the **ANSI_QUOTES** SQL mode is enabled, you may also quote an identifier by enclosing it within double quotes ("").

An alias identifier can include any character, but should be quoted if it's a reserved word (such as **SELECT** or **DESC**), contains special characters, or consists entirely of digits. Aliases may be quoted within single quotes (''), double quotes (""), or backticks.

Quote it!

If you aren't sure whether an identifier is legal, quote it. It's harmless to put quotes around an identifier that's legal without them.

259

9.4.2 Reserved Words as Identifiers

Reserved words are special. For example, function names cannot be used as identifiers such as table or column names, and an error occurs if you try to do so. The following statement fails because it attempts to create a column named **order**, which is erroneous because **order** is a reserved word (it's used in **ORDER BY** clauses):

```
mysql> CREATE TABLE t (order INT NOT NULL UNIQUE, d DATE NOT NULL);
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 'order INT NOT NULL UNIQUE, d DATE NOT NULL)' at line 1
```



Similarly, the following statement fails because it uses a reserved word as an alias:

```
mysql> SELECT 1 AS INTEGER;
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 'INTEGER' at line 1
```

260

The solution to these problems is to quote the identifiers properly. The rules depend on the type of identifier you're quoting:

- To use a reserved word as a database, table, column, or index identifier, there are either one or two allowable quoting styles, depending on the server SQL mode. By default, quoting a reserved word within backtick (`) characters allows it to be used as an identifier:

```
mysql> CREATE TABLE t (`order` INT NOT NULL UNIQUE,d DATE NOT NULL);
Query OK, 0 rows affected (#.# sec)
```

If the [ANSI_QUOTES](#) SQL mode is enabled, it's also allowable to quote using double quotes:

```
mysql> CREATE TABLE t ("order" INT NOT NULL UNIQUE,d DATE NOT NULL);
Query OK, 0 rows affected (#.# sec)
```

If an identifier must be quoted in a [CREATE TABLE](#) statement, it's also necessary to quote it in any subsequent statements that refer to the identifier.

- To use a reserved word as an alias, quote it using either single quotes, double quotes, or backticks. The SQL mode makes no difference; it's legal to use any of the three quoting characters regardless. Thus, to use [INTEGER](#) as an alias, you can write it any of these ways:

```
mysql> SELECT 1 AS 'INTEGER';
mysql> SELECT 1 AS "INTEGER";
mysql> SELECT 1 AS `INTEGER`;
```

It's a good idea to avoid using function names as identifiers. Normally, they aren't reserved, but there are circumstances under which this isn't true.

261

9.4.3 Using Qualified Names

Column and table identifiers can be written in qualified form—that is, together with the identifier of a higher-level element, with a period (‘.’) separator. Sometimes qualifiers are necessary to resolve ambiguity. Other times you may elect to use them simply to make a statement clearer or more precise.



A table name may be qualified with the name of the database to which it belongs. For example, the Country table in the world database may be referred to as world.Country, where a ‘.’ character is placed between the two identifiers in the name. If world is the default database, these statements are equivalent:

```
mysql> SELECT * FROM Country;
mysql> SELECT * FROM world.Country;
```

A column name may be qualified with the name of the table to which it belongs. For example, the Name column in the Country table may be referred to as Country.Name.

A further level of column qualification is possible because a table name may be qualified with a database name. So, another way to refer to the Name column is world.Country.Name. If world is the default database, the following statements are equivalent. They differ only in having successively more specific levels of name qualification:

```
mysql> SELECT Name FROM Country;
mysql> SELECT Country.Name FROM Country;
mysql> SELECT world.Country.Name FROM world.Country;
```

262

9.4.4 Case Sensitivity

A property that affects how you use identifiers is whether they're case sensitive; some identifiers are case sensitive and others are not. You should understand which is which and use them accordingly.

The rules that determine whether an identifier is case sensitive depend on what kind of identifier it is:

- For database and table identifiers, case sensitivity depends on the operating system and filesystem of the server host, and on the setting of the lower_case_table_names system variable.
- Databases and tables are represented by directories and files, so if the operating system has case-sensitive filenames, MySQL treats database and table identifiers as case sensitive. If filenames aren't case sensitive, these identifiers are not either.
- Regardless of the case-sensitive properties of your filesystem, database and table identifiers must be written consistently with the same lettercase throughout a given statement.
- Column, index, and stored routine identifiers are not case sensitive.
- Column aliases are not case sensitive.
- The case-sensitivity of trigger identifiers is dependent upon the operating system (like it is the case for table identifiers). However, that is unaffected by any setting of the lower_case_table_names system variable. So, on Windows and OS/X trigger names are not case-sensitive whereas they are case-sensitive on *nix.



263

9.5 Creating Databases

To create a new database, use the **CREATE DATABASE (SCHEMA)** statement. The following statement creates a database named mydb:

```
mysql> CREATE DATABASE mydb;
```

If you try to create a database that already exists, an error occurs. If you simply want to ensure that the database exists, add an **IF NOT EXISTS** clause to the statement:

```
mysql> CREATE DATABASE IF NOT EXISTS mydb;
```

With the additional clause, the statement creates the database only if it does not already exist. Otherwise, the statement does nothing and no error occurs. This can be useful in applications that need to ensure that a given database is available, without disrupting any existing database with the same name.

The **CREATE DATABASE** statement has two optional clauses, **CHARACTER SET** and **COLLATE**, that assign a default character set and collation for the database. If given, they appear at the end of the statement following the database name. The following statement specifies that the mydb database has a default character set of utf8 and collation of utf8_danish_ci:

```
mysql> CREATE DATABASE mydb CHARACTER SET utf8 COLLATE utf8_danish_ci;
```

The default character set and collation for the database are used as the defaults for tables created in the database for which no explicit character set or collation of their own are specified. The database defaults are stored in the db.opt file in the database directory.

Note: **CHARACTER SET** is really a column setting. By giving the table or the database a character set, it just affects the columns upon creation.

264

Creating a database has no effect on which database currently is selected as the default database. To make the new database the default database, issue a **USE** statement:

```
mysql> USE mydb;
```

After a database has been created, you can populate it with objects such as tables or stored routines. The **CREATE** statements for these objects are discussed in later chapters.

SHOW CREATE DATABASE shows the **CREATE DATABASE** statement that creates a database:

```
mysql> SHOW CREATE DATABASE world\G
***** 1. row *****
      Database: world
Create Database: CREATE DATABASE `world`
                  /*!40100 DEFAULT CHARACTER SET latin1 */
```





InLine Lab 9-B

In this exercise you will use the **CREATE DATABASE** statement. This will require a mysql command line client and access to a MySQL server.

Step 1. Create a new database

1. Create a new database called 'db_test':

```
mysql> CREATE DATABASE db_test;  
Returns Query OK, 0 rows affected (0.05 sec) when complete.
```

Step 2. List available databases

1. Show the list of current databases:

```
mysql> SHOW DATABASES;
```

Returns a list of databases. Note that db_test is on the list.

Step 3. Switch to the newly created database

1. Change the current database to the newly created one:

```
mysql> USE db_test;
```

Sets the current database to be use to the db_test database.

Step 4. Display the details of the database structure

1. Show the structure of the creation of db_test database:

```
mysql> SHOW CREATE DATABASE db_test\G
```

Displays the entire **CREATE DATABASE** statement that is used by the mysql client;

```
***** 1. row *****  
Database: db_test  
Create Database: CREATE DATABASE `db_test` /*!40100 DEFAULT CHARACTER SET  
latin1 */  
1 row in set (#.## sec)
```

Note: ALL the above lab statements can also be written with **SCHEMA** in place of **DATABASE**.



265

9.6 Altering Databases

The **ALTER DATABASE** statement changes options for an existing database. The allowable options are the same as for **CREATE DATABASE**; that is, **CHARACTER SET** and **COLLATE**. The following statement changes the default collation of the mydb database to utf8_polish_ci:

```
mysql> ALTER DATABASE mydb COLLATE utf8_polish_ci;
```

This statement changes both the default character set and collation:

```
mysql> ALTER DATABASE mydb CHARACTER SET latin1 COLLATE latin1_swedish_ci;
```

The database name is optional for **ALTER DATABASE**. If no database is named, the statement changes the options for the default database. This requires that there be a currently selected database. Otherwise, an error occurs.

You cannot use **ALTER DATABASE** to rename a database. One way to accomplish this is to dump the database, create a database with the new name, reload the data into the new database, and drop the old database.

Note: The clause **IF EXISTS** is not available with **ALTER DATABASE**.





InLine Lab 9-C

In this exercise you will use the **ALTER DATABASE** statement to change the character set and collation for the db_test database created in the last lab. This will require a mysql command line client and access to a MySQL server.

Step 1. Review available character sets

1. Display a list of available character sets:

```
mysql> SHOW CHARACTER SET;
```

Returns a full list of the available character sets.

Step 2. Review available collations

1. Display a list of all available collations.

```
mysql> SHOW COLLATION;
```

Returns a full list of the available collations.

Step 3. Make modifications to an existing database

1. Use the **ALTER DATABASE** command to change the character set of the db_test schema to the Kamenicky Czech-Slovak set, and the collation to the latin2 czech character set:

```
mysql> ALTER DATABASE db_test CHARACTER SET keybcs2  
      -> COLLATE latin2_czech_cs;
```

Results in an error message stating that this collation is not valid for this character set;

```
ERROR 1253 (42000): COLLATION 'latin2_czech_cs' is not valid for CHARACTER  
SET 'keybcs2'
```

2. Notice that the table of character sets includes a corresponding Default Collation for each character set. Use the **ALTER DATABASE** command to change the collation to the proper default:

```
mysql> ALTER DATABASE db_test CHARACTER SET keybcs2  
      -> COLLATE keybcs2_general_ci;
```

Completes the alteration with an **OK** response.



Step 4. Display the details of the database structure

1. Show the structure of the creation of db_test database:

```
mysql> SHOW CREATE DATABASE db_test\G
```

Displays the entire **CREATE DATABASE** statement that is used by the mysql client;

```
***** 1. row *****
Database: db_test
Create Database: CREATE DATABASE `db_test` /*!40100 DEFAULT CHARACTER SET
keybcs2 */
1 row in set (#.## sec)
```



266

9.7 Dropping Databases

When you no longer need a database, you can remove it with `DROP DATABASE`:

```
mysql> DROP DATABASE mydb;
```

It is an error if the database does not exist. To cause a warning instead, include an `IF EXISTS` clause:

```
mysql> DROP DATABASE IF EXISTS mydb;
```

Any warning generated when `IF EXISTS` is used can be displayed with `SHOW WARNINGS`.

`DROP DATABASE` does not require the database to be empty. When dropping the database, MySQL removes any objects that it contains, such as tables, stored routines, and triggers.

A successful `DROP DATABASE` returns a row count that indicates the number of tables dropped. (This actually is the number of .frm files removed, which amounts to the same thing.) You can check your current databases to make sure that it is gone by using the `SHOW DATABASES` command.

9.7.1 CAUTION: When Using `DROP DATABASE`

A database is represented by a directory under the data directory. The server deletes only files and directories that it can identify as having been created by itself (for example, .frm files or RAID directories). It does not delete other files and directories. If you have put non-table files in that directory, those files are not deleted by the `DROP DATABASE` command. This results in failure to remove the database directory and `DROP DATABASE` fails. In that case, the database will continue to be listed by `SHOW DATABASES`. To correct this problem, you can manually remove the database directory and any files within it.





InLine Lab 9-D

In this exercise you will use the **DROP** statement. This will require a mysql command line client and access to a MySQL server.

Step 1. Display current database

1. Show the list of current databases:

```
mysql> SHOW DATABASES;
```

Returns a list of databases. Note that 'db_test' is on the list.

Step 2. Delete the newly created database

1. Issue a **DROP DATABASE** statement which will delete the entire 'db_test' database:

```
mysql> DROP DATABASE db_test;
```

Will return the following when the action is complete:

```
Query OK, 1 row affected (#.## sec)
```

Step 3. Display current databases

1. Show the list of current databases:

```
mysql> SHOW DATABASES;
```

Returns a list of databases. Note that 'db_test' is no longer on the list.

Note: ALL the above lab statements can also be written with **SCHEMA** in place of **DATABASE**.





Further Practice

In this exercise, you will use the Database information covered in this chapter to normalize the world database and build new tables to reflect that normalization.

1. The world database can be normalized in many different ways. Attempt to come up with a better normalization scheme than is currently used. Using the various diagramming methods available, document your new world database design.

267

Note: There is no *right* or *wrong* answer for this exercise. Discuss any questions you may have as you go through the process with the instructor.



9.8 Chapter Summary

268

This chapter introduced the use of Databases in MySQL. In this chapter, you learned to:

- Understand the database properties of a data directory
- Employ good practices when designing a database structure
- Utilize proper identifiers for databases
- Create a database
- Alter a database
- Remove a database
- Obtain database metadata





10 TABLES

10.1 Learning Objectives

270

This chapter introduces the use of database tables in MySQL. In this chapter, you will learn to:

- Assign appropriate table properties
- Assign appropriate column options
- Create a table
- Alter a table
- Empty a table
- Remove a table
- Understand and use indexes accurately
- Assign and use foreign keys
- Obtain table and index metadata



271

10.2 Creating Tables

After the database structure is designed and the database has been created, you can add the individual tables. Using appropriate data types and the options, the tables from the design plan can be added to the database.

The syntax for creating a table is shown below, including the various column and table options;

```
CREATE TABLE <table> (
    <column name> <column type> [<column options>],
    <column name> <column type> [<column options>], ...,
    [<list of table constraints and or indexes>
)
[<table options>]
```

NOTE: A table always has a table name, and always has at least one column. Other items are optional.

Example:

```
mysql> CREATE TABLE CountryLanguage (
    ->     CountryCode CHAR(3) NOT NULL,
    ->     Language CHAR(30) NOT NULL,
    ->     IsOfficial ENUM('True', 'False') NOT NULL DEFAULT 'False',
    ->     Percentage FLOAT(3,1) NOT NULL,
    ->     PRIMARY KEY(CountryCode, Language)
    -> )
    -> ENGINE = MyISAM
    -> COMMENT = 'Lists Language Spoken';
```

We will now discuss each individual line this **CREATE TABLE** statement:

1. **CREATE TABLE** **CountryLanguage** - Here you create a table called **CountryLanguage**. The table name may or may not be case sensitive depending on the operating system (and the setting of the `lower_case_table_names` option). It is generally a good idea to

- Choose table names so that they are unique within a database even when ignoring letter case differences
- Always stick to the exact same spelling for the name of a particular table

Adhering to these two principles ensures that your database design works as expected regardless of the operating system and setting of the `lower_case_table_names` option.

After the table name, a definition of the structure of the table (that is, its constituent columns, table constraints and indexes) appears between parenthesis (the part between the left parenthesis immediately after the table name and the right parenthesis appearing immediately before the **ENGINE** keyword.)

2. **CountryCode** - Column in the table with a **CHAR** data type with a length of 3 characters. Specifying **NOT NULL** ensures the **NULL** value can not be stored in this column. After the column definition, a comma appears, indicating that the definition of the table structure is continued.



3. **Language** - another column of the **CHAR** a data type having a maximum length of 30 characters, that also does not allow **NULL** values.
4. **IsOfficial** – a not-nullable column of the **ENUM** data type that can only hold the values 'True' or 'False'. The **DEFAULT** clause specifies that the value 'False' is to be used when no value is supplied for the column.
5. **Percentage**—a not-nullable column of the **FLOAT** data type which can store values of 3 digits including 1 digit to the right of the decimal point.
6. **PRIMARY KEY** - Defines a primary key constraint which will ensure that any combination of values in the columns **CountryCode** and **Language** can appear at most once in the entire table. In other words, this enforces uniqueness over the **CountryCode** and **Language** columns, and guarantees that any row in the table is uniquely identified by these columns.
7. After the table structure definition, the **ENGINE** option defines the storage engine used by this table to store and retrieve data. In this case, the MyISAM engine is used.
8. The **COMMENT** option is used to store some human readable information about the purpose of this table.

272

10.2.1 Table Properties

Options can be added to the **CREATE TABLE** statement to define the overall behavior of the table. Table options may appear after the closing right parenthesis of the table structure definition. The following syntax is used to define them:

```
<option name> [=] <option value>
```

So, an option is defined by its name, followed by its value. Optionally, an equals sign may appear between the name and the value. The format and range of the value is dependent upon the specific option.

A table can have multiple options. When specifying multiple options, the order in which the options are listed is not important, but each option may appear exactly once. If multiple options are used, then they should be separated by whitespace, not a comma which is usually used as separator.

We will now discuss a number of common table options.

- **ENGINE** [=] <engine name>

Indicates that the storage engine with the name <engine name> is to be used for the table.

MySQL manages tables using storage engines, each of which handles tables that have a given set of characteristics. Different storage engines have differing performance characteristics, and you can choose which engine most closely matches the characteristics that you need. For example, you might require transactional capabilities and guaranteed data integrity even if a crash occurs, or you might want a very fast lookup table stored in memory for which the contents can be lost in a crash and reloaded at the next server startup. With MySQL, you can make this choice on a per-table basis. Any given table is managed by a particular storage engine.

A number of popular engines are: MyISAM, InnoDB, MEMORY, MERGE

MyISAM is the default storage engine (unless --default-storage-engine has been set)



- **COMMENT** [=] '<comment>'

Up to 60 characters of free form text for documentation purposes. Note that the comment text itself is assigned as a character string, and must thus be enclosed in single quotes (or double quotes if the sql mode does not include the `ANSI_QUOTES` value).

- **[DEFAULT] [CHARACTER SET | CHARSET]** [=] <character set name>

Specifies the default character set for the table. Any column with a characters string data type that does not have its own explicit **CHARACTER SET** option would use this default.

- **[DEFAULT] COLLATE** = <collation name>

Specifies the default collation for the table. Any column that does not have its own explicit **COLLATE** option would use this default.

Table options example:

```
CREATE TABLE CountryLanguage (
...<table structure definitions>
)
ENGINE MyISAM
COMMENT 'Lists Language Spoken'
CHARSET utf8
COLLATE utf8_unicode_ci;
```

273

10.2.2 Column Options

A table must have at least one column. Each column has to have a name and a data type. In addition, a column definition can have several options. These options offer an additional specification for the behavior of the column.

We will now discuss a number of common column options:

- **NULL** and **NOT NULL**: Specify the nullability for the column. A nullable column allows **NULL** values to be stored, and a not nullable column disallows **NULL** values to be stored. A nullable column can be explicitly defined using the **NULL** keyword. A not nullable column is explicitly defined by using **NOT NULL**. When omitting an explicit nullability specification the column will be nullable by default.

NOTE: Note that a **PRIMARY KEY** definition automatically makes all columns in the primary key not nullable. This is true even if the columns were explicitly defined as nullable.

Column nullability should be defined according to business and/or application requirements. For example, a Purchase Order table will most likely have a mandatory order date. In this case it would make sense to create a **NOT NULL** column to store the order date. This will prevent invalid Purchase Orders that have a missing order date from being entered into the system.

Likewise, columns are also defined to be nullable according to business need. For example, a Person table usually has a not nullable column for the last name, but may have a nullable column for storing an email address. This makes sense as there is always a possibility that a person simply does not have an email address. In many cases a stored **NULL** value is used to record that the data is either not applicable or not yet available.



In some cases, tables may be designed to only contain **NOT NULL** columns. For the columns where a value may not be applicable or where data may be missing, a special value is then chosen to record the fact that data is missing or not yet available. There are several reasons for avoiding nullable columns. In MySQL, tables that have only **NOT NULL** columns are a bit smaller, and often query performance is better when there are only not nullable columns. Another reason to avoid nullable columns is that particular queries becomes more complex when taking **NULL** values into account.

Avoiding nullable columns is a very common practice in data warehousing environments.

- **DEFAULT** – specifies an explicit default value for the column. The default value is used when creating a new row. The default value is used for all columns for which no value is supplied. When no explicit **DEFAULT** clause is present, the actual default value for the column is dependent upon the nullability of the column. For nullable columns, the implicit default is **NULL**. For not nullable columns the default value is dependent upon the data type. For example, the default value for integer columns is zero (0) and the default value for string columns is the empty string ('').

The column default value also plays a role for those columns that are part of a foreign key constraint that has a **SET DEFAULT** update rule or delete rule. In those cases, an update or delete of the parent row resets the foreign key columns to their default value.

- **AUTO_INCREMENT** – This option specifies that an integer value is to be automatically supplied in case a value is not explicitly supplied at **INSERT** time, or when a **NULL** value is supplied. The **AUTO_INCREMENT** option is used almost exclusively to implement a technical primary key.

This option is available only for columns that have an integer data type. In addition, the column must be part of an index (or **UNIQUE** or **PRIMARY KEY** constraint). For MyISAM tables, the column may appear in a composite index. For all other storage engines, the column must be the only column in the entire index.

If the **AUTO_INCREMENT** column is the only column in the index, the values are generated by taking the maximum value and incrementing that by a particular amount. The values that are thus generated form a unique sequence of integers which is usually used as a technical primary key. By default the increment is one. The increment and offset maybe controlled using the `auto_increment_increment` and `auto_increment_offset` server variables respectively.

For MyISAM table, an **AUTO_INCREMENT** column may be in a composite index. In this case, a sequence of incrementing values is generated for each unique combination of values in the columns that precede the **AUTO_INCREMENT** column in the index.

NOTE: **AUTO_INCREMENT** may also be used as a table option. In this particular context, **AUTO_INCREMENT** specifies the offset for the sequence for this particular table.

There can only be one **AUTO_INCREMENT** column per table.



10.2.3 Constraints

A constraint is a rule that defines a particular restriction on the rows that are stored in the table. Constraints prevent the contents of a table from being changed unless the change falls within the restrictions set out by the rule.

Constraints serve to protect the consistency of the database. For example, if the MySQL server allowed you to modify the **world** database by changing the **CountryCode** in the **CountryLanguage** table without changing the corresponding **CountryCode** in the **City** table, then you would end up with accounts that no longer point to valid rows in the **Country** table (known as orphaned rows). Constraints may be used to prevent such changes from happening in the first place by only allowing changes if they leave the database in a consistent state.

Types of Constraints:

- **PRIMARY KEY** – Specifies that the values for a particular set of columns cannot be **NULL** and that the combination of those values must be unique throughout the entire table. By defining a primary key it becomes possible to uniquely identify each row in the table.
Any given table can at most have one **PRIMARY KEY**.
- **UNIQUE** - Specifies that the combination of values for a particular set of columns must be unique throughout the entire table. This is like a primary key, except that the columns of a **UNIQUE** constraint may be nullable. This means that a **UNIQUE** constraint will guard the uniqueness for the columns upon which it is defined, but only if there is a **NOT NULL** value present for each of the columns that make up the **UNIQUE** constraint.
- **FOREIGN KEY** – Specifies that for each row, the combination of values for a particular set of columns must be present in a row in another set of columns (in either the same or another table in some database managed by the server).

Foreign keys are used to enforce the referential integrity of the database. The concept of referential integrity entails that the rows in a table can maintain a valid reference to some other row (often in another table but usually in the same database). The reference itself is implemented by having the same combination of values in the related rows.

The table with the **FOREIGN KEY** constraint is the referencing table and the other table is the referenced table. In most cases, the columns in the referenced table together form either a **PRIMARY KEY** constraint or a **UNIQUE** constraint. That is why it is called a foreign key: the values in the referencing table are a key in the referenced table. So the referencing table contains a key from another (foreign) table, and hence it is called a foreign key.

Foreign keys ensure that a row that is being referenced cannot suddenly be discarded or changed without ensuring that the rows that reference it are processed accordingly. Foreign keys offer a number of actions to control this behavior. For example, a foreign key can be used to ensure that a row in the **Country** table cannot be deleted before there are no more rows in the **City** table that reference that particular row from the **Country** table. Alternatively, the foreign key can be created so that it automatically removes or changes the referencing rows.

The MySQL server uses an index to enforce primary-key, foreign-key, and unique constraints. Indexes are addressed in a later chapter.



274

Column options example:

```
mysql> CREATE TABLE City (
->     ID          INT      NOT NULL AUTO_INCREMENT,
->     Name        CHAR(35) NOT NULL                      DEFAULT '',
->     CountryCode CHAR(3)  NOT NULL                      DEFAULT '',
->     District    CHAR(20) NOT NULL                     DEFAULT '',
->     Population  INT      NOT NULL                     DEFAULT 0,
->     PRIMARY KEY (ID),
->     FOREIGN KEY (CountryCode) REFERENCES Country(Code)
-> ) ENGINE=MyISAM CHARSET=latin1;
```

275

10.2.4 SHOW CREATE TABLE

The **SHOW CREATE TABLE** statement can be used to obtain the text that makes up a **CREATE TABLE** statement that describes the structure of an existing table. This can be helpful in understanding the structure of a table. It can also be used as a basis for the syntax to create a new table. For example, to obtain the **CREATE TABLE** statement for the **City** table from the **world** database, use the following statement:

```
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=MyISAM DEFAULT CHARSET=latin1
1 row in set (#.## sec)
```





InLine Lab 10-A

In this exercise you will use the **SHOW CREATE TABLE** and **CREATE TABLE** command statement to view the creation of a **world** table and create a new table. This will require a mysql command line client and access to a MySQL server.

Step 1. Set the default database

1. From the mysql prompt type;

```
mysql> USE world;
```

Sets the **world** database as the default database.

Step 2. Using **SHOW CREATE TABLE**

1. Obtain the **CREATE TABLE** statement for the **City** table in the **world** database:

```
mysql> SHOW CREATE TABLE City\G
```

A **CREATE TABLE** statement for the **City** table is returned:

```
***** 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=MyISAM AUTO_INCREMENT=4080 DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

Step 3. Use the **test** database

1. At the mysql prompt type:

```
mysql> USE test;
```

Sets the **test** database as the default database.



Step 4. Using the `CREATE TABLE` syntax

1. Use a single statement to create a table called `db`. The table should have one not nullable column called 'name', having 10 character long fixed length string type. Specify InnoDB as the table's storage engine, set the table's character set to be latin2, and add a table comment that reads 'Create Table lab':

```
mysql> CREATE TABLE db (
->     name CHAR(10) NOT NULL
-> )
-> ENGINE = InnoDB
-> CHARSET = latin2
-> COMMENT = 'Create Table lab'
```

Creates the table and returns an OK message.

Step 5. Using `SHOW CREATE TABLE`

1. Obtain the `CREATE TABLE` statement for the `db` table in the `test` database.

```
mysql> SHOW CREATE TABLE db\G
```

A representation of the `db` table creation statement is shown. Confirm that the table properties are as originally specified.

```
***** 1. row *****
Table: db
Create Table: CREATE TABLE `db` (
  `name` char(10) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin2 COMMENT='Create Table lab'
1 row in set (#.## sec)
```

Step 6. Using `AUTO_INCREMENT`

1. Create a table called `auto_text` with an auto incrementing column called `ID` and a variable length character string column called `Name` that can store at most 10 characters:

```
mysql> CREATE TABLE auto_test (
->     ID INT NOT NULL AUTO_INCREMENT,
->     Name VARCHAR(10),
->     PRIMARY KEY (ID)
-> )
```

A new table is created.



Step 7. Insert rows

1. Insert a few rows into the table:

```
mysql> INSERT INTO auto_test (Name) VALUES
-> ('Pat'), ('Sarah'), ('Kai'), ('Carsten'), ('Max');
```

Five rows are inserted into the table.

Step 8. Verify the results

1. List the contents of the table:

```
mysql> SELECT * FROM auto_test;
```

All the rows have got integer values in the `Id` column:

Id	Name
1	Pat
2	Sarah
3	Kai
4	Carsten
5	Max



276

10.2.5 Creating Tables Based on Existing Tables

MySQL provides two ways to create a table based on another table:

- **`CREATE TABLE...AS SELECT`** creates a table and populates it from the result set returned by an arbitrary **`SELECT`** statement. In this case, the “other table” is the result set retrieved by the **`SELECT`**. The **`SELECT`** can be of (nearly) any type: it may include joins, unions etc and is therefore not bound to only one table.
- **`CREATE TABLE ... LIKE`** creates an empty table using the structure of another existing table.

277

`CREATE TABLE...AS SELECT` can create a table that is empty or non-empty, depending on what is returned by the **`SELECT`** part. The following statements create a table that contains all contents of the **City** table:

```
mysql> CREATE TABLE CityCopy1
-> AS
-> SELECT * FROM City;
```

NOTE: The **AS** keyword is optional in MySQL. However, standard SQL requires the **AS** keyword, and we will usually write the statement including the **AS** keyword

The following statement creates a table that contains a selection of the contents from the **City** table:

```
mysql> CREATE TABLE CityCopy2
-> AS
-> SELECT * FROM City
-> WHERE Population > 2000000;
```

The following statement creates an empty copy of the **City** table:

```
mysql> CREATE TABLE CityCopy3
-> AS
-> SELECT * FROM City LIMIT 0;
```

One of the advantages to using the **`SELECT`** is that you do not have to copy all the columns. The following statement creates a table that contains only specific columns from the **City** table,:

```
mysql> CREATE TABLE CityCopy4
-> AS
-> SELECT col1, col2 FROM City;
```

Using the **`LIKE`** keyword with **`CREATE TABLE`** creates an empty table based on the definition of another table. The result is a new table with a definition that includes all column attributes and most table attributes from the original table, and which also includes primary key and unique constraint definitions as well as index definitions.



278

Suppose that table t looks like this:

```
mysql> CREATE TABLE t(
->     i INT NOT NULL AUTO_INCREMENT,
->     PRIMARY KEY (i)
-> ) ENGINE = InnoDB;
```

The result of **CREATE TABLE...LIKE** differs from the result of using **CREATE TABLE...SELECT** to create an empty table. Either of the following statements will create an empty copy of the table t:

```
mysql> CREATE TABLE copy1 SELECT * FROM t LIMIT 0;
mysql> CREATE TABLE copy2 LIKE t;
```

279

However, the resulting copies differ in the amount of information retained from the original table structure:

```
mysql> SHOW CREATE TABLE copy1\G;
***** 1. row *****
      Table: copy1
Create Table: CREATE TABLE `copy1` (
  `i` int(11) NOT NULL default '0'
) ENGINE=MyISAM DEFAULT CHARSET=latin1

mysql> SHOW CREATE TABLE copy2\G;
***** 1. row *****
      Table: copy2
Create Table: CREATE TABLE `copy2` (
  `i` int(11) NOT NULL auto_increment,
  PRIMARY KEY (`i`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

The **CREATE TABLE...AS SELECT** statement uses the runtime column names and data types from the result set. However, because it creates the table based only on a query result and not on a proper definition of the original base table, it does not retain the **PRIMARY KEY** and/or index definitions, nor the **AUTO_INCREMENT** column attribute. Also, the new table is created to use the default storage engine rather than the actual storage engine utilized by table t.

It is not surprising that **CREATE TABLE...AS SELECT** omits this information. For the runtime query result, concepts such as indexes., constraints and **AUTO_INCREMENT** columns are not applicable, and hence this information cannot be used to create the new table. For **CREATE TABLE LIKE**, this limitation does not apply. Rather than deriving a table definition from a runtime query result, it creates a new table according to the definition of a base table. Therefore, typical table properties and structural elements like table constraints can be recreated without issues.



Some table attributes are not copied, even when issuing `CREATE TABLE...LIKE`. The most notable examples are:

- If the original table is a MyISAM table for which the `DATA DIRECTORY` or `INDEX DIRECTORY` table options are specified, those options are not copied to the new table. The data and index files for the new table will reside in the database directory for the chosen database.
- Foreign key definitions in the original table are not copied to the new table. If you want to retain the foreign key definitions, they must be re-specified with `ALTER TABLE` after creating the copy.





InLine Lab 10-B

In this exercise you will use the **SHOW CREATE TABLE** and **CREATE TABLE...AS SELECT** statements to view the creation of a table from the **world** database and create a new table from an existing table. This will require a mysql command line client and access to a MySQL server.

Step 1. Set the default database

1. From the mysql prompt type;

```
mysql> USE world;
```

Returns the message Database changed confirming the use of the **world** database.

Step 2. Using CREATE TABLE...AS SELECT

1. Create a new table called **CitySelect** by issuing the following statement:

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

A new table is created with all columns of the original table.

Step 3. Check the contents of the new table

1. Check the number of rows in the original **City** table and the newly created table:

```
mysql> SELECT COUNT(*) FROM City;  
mysql> SELECT COUNT(*) FROM CitySelect;
```

Is there a difference in the number of rows? The number of rows is equal to 4079 in both cases

Step 4. Check the definition if the new table

1. Examine the definition of the original **City** Table and compare it to the definition of the newly created table:

```
mysql> SHOW CREATE TABLE City;  
mysql> SHOW CREATE TABLE CitySelect;
```

Do you notice any differences between these table definitions? At least, the **PRIMARY KEY** (and if applicable, **FOREIGN KEYS**) are missing. Depending on the local defaults and the script used to create the **world** database, the defaults for character set, collation and storage engine may differ as well.



Step 5. Using CREATE TABLE..LIKE

1. Create a new table called CityLike by issuing the following statement:

```
mysql> CREATE TABLE CityLike LIKE City;
```

The new table is created.

Step 6. Verify the content of the table

1. Check the number of rows in the original **City** table and the newly created table:

```
mysql> SELECT COUNT(*) FROM City;
mysql> SELECT COUNT(*) FROM CityLike;
```

Is there a difference in the number of rows? Yes - the number of rows in the **City** table is still 4079, but 0 in **CityLike**

Step 7. Verify the structure and table options

1. Examine the definition of the original **City** Table and compare it to the definition of the newly created table:

```
mysql> SHOW CREATE TABLE City;
mysql> SHOW CREATE TABLE CityLike;
```

Do you notice any differences between these table definitions? The new table should have the same layout – at least, all indexes, primary key and unique constraints should be preserved, and table options things like the storage engine.



280

10.2.6 Temporary Tables

It is sometimes useful to create tables for temporary use. This can be done by using the **TEMPORARY** keyword in a **CREATE TABLE** statement. A temporary table differs from a non-temporary table in the following ways:

- It's visible only within the session where it was created. Other sessions cannot access a temporary table created in another session. Multiple sessions can create temporary tables that have the same name and no conflict occurs.
- A temporary table exists only for the duration of the session in which it was created. The server drops a temporary table automatically when the session ends (if it was not already explicitly dropped). This is convenient because you need not remember to remove the table yourself.
- A temporary table may have the same name as a non-temporary table. The non-temporary table becomes temporarily masked for the duration of the temporary table's existence.
- A temporary table can be renamed only with **ALTER TABLE...RENAME TO**. You cannot use **RENAME TABLE**.
- Temporary tables do not show up in the output of **SHOW TABLES**, nor in the `information_schema.TABLES` table.

In this example a temporary table is created from the existing **City** table, which contains the names of all the cities in Texas (US);

```
mysql> CREATE TEMPORARY TABLE Texas
      -> AS
      -> SELECT Name
      -> FROM City
      -> WHERE District='Texas';
```



281

10.3 Altering Tables

After creating a table, you might discover that its structure is not quite suited to its intended use. If that happens, you can change the table's structure. This section addresses some of the methods available for modifying your table structure and data.

10.3.1 Add Columns

For example, to add a column named LocalName the `City` table, you can issue this statement:

```
mysql> ALTER TABLE City ADD COLUMN LocalName VARCHAR(35) CHARACTER SET
utf8
-> NOT NULL DEFAULT '' COMMENT 'The local name of this City';
```

This `ALTER TABLE` statement changes the table structure as follows:

Field	Type	Null	Key	Default	Extra
ID	int(11)	NO	PRI	NULL	auto_increment
Name	char(35)	NO			
CountryCode	char(3)	NO			
District	char(20)	NO			
Population	int(11)	NO		0	
LocalName	varchar(35)	NO			

NOTE: Column names within a table must be unique, so you cannot add a column with the same name as one that already exists in the table. Also, column names are not case sensitive

To add a column at a specific position within a table row, use `FIRST` or `AFTER column_name`. If none of these is specified, the new column is appended after the last column.

Adding a column to a table populates the new column with the `NULL` value, unless the column has a `AUTO_INCREMENT`, `DEFAULT`, or `NOT NULL` option. If `AUTO_INCREMENT` is present, a number will be generated. If a `DEFAULT` option is present, the specified default value will be used. If `NOT NULL` is present, then the default value for the data type will be used.

282

10.3.2 Remove Columns

To remove a column, use a `DROP` clause that names the column to be removed:

```
mysql> ALTER TABLE City DROP COLUMN LocalName;
```

The previous `ALTER TABLE` statement discards the `LocalName` column and restores the original table structure.



283

10.3.3 Altering column definitions

The **ALTER TABLE** statement can also be used to change the definition of columns. This is done using either the **MODIFY** or the **CHANGE** clause.

The **MODIFY** clause must specify the name of the column that you want to change, followed by its new definition. For example, to change the data type of the **ID** column of the **City** from **INT** to **BIGINT UNSIGNED**, us the following statement:

```
mysql> ALTER TABLE City MODIFY ID BIGINT NOT NULL AUTO_INCREMENT;
```

Note that the entire column definition must be provided in the **ALTER TABLE** statement. **MODIFY** will replace the existing definition with the new definition, and for that reason, the **NOT NULL** and **AUTO_INCREMENT** options are also given in the **ALTER TABLE** statement. If those would have been omitted, they would have been discarded.

NOTE: MySQL will sometimes try to fit existing data in to the new data type by applying data type conversions.

There are many more options for table modification using **ALTER TABLE**. Refer to the MySQL Reference Manual for more information.





InLine Lab 10-C

In this exercise you will use **ALTER TABLE** statements to add, remove and modify tables. This will require a mysql command line client and access to a MySQL server.

Step 1. Change the default database

1. Change the database to the **test** database and make sure the SQL Mode is the empty mode:

```
mysql> USE test;
mysql> SET SQL_MODE = '';
```

Returns the message Database changed confirming the use of the **test** database.

Step 2. Create a table

1. Create a new table called **foo** with one column called **bar** which has the **TEXT** data type. Set the storage engine to MyISAM:

```
mysql> CREATE TABLE foo (bar TEXT) ENGINE=MyISAM;
```

Returns OK and creates new table.

Step 3. Verify

1. Use the **DESCRIBE** statement to show the current structure of the **foo** table:

```
mysql> DESCRIBE foo;
```

Confirm that the table has been created :

Field	Type	Null	Key	Default	Extra
bar	text	YES		NULL	

Step 4. Adding a column

1. Add a not-nullable column to the **foo** table called **Id**, with a **TINYINT** data type:

```
mysql> ALTER TABLE foo ADD Id TINYINT NOT NULL;
```

Returns OK.



Step 5. Verify the structure

1. Use the **DESCRIBE** statement to inspect the current structure of the **foo** table:

```
mysql> DESCRIBE foo;
```

Confirm that the **Id** column has been added with the proper attributes

Field	Type	Null	Key	Default	Extra
bar	text	YES		NULL	
Id	tinyint(4)	NO		NULL	

Step 6. Insert rows

1. Add a few rows to the **foo** table:

```
mysql> INSERT INTO foo (Id, bar) VALUES
    (1, 'This is the 1st entry')
    ,(2, 'I guess this is the 2nd entry then')
```

The rows are added to the table

Step 7. Verify

1. Retrieve all contents from the **foo** table

```
mysql> SELECT * FROM foo;
```

The rows are retrieved

Step 8. Remove the **Id column**

1. Remove the **Id** column from the **foo** table:

```
mysql> ALTER TABLE foo DROP Id;
```

Returns OK.

Step 9. Verify

1. Use the **DESCRIBE** statement to verify that the column is dropped.

```
mysql> DESCRIBE foo;
```

Confirm that the **Id** column has been removed.

Step 10. Change the **foo column**

1. Change the data type of the bar column to not-nullalble a **CHAR** (8) column

```
mysql> ALTER TABLE foo MODIFY bar CHAR(8) NOT NULL;
```

Returns OK , but gives warnings.

```
Query OK, 2 rows affected, 2 warnings (#.## sec)
Records: 2  Duplicates: 0  Warnings: 2
```

Step 11. Inspect the warnings

1. Examine the warnings:

```
mysql> SHOW WARNINGS
```

Reports that data was truncated.

Level	Code	Message
Warning	1265	Data truncated for column 'bar' at row 1
Warning	1265	Data truncated for column 'bar' at row 2



284

10.3.4 Changing Columns

The second way to alter a column definition is to use a **CHANGE** clause. **CHANGE** enables you to modify both the column's definition and its name. To use this clause, specify the **CHANGE** keyword, followed by the column's existing name, its new name, and its new definition, in that order. Note that this means you must specify the existing name twice if you want to change only the column definition (and not the name). For example, change the LastName column from **CHAR(30)** to **CHAR(40)** without renaming the column as follows:

```
mysql> ALTER TABLE HeadOfState CHANGE LastName LastName CHAR(40) NOT NULL;
```

To change the name as well (for example, to Surname), provide the new name following the existing name:

```
mysql> ALTER TABLE HeadOfState CHANGE LastName Surname CHAR(40) NOT NULL;
```

To change a column at a specific position within a table row, use **FIRST** or **AFTER col_name**. The default is to change the last column.

285

10.3.5 Renaming Tables

Renaming a table changes neither a table's structure nor its contents. The following statement renames table **t1** to **t2**:

```
mysql> ALTER TABLE t1 RENAME TO t2;
```

Another way to rename a table is by using the **RENAME TABLE** statement:

```
mysql> RENAME TABLE t1 TO t2;
```

RENAME TABLE has an advantage over **ALTER TABLE** in that it can perform multiple table renames in a single operation. One use for this feature is to swap the names of two tables:

```
mysql> RENAME TABLE t1 TO tmp, t2 TO t1, tmp TO t2;
```

For **TEMPORARY** tables, **RENAME TABLE** does not work. You must use **ALTER TABLE** instead.





InLine Lab 10-D

In this exercise you will use **ALTER TABLE** and **RENAME TABLE** statements to change and rename a table. This will require a mysql command line client and access to a MySQL server.

Step 1. Set the default database

1. Change the database to the **test** database:

```
mysql> USE test;
```

Returns the message Database changed confirming the use of the **test** database.

Step 2. Inspect the **foo table**

1. Use the **DESCRIBE** command to show the current structure of the **foo** table:

```
mysql> DESCRIBE foo;
```

Confirm that the bar column exists with the **CHAR(8)** data type:

Field	Type	Null	Key	Default	Extra
bar	char(8)	NO			

Step 3. Change

1. Change the bar column in the **foo** table to allow **NULL** values:

```
mysql> ALTER TABLE foo CHANGE bar bar CHAR(8) NULL;
```

Returns OK.

Step 4. Verify

1. Use the **DESCRIBE** statement to show the current structure of the **foo** table:

```
mysql> DESCRIBE foo;
```

Confirm that the bar column has been changed to allow **NULL** values.



Step 5. Rename table

1. Rename the **foo** table to **bar**:

```
mysql> RENAME TABLE foo TO bar;
```

Returns OK.

Step 6. Verify

1. Use the **DESCRIBE** command to show the current structure of the **bar** table:

```
mysql> DESCRIBE bar;
```

Confirm that the **bar** table is identical to the renamed **foo** table.



286

10.4 Dropping Tables

DROP TABLE removes one or more tables. All table data and the table definition are permanently removed, so be careful with this statement!

```
DROP [TEMPORARY] TABLE [IF EXISTS] table1;
```

Use **IF EXISTS** to prevent an error from occurring in case the table does not exist. Otherwise an error is generated for when the specified table does non-exist.

NOTE: Note: Like **DROP DATABASE**, a **DROP TABLE** statement cannot be undone.

The **TEMPORARY** keyword has the following effects:

- The statement drops only **TEMPORARY** tables.
- The statement does not end an ongoing transaction (transactions are covered later in the course).

Using **TEMPORARY** is a good way to ensure that you do not accidentally drop a non-**TEMPORARY** table.

DROP TABLE examples:

```
mysql> DROP TABLE table1;  
mysql> DROP TABLE IF EXISTS table1;  
mysql> DROP TEMPORARY TABLE CityTmp;
```





InLine Lab 10-E

In this exercise you will use **DROP TABLE** statement to remove previously created tables. This will require a mysql command line client and access to a MySQL server.

Step 1. Set the default database

1. Change the database to the **test** database:

```
mysql> USE test;
```

Returns the message: Database changed confirming the use of the **test** database.

Step 2. Show the available tables

1. Show all the current tables:

```
mysql> SHOW TABLES;
```

Lists all the **test** database tables.

Step 3. Drop a table

1. Remove the **db** table:

```
mysql> DROP TABLE db;
```

Returns an OK message.

Step 4. Drop another table

1. Remove the **bar** table:

```
mysql> DROP TABLE bar;
```

Returns an OK message.

Step 5. Verify

1. Show all the current tables:

```
mysql> SHOW TABLES;
```

Confirm that the above tables have been removed.



287

10.5 Foreign Key Constraints

In this section we will discuss a number of related but distinct concepts:

- Relationships
- Foreign keys
- Foreign key constraints
- Referential integrity

Foreign keys are used to symbolize references between rows throughout the databases. As such, foreign keys are used to implement *relationships* between rows of data. *Foreign key constraints* are used to maintain foreign keys and to ensure the references are kept consistent. As such foreign key constraints are used to enforce *referential integrity*.

NOTE: Often, the term 'foreign key' is used to denote 'foreign key constraint'. In many cases it is fine to blur this distinction. However, in this section it is necessary to make the distinction to explain the benefit of foreign key constraints as opposed to foreign keys that are not enforced using constraints.

288

10.5.1 Foreign keys and relationships

Let's take a look at the **world** database to see which relationships are present there.

The tables in the **world** database each contain a particular kind of data. A row in one such table represents a particular thing in the real world: a row in the **Country** table represents a real country, and a row in the **City** table represents a real city.

In the real world, cities do not occur by themselves: they are always cities that reside in a particular country. In other words, countries contain cities, and countries and cities are related to one another through this containment relationship. Because of this relationship we can ask questions like "How many cities are situated in Finland?" or "In what country is the city called London?".

It doesn't stop there: there is another relationship between countries and cities. In most cases, of all the cities that belong to a country, there is one 'special' city known as the capital. Often, the capital is the largest cultural and political center of the entire country.

So, cities and countries can be related in more than one way: a particular city A resides in a particular country B, and that country B can have a capital C which is also a city in that country. Because it is possible, or in fact very likely that the city A is different from the city C, we can see that these must be two entirely independent relationships between cities and countries.



289

Foreign keys represent relationships

Now, we just said that in the **world** database the individual objects from the real world such as cities and countries are represented as rows in a table. The relationships between the real cities and the real countries should be captured too in the **world** database. In other words, if real countries and cities are related to one another, then rows from the **City** table and the rows from the **Country** table must be likewise related.

In the real world, the relationship between a country and its cities is fairly physical: if we take the border of the country, then all cities that have a border that is enclosed within the border of that country are cities that belong to that country (there are actually interesting exceptions to this rule which we won't discuss here). Now, in a relational database, the relationship between rows are not of such a physical nature: rows do not contain collections of other rows. So there must be another way to represent these relationships.

The device that is used to represent relationships between rows in a database is called a *foreign key*. A foreign key is nothing more than a collection of one or more columns that has a combination of values in common with that of another collection of columns, usually in another table. Because there is a common combination of values in these two sets of columns, two sets of corresponding rows can be identified that are apparently related to each other through this common set of values.

So, in the database the relationship is not physically present, rather, it is represented symbolically using column values. The column values are used as a reference to the related row.

290

Foreign keys in the **world** database

For example, in the **world** database, the **City** table has a **CountryCode** column. This is a foreign key that relates to the **Code** column in the **Country** table. When we have a particular row from the **Country** table, we can use the value of its **Code** column to look up those rows in the **City** table that have an identical value in their **CountryCode** column. All the rows from the **City** table that have a **CountryCode** value that matches the **Code** value from that row from the **Country** table apparently belong to the country it represents.

Likewise, the **Country** table contains a **Capital** column: this may be used to find a row in the **City** table that has an identical value in its **ID** column. The row from the **City** table that has an identical value in its **ID** column apparently represents the city that is the capital of the country.

So, common column values are used to represent a relationship between the respective rows from these two tables.

291

Foreign key examples

Let's make this more concrete using some SQL statements. Let's find a particular row from the **City** table:

```
mysql> SELECT ID, Name, CountryCode FROM City
      -> WHERE Name LIKE 'Helsinki %';
+----+-----+-----+
| ID | Name           | CountryCode |
+----+-----+-----+
| 3236 | Helsinki [Helsingfors] | FIN         |
+----+-----+-----+
```



In the **City** table, the value for **CountryCode**, 'FIN' is a foreign key. It refers to the value of the **Code** column (the primary key) of some row in the **Country** table. Most people will immediately recognize 'FIN' as the code for Finland, and the next statement confirms this assumption:

```
mysql> SELECT Code, Name, Capital FROM Country WHERE Code = 'FIN';
+----+-----+-----+
| Code | Name   | Capital |
+----+-----+-----+
| FIN  | Finland |      3236 |
+----+-----+-----+
```

So, by using the foreign key stored in the **CountryCode** column of the **City** table, we can successfully look up the corresponding row from the **Country** table. This lookup allowed us to answer the question "In which country is the city called 'Helsinki' situated?"

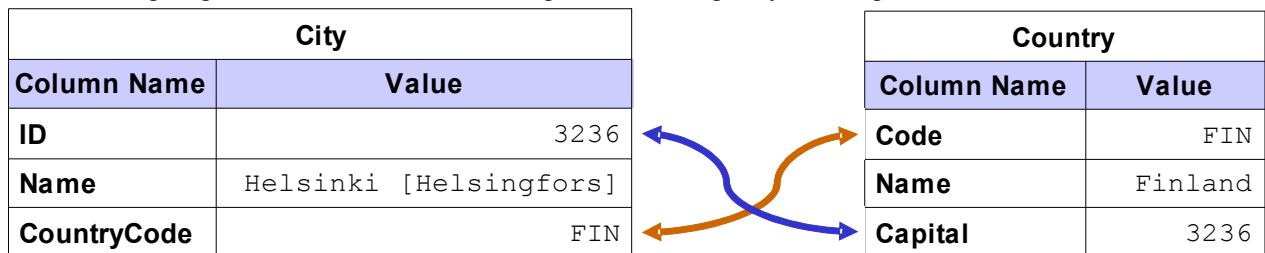
We used the previous statement also to obtain the foreign key for the country's capital. The value 3813 refers to a particular entry in the **ID** column (primary key) of the **City** table, and we may retrieve the corresponding row using the following statement:

```
mysql> SELECT ID, Name, CountryCode FROM City WHERE ID = 3236;
+----+-----+-----+
| ID  | Name           | CountryCode |
+----+-----+-----+
| 3236 | Helsinki [Helsingfors] | FIN          |
+----+-----+-----+
```

So this last statement answers the question: "What is the capital of Finland?"

292

The following diagram illustrates these relationships and the foreign keys that implement them:



293

10.5.2 Foreign Key Constraints and Referential Integrity

Although the **world** database uses foreign keys to symbolize relationships between countries and cities, there is an important problem that needs to be considered. What would happen if the country code would change for a particular country?

Suppose we would change the country code for Romania from 'ROM' to 'ROU'. The following statement would achieve that change:

```
mysql> UPDATE Country SET Code = 'ROU' WHERE Code = 'ROM';
query OK, 1 row affected (#.# sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Although the country code is now changed in the **Country** table, the corresponding rows in the **City** table remain unchanged. They still use the old country code 'ROM', even though the **Country** table does not contain a row that has 'ROM' as country code anymore.

Basically, the 'ROM' rows in the **City** table now point to no country at all: the link to the referenced row is broken and the city rows are said to be orphaned. A more formal way of putting it is to say that the referential integrity was compromised. The **City** rows that have the value 'ROM' in the **CountryCode** column look like they refer to the country with the code 'ROM', but when we try to find the actual row that represents that country, we find that it does not exist.

For the row from the **Country** table with the **Code** 'ROU', the situation is of course also bad, as there seem to be no cities inside it. However, technically this does not constitute a violation of the referential integrity, as the country does not reference a city that does not exist.

Obviously, we have caused a very undesirable situation by changing the country code. Essential information was unintentionally lost by changing just one little piece of data. Of course, it is possible to update the rows in the **City** table and replace the value for **CountryCode** from 'ROM' to 'ROU' there as well. But still, one should realize that in the mean while, the database is in an inconsistent state. Worse, there is nothing in the data itself from which we can derive that the **City** rows that have 'ROM' **CountryCode** values should actually have 'ROU' instead.

Let's recall that a foreign key is nothing more than the occurrence of a common value. We happen to have attached the special meaning to it, namely that **CountryCode** values in the **City** table reference the **Code** values in the **Country** table. However, the database does nothing to support that. It allows us to unintentionally and irrecoverably lose information about relationships, orphaning many rows in the process. In other words, the database does not *enforce* referential integrity.

In order to sanitize the behavior, we need something to guard our foreign keys. This is what foreign key *constraints* are for.



294

Foreign key Constraints

By creating a foreign key constraint, one can ensure that changes do not accidentally result in lost references. Another way of putting it is to say that foreign keys enforce referential integrity. A foreign key constraint will simply prevent a foreign key from referencing something that is not there.

Foreign key constraints take care of two things:

- They prevent additions or changes to the referencing table that would result in a reference to something that does not exist.

For example, currently the **City** table would allow changes that update the **CountryCode** column to, say '**ZZZ**'. As there is no corresponding row in the **Country** table that has a **Code** equal to '**ZZZ**' this change would result in compromising the referential integrity, and this should of course not be allowed. Similarly, new rows must be prevented from being entered into the table if they do not have a valid value for the **CountryCode** column.

- Changes to referenced rows can either be prevented or propagated to the referencing rows.

For example, we just presented the example where the row from the **Country** table with the value '**ROM**' in the **Code** column was changed to '**ROU**'. A foreign key constraint can be defined in such a way that this change would have been prevented as long as there are still rows that still use the '**ROM**' **Code** to refer to the corresponding row in the **Country** table.

Alternatively, a foreign key constraint can be defined to automatically propagate (cascade) the change in **Code** column of the **Country** table to the corresponding rows in the **City** table. In this case, the change from '**ROM**' to '**ROU**' in the **Country** table would be automatically applied also to the '**ROM**' rows in the **City** table, changing the code from '**ROM**' to '**ROU**' in all places where it occurs.

295

Creating foreign key constraints

Foreign keys constraints may be specified as part of the **CREATE TABLE** syntax. For example, the following **CREATE TABLE** statement creates the **City** table with a foreign key constraint to enforce the referential integrity of the **CountryCode** foreign key:

```
mysql> CREATE TABLE City (
->   ID INT NOT NULL,
->   Name CHAR(35) NOT NULL,
->   CountryCode CHAR(3) NOT NULL,
->   District CHAR(20) NOT NULL,
->   Population INT NOT NULL,
->   PRIMARY KEY (ID),
->   FOREIGN KEY (CountryCode) REFERENCES Country (Code)
-> ) ENGINE=InnoDB;
```



Alternatively they can be added to existing tables using an **ALTER TABLE** statement. The following statement adds a foreign key constraint to the existing **City** table:

```
mysql> ALTER TABLE City
-> ADD FOREIGN KEY (CountryCode) REFERENCES Country (Code);
```

The full foreign key syntax is a bit more elaborate than shown in these examples:

296

```
[CONSTRAINT [name]]
FOREIGN KEY [name] (referencing_col1[, ..., referencing_colN])
REFERENCES referenced_tab (referenced_col1[, ..., referenced_colN])
[ON DELETE {CASCADE | NO ACTION | RESTRICT | SET NULL}]
[ON UPDATE {CASCADE | NO ACTION | RESTRICT | SET NULL}]
```

So, the mandatory elements in the foreign key constraint syntax are:

- A list of referencing columns (columns in the foreign key table)
- The name of the referenced table
- A list of referenced columns. For each referencing column, exactly one matching column from the referenced table must be supplied. The data types of each referencing/referenced column pair must be identical, including any **UNSIGNED**, **CHARACTER SET** and **COLLATE** attributes.
- For practical purposes, the referenced columns should together form a **PRIMARY KEY** constraint. As we shall see, the rule is actually a bit more relaxed, however it is best to assume that all referenced columns are **PRIMARY KEY** columns and, vice versa, that all columns from the **PRIMARY KEY** are referenced.

The optional elements are:

297

- The constraint name. This is optional in the syntax, however, each foreign key must have a name, and if no name is given in the DDL statement, a name is automatically generated.
- **DELETE** rule – specifies what should happen to the referencing rows in case a referenced row is removed.
 - **CASCADE** means that the **DELETE** must be propagated to any referencing rows. In other words, removal of the referenced row will lead to removal of the referencing rows as well.
 - **NO ACTION** means that a **DELETE** of a row from the referenced table must not occur if there are still referencing rows. If an attempt is made to remove a referenced row, a runtime error occurs.
 - **RESTRICT** means the same as **NO ACTION**
 - **SET NULL** means that the referencing columns in the referencing rows are changed to **NULL**. This is only possible if these columns are nullable.

Note that if all foreign key columns are **NULL**, than that foreign key does not reference any row in particular. Therefore, it is **not** an orphaned row. An orphaned row does refer to another row, but the referenced row is simply not present, indicating a violation of the referential integrity.

It should be noted that in case no explicit **DELETE** rule is given, a **DELETE** of a referenced row is prevented as if **RESTRICT** would be explicitly used to define a **DELETE** rule.



- **UPDATE** rule - specifies what should happen to the referencing rows in case a referenced row is changed. The values **CASCADE**, **NO ACTION**, **RESTRICT**, and **SET NULL** have similar semantics as for the **DELETE** rule.

It should be noted that in case no explicit **UPDATE** rule is given, a **UPDATE** of a referenced row is prevented as if **UPDATE** would be explicitly used to define a **UPDATE** rule.

298

10.5.3 Foreign Key Constraints and MySQL Storage Engines

Currently, MySQL foreign keys are implemented at the storage engine level.

The **InnoDB** engine is currently the only supported engine that provides a foreign key implementation. This means that foreign keys constraints can be created only if both the referencing as well as the referenced table are **InnoDB** tables.

So, in order to add foreign key constraints to the **world** database, we first need to change the storage engine to **InnoDB**:

```
mysql> ALTER TABLE City ENGINE = InnoDB;
Query OK, 4079 rows affected (#.## sec)
Records: 4079  Duplicates: 0  Warnings: 0
```

NOTE: There are third party engines that also support foreign key constraints, but **InnoDB** is currently the only engine supported by Sun Microsystems, Inc.

When attempting to create a foreign key on a non-**InnoDB** table, MySQL will silently ignore the request. Not even a warning will be issued so it is always a good idea to double check whether a foreign keys was indeed created.

When attempting to create a foreign key that references a non-**InnoDB** table, a runtime error occurs:

```
mysql> ALTER TABLE City
      -> ADD CONSTRAINT fk_city_country
      -> FOREIGN KEY (CountryCode) REFERENCES Country(Code);
ERROR 1005 (HY000): Can't create table 'world.#sql-818_2' (errno: 150)
```

The error number 150 indicates some structural error in creating a foreign key constraint. More information can be obtained by observing the status of the **InnoDB** engine:

299

```
mysql> SHOW ENGINE InnoDB STATUS;
```



This returns a rather lengthy report of what is going on inside the InnoDB engine. You should be looking for the heading marked LATEST FOREIGN KEY ERROR:

```
-----  
LATEST FOREIGN KEY ERROR  
-----  
080310 17:04:04 Error in foreign key constraint of table world/#sql-818_2:  
FOREIGN KEY bla (CountryCode) REFERENCES Country(Code):  
Cannot resolve table name close to:  
(Code)
```

Here, the message that the table name cannot be resolved should be taken to mean that InnoDB is looking for a table name near the occurrence of **(Code)** in the DDL statement, but can't seem to find one. This is because the **Country** table is not a InnoDB table: InnoDB doesn't know anything about the existence of any non-InnoDB tables.

So, we must first change the engine of the **Country** table to be InnoDB too before we can successfully create the foreign key constraint.

A number of things concerning the InnoDB implementation of foreign keys should be noted here:

300

- InnoDB requires an index to be present on the referencing columns. If such an index is not present already, one is automatically created
- InnoDB requires the referenced columns to be the leftmost columns of some index defined on the referenced table. The index need not be unique, and not all columns of the index need be referenced (that is, there may be more columns to the right that are not referenced by the foreign key).

It should be noted that this is unlike most (if not all) other RDBMS products. Most RDBMS products require the referenced columns in a **FOREIGN KEY** constraint to define either a **PRIMARY KEY** or a **UNIQUE** constraint, ensuring that a single foreign key references exactly one row in the referenced table.

- When changes are made to the data in either the referencing or the referenced tables, the foreign key constraint is checked in a row-by-row fashion. The first change that would introduce a violation of the referential integrity results in a runtime error, even if execution of the entire statement would leave the tables in a consistent state.
- MySQL accepts the syntax for 'inline' foreign key constraints, that is, foreign key constraint definitions at the column level. However, those are silently discarded.





InLine Lab 10-F

In this exercise you will set-up a table and manipulate the foreign key settings. This will require a MySQL command line client and access to the `mysql` server and the `world` database.

Step 1. Set the default database

1. Change the database to `world`. Type;

```
mysql> USE world;
```

Database changed.

Step 2. Create Tables

1. Create the tables from the above examples (`CountryParent` and `CityChild`). Type;

```
mysql> CREATE TABLE CountryParent(
    ->     Code CHAR(3) NOT NULL,
    ->     Name CHAR(52) NOT NULL,
    ->     PRIMARY KEY (Code)
    -> ) ENGINE = InnoDB;
```

...and...

```
mysql> CREATE TABLE CityChild(
    ->     ID INT NOT NULL AUTO_INCREMENT,
    ->     Name CHAR(35) NOT NULL,
    ->     CountryCode CHAR(3)NOT NULL,
    ->     PRIMARY KEY (ID),
    ->     FOREIGN KEY (CountryCode) REFERENCES CountryParent (Code)
    ->     ON UPDATE CASCADE ON DELETE CASCADE
    -> ) ENGINE = InnoDB;
```



Step 3. Verify the table structure

1. Use **SHOW CREATE TABLE** to inspect the table structure

```
mysql> SHOW CREATE TABLE CountryParent
```

...and...

```
mysql> SHOW CREATE TABLE CityChild
```

Shows the definition of both tables. Verify that a foreign key constraint is present in the **CityChild** table.
Note that an index was automatically created:

```
***** 1. row *****
Table: CountryParent
Create Table: CREATE TABLE `CountryParent` (
`Code` char(3) NOT NULL,
`Name` char(52) NOT NULL,
PRIMARY KEY (`Code`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

and

```
***** 1. row *****
Table: CityChild
Create Table: CREATE TABLE `CityChild` (
`ID` int(11) NOT NULL AUTO_INCREMENT,
`Name` char(35) NOT NULL,
`CountryCode` char(3) NOT NULL,
PRIMARY KEY (`ID`),
KEY `CountryCode` (`CountryCode`),
CONSTRAINT `citychild_ibfk_1` FOREIGN KEY (`CountryCode`) REFERENCES
`CountryParent` (`Code`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```



Step 4. Insert data

1. Insert some rows into both tables:

```
mysql> INSERT INTO CountryParent  
-> SELECT Code, Name FROM Country WHERE Region LIKE 'Nordic%'
```

...and...

```
mysql> INSERT INTO CityChild  
-> SELECT Id, Name, CountryCode FROM City  
-> WHERE CountryCode IN (  
-> SELECT Code FROM Country WHERE Region LIKE 'Nordic%')
```

Fills the tables with some rows.

Step 5. Verify

1. Examine the contents of both tables:

```
mysql> SELECT * FROM CountryParent;
```

...and...

```
mysql> SELECT * FROM CityChild;
```

Shows the contents of both tables

Step 6. Perform a DELETE

1. Delete Finland from the **CountryParent** table and examine the rows in the **CityChild** table

```
mysql> DELETE FROM CountryParent WHERE Code = 'FIN';
```

...and...

```
mysql> SELECT * FROM CityChild;
```

All the Finnish cities from **CityChild** have disappeared as well



Step 7. Attempt an insert

1. Insert a row into the **CityChild** table:

```
mysql> INSERT INTO CityChild (Name, CountryCode)
-> VALUES ('Atlantis', 'ATL');
```

An error occurs:

```
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`world`.`CityChild`, CONSTRAINT `citychild_ibfk_1` FOREIGN
KEY (`CountryCode`) REFERENCES `country` (`Code`))
```





Further Practice

In this exercise you will use your knowledge about the topics covered in this chapter to create, view and manipulate tables.

301

1. Change the current database to the **test** database.

2. Create two new tables:

1. 'students' with two columns; ['id' of the **SMALLINT UNSIGNED** data type, having following attributes; **AUTO_INCREMENT PRIMARY KEY**] and ['name' with the following attributes; **VARCHAR(100)**]
2. 'enrollments' with two columns; ['studentid' of the **SMALLINT UNSIGNED** data type; and ['name' of the **VARCHAR(100)** data type and a default value of **NULL**]

Show the **CREATE TABLE** statement for both the above tables to confirm accurate creation.

3. Alter the **enrollments** table to specifically set the storage engine to **InnoDB** in order to support foreign keys.
4. Assign the **studentid** column in the **enrollments** table as a foreign key with a reference to the **id** column of the **students** table. Show the **CREATE TABLE** statement to confirm the change.

NOTE: Note: Both tables need to use the **InnoDB** storage engine in order for this exercise to work.

5. Modify the **enrollments** table to change the string length for the 'name' column to 50 and add a default string value of 'New Student' and disallow **NULL** values. Confirm new table structure.
6. Rename the **enrollments** table to **t2**.
7. Remove the **enrollments** table if it exists. Check the warnings.
8. Remove the **t2** and **students** tables. Confirm that the tables no longer exist.
9. You will need to create two new tables within the **world** database in order to do this exercise:

Table 1: Call this table 'europe' and it should be created from the **Country** table. 'europe' will consist of the name and population from **Country** for the entire continent of 'Europe'.

Table 2: Call this table 'asia' and it should be created from the **Country** table. 'asia' will consist of the name and population from **Country** for the entire continent of 'Asia'.

- a) Perform a query that will return all the data from both of the above tables simultaneously.
- b) Perform a query that will return a list of the country name and population (in that order) from the **europe** table, where the population is less than 500,000, and simultaneously returns the population and name (in that order) from the **asia** table, where the population exceeds 10,000,000.

Hint: Try using the **UNION** clause with the **SELECT**.

Optional exercise extended from the Further Practice in the "Databases" chapter:

10. Create a new database called '**new_world**'.
11. Create the newly planned tables with the respective columns and column designations from your database design in the "Databases" chapter.



10.6 Chapter Summary

This chapter introduced MySQL database tables. In this chapter, you learned to:

302

- Assign appropriate table properties
- Assign appropriate column options
- Create a table
- Alter a table
- Empty a table
- Remove a table
- Understand and use primary and Unique constraints
- Assign and use foreign keys





11 MANIPULATING TABLE DATA

11.1 Learning Objectives

This chapter introduces manipulating table data in MySQL. In this chapter, you will learn to:

304

- Insert data into a table
- Delete data from a table
- Update data in a table
- Replace data in a table
- Truncate data from a table



305

11.2 The INSERT Statement

Now that you have created your tables, the next step is to populate them with data. Although there are a variety of ways to get data into MySQL tables, you'll use the **INSERT** statement.

The **INSERT** statement is a common method for adding new rows of data into a table. There are three variants of the **INSERT** syntax, the **INSERT ... VALUES**, **INSERT ... SET** and the **INSERT ... SELECT** syntax

11.2.1 INSERT ... VALUES Syntax

The **INSERT ... VALUES** syntax is useful for adding new rows created from 'scratch' or more precisely, created from literal values:

```
INSERT  
INTO table_name [(column_list)]  
VALUES row_list
```

- The *table_name* is the name of the table to which new rows are to be added.
- The *column_list* is optional. If it is specified, it must be a comma-separated list of column names from the specified table, enclosed in parentheses. If present, the order of the columns dictates the order of the values as they appear in the rows that are to be added to the table. Here's an example of a valid column list for the **City** table from the **world** database:

```
(ID, Name, CountryCode)
```

The column list may be absent, in which case the **INSERT** statement will implicitly assume a column list that contains

- The *row_list* is a list of one or more so called *row constructors*. A row-constructor is basically a comma-separated list of value-expressions (such as literals) enclosed in parentheses. If the column list is present, the order and number of the values in the row constructor must correspond to the order (and number) of columns in the column list. Here's an example of a row-constructor that corresponds with the column list we just presented for the **City** table:

```
(NULL, 'Essaouira', 'MAR')
```

There must be at least one row constructor, but multiple rows may be specified provided they are separated by a comma:

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



306

11.2.2 INSERT ... SET Syntax

The `INSERT ... SET` clause can also be used to indicate column names and values. The above example can also be written with `SET` as follows;

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

The resulting rows will be the same as the `INSERT ... VALUE` example shown earlier.

307

11.2.3 INSERT ... SELECT Syntax

The `INSERT...SELECT` syntax is useful for copying rows from an existing table, or (temporarily) storing a result set from a query:

```
INSERT
INTO table_name [(column_list)]
query_expression
```

- The `table_name` and `column list` have the same meaning as discussed for the `INSERT...VALUES` statement. The table name specifies the table that is to receive new rows, and the optional column list specifies the order of the receiving columns
- The `query_expression` is mandatory. If a `column_list` is specified, this query must produce exactly as many columns as specified by the `column_list`:

```
mysql> INSERT INTO Top10_Cities(ID, Name, CountryCode)
-> SELECT ID, Name, CountryCode
-> FROM City
-> ORDER BY Population DESC
-> LIMIT 10;
```

If no `column_list` is specified, the query must produce exactly the same number of columns as is present in the specified table.



308

11.2.4 INSERT with LAST_INSERT_ID

You can use the MySQL-specific function `LAST_INSERT_ID()` to retrieve the first value generated for an `AUTO_INCREMENT` column by the last successful `INSERT` statement:

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

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

1 row in set (#.## sec)
```

It is important to realize that the value returned by `LAST_INSERT_ID()` may not be the value generated for last inserted row. When last `INSERT` statement inserted multiple rows, only the first value generated for `AUTO_INCREMENT` an column is returned.

If a table contains a column with the `AUTO_INCREMENT` attribute and `INSERT` statement inserts a row, the `LAST_INSERT_ID()` function returns the generated value.. If the statement updates a row instead, `LAST_INSERT_ID()` is not meaningful.

Instructor Note: The result from the query "SELECT LAST_INSERT_ID()" may produce a different result based on what queries have been executed up to this point. If the instructor (or student) has been issuing the queries up to this point, the result will definitely not be accurate. However, if you wish to obtain the exact same results, feel free to drop the world database and source the world.sql file prior to running these commands and the results will be the same as printed here.





InLine Lab 11-A

In this exercise you will use the **INSERT** statement to add new data to existing tables. This will require a mysql command line client and access to a MySQL server.

Step 1. Insert a single value

1. Enter the following statement to create a new table;

```
mysql> CREATE TABLE numbers (n SMALLINT UNSIGNED);
```

2. Now insert the value 5 into the table:

```
mysql> INSERT INTO numbers VALUES (5);
```

3. Retrieve the contents of the new numbers table:

```
mysql> SELECT * FROM numbers;
```

The new table numbers will contain a single column value of 5, shown in the query.

Step 2. Insert multiple values

1. Create another table:

```
mysql> CREATE TABLE birthdates (name VARCHAR(60), bdate DATE);
```

2. Now insert a birthday for 'joe' on 'February 15, 1950' (using the default date format):

```
mysql> INSERT INTO birthdates VALUES ('joe', '1950-02-15');
```

3. Now insert a birthday for 'jane' on 'July 30, 1980' (using the default date format):

```
mysql> INSERT INTO birthdates VALUES ('jane', '1980-07-30');
```

4. View the contents of the birthdates table:

```
mysql> SELECT * FROM birthdates;
```

Returns a list of two rows with two columns (name, bdate), with information for joe and jane;

name	bdate
joe	1950-02-15
jane	1980-07-30



NOTE: Although it is acceptable to use the above syntax (leaving out the column list), it is good practice to include the column list as follows:

```
INSERT INTO birthdates (name, bdate) VALUES ('joe', '1950-02-15')
INSERT INTO birthdates (name, bdate) VALUES ('jane', '1980-07-30')
```

Step 3. Causing an error

- Using the existing **Country** table in the **world** database, insert a row using the values 'SWE' and 'MySQL' for the columns code and name respectively:

```
mysql> INSERT INTO Country (Code, Name) VALUES ('SWE', 'MySQL')
```

You will get an error indicating that an attempt was made to insert a duplicate row.

NOTE: The method for avoiding this is covered later in this chapter.

Step 4. TIMESTAMP and INSERT

- Enter the following statement to create a new table which contains a column that records the current time when data is inserted;

```
mysql> CREATE TABLE students (name VARCHAR(60), modtime TIMESTAMP)
```

- Now insert a student named 'Peter':

```
mysql> INSERT INTO students (name) VALUES ('Peter')
```

- Now insert a student named 'Tariq':

```
mysql> INSERT INTO students (name) VALUES ('Tariq')
```

- View the contents of the new **students** table to confirm the inserts:

```
mysql> SELECT * FROM students;
```

The new table **students** will contain a row for each new student, including the name and the exact date/time of the entry;

name	modtime
Peter	2006-08-14 16:04:51
Tariq	2006-08-14 16:05:03

NOTE: Your date/time information will be current, therefore different than that shown above.



309

11.3 The DELETE Statement

To remove rows from tables, use the **DELETE** statement. To empty a table entirely, delete its rows with the following syntax:

```
DELETE FROM table_name
```

The **DELETE** statement allows a **WHERE** clause to specify exactly which rows are to be removed. If a **WHERE** clause is present, only those rows that satisfy the condition following the **WHERE** keyword are deleted:

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

For example, the following statement will remove all languages that are not official languages for the country in which the language is spoken:

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

The syntax for the condition following the **WHERE** keyword clause is exactly like described earlier.

The **DELETE** statement removes entire rows. Therefore the **DELETE** syntax does not include a specification of columns.

310

11.3.1 Using DELETE with ORDER BY and LIMIT

DELETE supports **ORDER BY** and **LIMIT** clauses, which provide finer control over the way records are deleted. For example, **LIMIT** can be useful if you want to remove only some instances of a given set of records. Suppose that the **people** table contains five records where the name column equals 'Emily'. If you want only one such record, use the following statement to remove four of the duplicated records.

```
mysql> DELETE FROM people WHERE name='Emily' LIMIT 4;
```

Normally, MySQL makes no guarantees about which four of the five records selected by the **WHERE** clause it will delete. An **ORDER BY** clause in conjunction with **LIMIT** provides better control. For example, to delete four of the records containing 'Emily' but leave the one with the lowest id value, use **ORDER BY** and **LIMIT** together as follows:

```
mysql> DELETE  
-> FROM people  
-> WHERE name='Emily'  
-> ORDER BY id DESC  
-> LIMIT 4;
```

The **DELETE** result will indicate number of rows affected, which can be zero (0) if the statement did not cause a change to be made.





InLine Lab 11-B

In this exercise you will use the **DELETE** statement to remove data from existing tables. This will require a mysql command line client and access to a MySQL server.

Step 1. Create a table

1. Create a table called **City2** with the same definition as the **City** table

```
mysql> CREATE TABLE City2 LIKE City;
```

The new table is created

Step 2. Copy data

1. Copy all the rows from the **City** table into the **City2** table

```
mysql> INSERT INTO City2 SELECT * FROM City;
```

4079 rows are inserted.

Step 3. Delete a specific row

1. Delete the city from the **City2** table with the value for the **ID** column equal to 3803:

```
mysql> DELETE FROM City2 WHERE ID=3803;
```

2. Retrieve the row with **ID** 3803 to confirm that it has been removed:

```
mysql> SELECT * FROM City2 WHERE ID=3803;
```

Returns an empty set, indicating the row was deleted.

Step 4. Delete multiple specific rows

1. Delete all cities from the **City2** table that have the **CountryCode** 'SWE' (Sweden).

```
mysql> DELETE FROM City2 WHERE CountryCode = 'SWE';
```

15 rows are deleted.



Step 5. DELETE with ORDER BY and LIMIT

- From the **City2** table, select the rows that represent the three largest cities (according to population)

```
mysql> SELECT Population FROM City2 ORDER BY Population DESC LIMIT 3;
```

Will show a list with the three largest population numbers.

Population
10500000
9981619
9968485

Step 6. DELETE with ORDER BY and LIMIT

- Delete the row that represents the city with the largest population:

```
mysql> DELETE FROM City2 ORDER BY Population DESC LIMIT 1;
```

Step 7. Confirm

- Select the 3 largest city populations from the **City2** table again to confirm the deletion, using the same query from step 5.

Population
9981619
9968485
9696300

Shows a new list with the originally largest population number gone.



311

11.4 The UPDATE Statement

The **UPDATE** statement modifies the contents of the existing rows. Here's the general syntax of the **UPDATE** statement:

```
UPDATE <table_name>
SET column=<expression>[, column=<expression>, ..., column=<expression>]
[WHERE where_condition] [ORDER BY...][LIMIT row_count]
```

As is apparent from the syntax described above, the **UPDATE** statement requires a table name and a **SET** clause listing one or more column assignments. Optionally, a **WHERE** clause may be present to specify which rows are to be modified.

For example, if you want to modify rows in the **Country** table by increasing the **Population** of each country by 10%, you would use the following statement;

```
mysql> UPDATE Country
      ->   SET Population = Population * 1.1;
Query OK, 232 rows affected, 0 warning (#.# sec)
Rows matched: 239  Changed: 232  Warnings: 0
```

NOTE: The above statement does not change all the rows. This may be concluded from the messages returned after executing the statement. The Matched number (239) indicates the number of rows that were processed. The Changed number (232) indicates that all but 7 rows were changed. The difference is caused by the fact that 7 rows have 0 for the **Population**. As there is no net change for these rows, they are not considered to be modified.

312

The effects of column assignments made by an **UPDATE** are subject to column type constraints, just as they are for an **INSERT** or **REPLACE**. By default, if you attempt to update a column to a value that doesn't match the column definition, MySQL converts or truncates the value. If you enable strict SQL mode, the server will be more restrictive about allowing invalid values, and will usually issue an error.

It is possible for an update to have no effect. This can occur under the following conditions:

- When it matches no records for updating; due to an empty table or if no records match the **WHERE** clause.
- When it does not actually change any column values (i.e. the value given is the same as the existing value).

313

By default, **UPDATE** does not make any guarantee regarding the order in which rows are updated. This can sometimes result in problems. Suppose that the **people** table has an **id** column which is the primary key. The table contains the following rows:

```
mysql> SELECT * FROM people;
+----+-----+----+
| id | name  | age |
+----+-----+----+
| 2  | Victor | 21  |
| 3  | Susan   | 15  |
| 4  | Victor   | 31  |
+----+-----+----+
3 rows in set (#.# sec)
```



314

If you want to renumber the id values to begin at 1, you might issue this **UPDATE** statement:

```
mysql> UPDATE people SET id=id-1;
```

After the update, the table's contents are as follows:

id	name	age
2	Susan	15
1	Victor	21
3	Victor	31

The statement succeeds if it updates id values first by setting 2 to 1 and then 3 to 2. However, it fails if it first tries to set 3 to 2. In that case, the change cannot take place as it would introduce a duplicate key. In response, the primary key constraint would be violated, resulting in an error.

To solve this problem, add an **ORDER BY** clause to cause the row updates to occur in a particular order:

```
mysql> UPDATE people SET id=id-1 ORDER BY id;
```

The rows are then processed in the following order:

id	name	age
1	Victor	21
2	Susan	15
3	Victor	31

UPDATE also allows a **LIMIT** clause, which places a limit on the number of records updated. For example, if you have two identical rows in the **people** table with a name equal to 'Victor' and you want to change the name of just one of them to 'Vic', use this statement:

```
mysql> UPDATE people SET name='Vic' WHERE name='Victor' LIMIT 1;
```

NOTE: **ORDER BY** and **LIMIT** may be used together in the same **UPDATE** statement.





InLine Lab 11-C

In this exercise you will use the **UPDATE** statement to modify existing rows. This will require a mysql command line client and access to a MySQL server.

Step 1. Updating a single row and single column

1. Using the newly created **students** table, change the name 'Peter', to all upper-case 'PETER':

```
mysql> UPDATE students SET name='PETER' WHERE name='peter';
```

2. View the contents of the **students** table to confirm update:

```
mysql> SELECT * FROM students;
```

The table **students** will now contain the updated name 'PETER', and the exact time of the new entry:

name	modtime
PETER	2006-08-14 16:05:52
Tariq	2006-08-14 16:05:03

Step 2. Updating multiple rows

1. Before updating the **City2** table for this exercise, retrieve all the cities called 'San Jose':

```
mysql> SELECT * FROM City2 WHERE Name='San Jose';
```

2. Change all the cities with the name 'San Jose' by adding 2% to the current population and renaming all of the districts to 'Ocean':

```
mysql> UPDATE City2 SET Population=Population*1.02, District='Ocean'
-> WHERE Name='San Jose';
```

3. Retrieve the newly updated rows.

```
mysql> SELECT * FROM City2 WHERE Name='San Jose';
+----+-----+-----+-----+
| ID | Name | CountryCode | District | Population |
+----+-----+-----+-----+
| 584 | San José | CRI | Ocean | 345914 |
...
| 3803 | San Jose | USA | Ocean | 912842 |
+----+-----+-----+-----+
```

Returns the 4 cities named San Jose in the **City2** table, with the updated district and population.



Step 3. Using SQL_SAFE_UPDATES

1. Turn on the safe updates option before attempting the following change:

```
mysql> SET SQL_SAFE_UPDATES=1;
```

2. Set the **City2** name column to 'NYC':

```
mysql> UPDATE City2 SET Name='NYC';
```

The result of the update yields an error;

```
ERROR 1175 (HY000): You are using safe update mode and you tried to update  
a table without a WHERE that uses a KEY column
```

This would change every single city name to NYC. Without safe updates set, it will allow you to make this change.



315

11.5 The REPLACE Statement

The MySQL-specific statement **REPLACE** works like **INSERT**, except that if a **UNIQUE** or **PRIMARY KEY** constraint prevents the insertion of a duplicate row, the existing row that blocks the **INSERT** of the new row is deleted first in order to allow the **INSERT** to take place after all. The newer, duplicate row, thus replaces the older existing one.

REPLACE uses the following general syntax:

```
REPLACE INTO <table-name> (<column_list>
VALUES (<expression_list>)
```

In this example, we replace a row in the **people** table:

```
mysql> REPLACE INTO people (id, name, age)
-> VALUES(12, 'Bruce', 25);
```

NOTE: Unless the table has a **PRIMARY KEY** or a **UNIQUE** constraint, using a **REPLACE** statement makes no sense. In that case, **REPLACE** becomes equivalent to **INSERT**, because there no means to detect duplicate rows.

Values for all columns are taken from the values specified in the **REPLACE** statement. Any missing columns are set to their default values, exactly as it is the case for the **INSERT** statement. You cannot refer to values from the current row and use them in the new row.

316

The **REPLACE** statement returns a count to indicate the number of rows affected. This is the sum of the rows deleted and inserted. If the count is 1 for a single-row **REPLACE**, a row was inserted and no rows were deleted. If the count is greater than 1, one or more old rows were deleted before the new row was inserted. It is possible for a single row to replace more than one old row if the table contains multiple unique indexes and the new row duplicates values for different old rows in different unique indexes.

The affected-rows count makes it easy to determine whether **REPLACE** only added a row or whether it also replaced any rows: Check whether the count is 1 (added) or greater (replaced).

MySQL uses the following algorithm for **REPLACE**:

1. Try to insert the new row into the table
2. While the insertion fails because a duplicate-key error occurs for a primary key or unique index:
 - Delete from the table the conflicting row that has the duplicate key value
 - Try again to insert the new row into the table





InLine Lab 11-D

In this exercise you will use the **REPLACE** statement to add new data to existing **world** database tables. This will require a mysql command line client and access to a MySQL server.

Step 1. Using REPLACE

1. Using a **REPLACE** statement, add a new country to the **Country** table called with the code 'FOO' and the name 'Sakila Land'. Confirm the change by showing the data with the new code:

```
mysql> REPLACE INTO Country (code, name) VALUES ('FOO', 'Sakila Land');
```

2. Retrieve the new row to confirm:

```
mysql> SELECT * FROM Country WHERE code='FOO'\G
```

Returns the new row entry with the code 'FOO' and country named 'Sakila Land':

```
***** 1. row *****
Code: FOO
Name: Sakila Land
Continent: Asia
```

Step 2. Replacing existing rows

1. Using **REPLACE** add a country to the **Country** table with the code 'FOO' and the Continent 'Europe'.

```
mysql> REPLACE INTO Country (Code, Continent) VALUES ('FOO', 'Europe');
```

2. Examine the 'FOO' country in the table.

```
mysql> SELECT * FROM Country WHERE Code = 'FOO';
```

The old 'FOO' is replaced by the new 'FOO'. Both the name and the continent have changed:

```
***** 1. row *****
Code: FOO
Name:
Continent: Europe
```



317

11.6 INSERT with ON DUPLICATE KEY UPDATE

Apart from the **REPLACE** statement, MySQL offers another method to deal with errors occurring due to the insertion of duplicate rows.

The MySQL-specific option **ON DUPLICATE KEY UPDATE** may be used in **INSERT** statements to prevent an error from occurring whenever an attempt is made to **INSERT** a row that violates a **PRIMARY KEY** or **UNIQUE** constraint. When a **PRIMARY KEY** or **UNIQUE** constraint is violated, the presence of the **ON DUPLICATE KEY UPDATE** option will result in an attempt to update the existing row. This update of the existing row is performed instead of (and not in addition to) the original insertion of the new, offending row.

The main difference between **REPLACE** and **ON DUPLICATE KEY UPDATE** is that in case of **REPLACE**, the new row is added to the table, discarding the old row. In the case of **ON DUPLICATE KEY UPDATE**, the old row is preserved, discarding the new row.





InLine Lab 11-E

In this exercise you will build and populate a user login table, and then utilize the **ON DUPLICATE KEY UPDATE** clause to “replace” values.

Step 1. Build and populate the user login table

1. Create the table **current_users**.

```
mysql> CREATE TABLE current_users (
    ->     userid INT UNSIGNED,
    ->     username VARCHAR(100),
    ->     login_time TIMESTAMP,
    ->     visits INT UNSIGNED DEFAULT 1,
    ->     PRIMARY KEY (userid)
    -> );
Query OK, 0 rows affected (#.# sec)
```

2. Enter the following **INSERT INTO** statement to simulate the user logging in:

```
mysql> INSERT INTO current_users (userid, username)
    -> VALUES (100, 'Tobias');
Query OK, 1 rows affected (#.# sec)
```

3. Retrieve the user login row from the table. The output shows only one login (and time) for user 'Tobias':

```
mysql> SELECT * FROM current_users;
+-----+-----+-----+-----+
| userid | username | login_time           | visits |
+-----+-----+-----+-----+
|    100 | Tobias   | 2007-04-25 18:20:05 |      1 |
+-----+-----+-----+-----+
1 row in set (#.# sec)
```

Step 2. Replace the existing row utilize the **REPLACE command**

1. Perform the following SQL to simulate the user attempting to log in a second time:

```
mysql> INSERT INTO current_users (userid, username)
    -> VALUES (100, 'Tobias');
ERROR 1582 (23000): Duplicate entry '100' for key 'PRIMARY'
```

User tries to log in again using the same statement, but an error is received. This occurs due to the fact that the **INSERT** cannot be completed using the same exact values



2. The above error condition can be avoided by using a **REPLACE INTO** statement instead.:

```
mysql> REPLACE INTO current_users (userid, username)
-> VALUES (100, 'Tobias');
Query OK, 2 rows affected (#.## sec)
```

3. When you look at the table contents again after the **REPLACE INTO**, you will notice that the previous entry was replaced with the new entry (including a new time), but the number of rows has not changed.

```
mysql> SELECT * FROM current_users;
+-----+-----+-----+-----+
| userid | username | login_time          | visits |
+-----+-----+-----+-----+
|    100 | Tobias   | 2007-04-25 18:32:50 |      1 |
+-----+-----+-----+-----+
1 row in set (#.## sec)
```

Step 3. Confirm duplicate key

1. Instead of **REPLACE INTO** (which would remove their past entry), when a user logs in you can issue an **INSERT INTO** statement with a **ON DUPLICATE KEY UPDATE** clause which will add to the previous entry. Simulate the user login in again with the following SQL statement:

```
mysql> INSERT INTO current_users (userid, username)
-> VALUES (100, 'Tobias')
-> ON DUPLICATE KEY UPDATE visits=visits+1;
Query OK, 2 rows affected (#.## sec)
```

2. Review the content of the `current_users` table to see if the most current visit has been recorded using the following SQL statement:

```
mysql> SELECT * FROM current_users;
+-----+-----+-----+-----+
| userid | username | login_time          | visits |
+-----+-----+-----+-----+
|    100 | Tobias   | 2007-04-25 18:47:00 |      2 |
+-----+-----+-----+-----+
1 row in set (#.## sec)
```

This last entry is now updated and given a new number of visits, incremented by 1. If the table has multiple unique constraints, then several rows can cause the duplicate key error. In this case only one row is updated. In general, you should try to avoid using an **ON DUPLICATE KEY** clause on tables with multiple unique constraints.



11.7 The TRUNCATE TABLE Statement

Apart from the standard **DELETE** syntax, the **TRUNCATE** statement offers another, non-standard way to remove rows from a table:

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

NOTE: The word **TABLE** keyword is optional. However, we prefer to talk about this type of as a **TRUNCATE TABLE** statement to avoid confusion with the **TRUNCATE ()** function

The **DELETE** statement allows a **WHERE** clause that identifies which rows to remove, whereas **TRUNCATE TABLE** always empties the entire table. So, to remove only specific rows from a table, **TRUNCATE TABLE** cannot be used. To do that, you must issue a **DELETE** statement that includes a **WHERE** clause that identifies which records to remove:

When you omit the **WHERE** clause from a **DELETE** statement, it's logically equivalent to a **TRUNCATE TABLE** statement in its effect, but there are operational differences:

- If you need to know how many records were removed, **DELETE** returns a true row count, but **TRUNCATE TABLE** may return a count of 0.
- If a table contains an **AUTO_INCREMENT** column, emptying it completely with **TRUNCATE TABLE** may have the side effect of resetting the sequence.
- Issuing a **TRUNCATE TABLE** statement causes an implicit rollback.
- A **TRUNCATE TABLE** statement itself cannot be rolled back.

The following comparison summarizes the differences between **DELETE** and **TRUNCATE TABLE**:

DELETE:

- Can delete specific rows from a table if a **WHERE** clause is included
- Is usually slower than **DELETE**
- Returns a true row count indicating the number of removed rows
- Transactional (if applied on transactional tables)

TRUNCATE TABLE

- Cannot delete just certain rows from a table; always completely empties it
- Usually executes more quickly
- May return a row count of zero rather than the actual number of rows deleted
- Not transactional

Instructor Note: At this point in the training, the topic of Transactions and Triggers have most likely not been discussed. However, a discussion of the TRUNCATE TABLE syntax should include how these two topics act in association with this specific SQL command. We leave it up to your discretion to decide to either discuss the following issues here, or to wait for a time when the these two topics are discussed. Either way, here are some notes to assist in the discussion:

- For most storage engines, TRUNCATE TABLE is equivalent to dropping the table and then recreating it. For InnoDB, TRUNCATE TABLE is equivalent to DELETE when there are foreign key constraints that reference the table; otherwise, the table is dropped and recreated.
- Since truncation of a table does not make any use of DELETE, the TRUNCATE statement does not invoke ON DELETE triggers.
- To support the statement "May return a row count of zero rather than the actual number of rows deleted". Please refer to bug #29507 which was corrected in 5.1.28: "May return a row count of zero rather than the actual number of rows deleted".



InLine Lab 11-F

In this exercise you will use the **TRUNCATE TABLE** statement to empty tables. This will require a mysql command line client and access to a MySQL server.

Step 1. Inspect AUTO_INCREMENT

1. Inspect the next to be generated value for the **AUTO_INCREMENT** column of the **City2** table:

```
mysql> SHOW CREATE TABLE City2;
```

The DDL statement that would create the **City2** table is returned, indicating the next value for the **ID** column in the **AUTO_INCREMENT** table option.

Step 2. Empty the table

1. Empty the **City2** table:

```
mysql> TRUNCATE TABLE City2;
```

2. Confirm that all rows have been removed:

```
mysql> SELECT * FROM City2;
```

An empty set is returned indicating the table is completely empty

Step 3. Inspect the AUTO_INCREMENT

1. Inspect the **AUTO_INCREMENT** value again:

```
mysql> SHOW CREATE TABLE City2;
```

The **AUTO_INCREMENT** table option is not present in the output.

2. Issue the following to insert a single row into the **City2** table;

```
mysql> INSERT INTO City2 SELECT * FROM LIMIT 1;
```

A single row has been added to the **City2** table.

3. Inspect the **AUTO_INCREMENT** value again:

```
mysql> SHOW CREATE TABLE City2;
```

The **AUTO_INCREMENT** table option is now present and the value is 2.





Further Practice

This section contains a number of extra exercises for manipulating table data.

319

1. Create a database called **world_copy**.
2. Create the same tables in **world_copy** as in the **world** database (same structure, same contents).
3. Insert a new city into the **City** table of the new **world_copy** database with the following values for the columns **Name**, **CountryCode**, **Population** and **District** respectively: 'Sarah City', 'USA', 1, 'California'.

NOTE: Make sure to change the database to **world_copy** before doing this and the following steps.

4. Obtain the value generated for the **ID** column for the newly inserted row.
5. Use **REPLACE** instead of **INSERT** to execute the same change as for number 3 (above), with the exception of changing the city name to 'Steve City'. Does this remove the original 'Sarah City'?
6. Make sure that your SQL Mode is set to " (empty).
7. Create a table with one **TINYINT** column.
8. **INSERT** the values 'test' and 500 into the table. Verify the values.
9. Change the SQL Mode to 'TRADITIONAL'.
10. Try inserting 'test' and 500 again.
11. Make up a European country of your own and add it to the **Country** table in the **world_copy** database, using a **REPLACE** statement.
12. Change the GNP of your country to 123000.
13. Increase the population of your country by 10%.
14. Create a new table containing all European countries.
15. Delete your country from the **Country** table.



11.8 Chapter Summary

320

This chapter introduced the Manipulating Table Data in MySQL. In this chapter, you learned to:

- Insert data into a table
- Delete data from a table
- Update data in a table
- Replace data in a table
- Truncate data from a table





12 TRANSACTIONS

12.1 Learning Objectives

322

This chapter introduces using Transactions in MySQL. In this chapter, you will learn to:

- Use transaction control statement to execute multiple SQL statements concurrently
- Describe and understand the ACID properties of transactions
- Understand and use transaction isolation levels



12.2 What is a Transaction?

323

In database programming, a transaction is a collection of data manipulation execution steps that are treated as a single unit of work. That is, these execution steps are to be performed as if there were a single specialized command that accomplishes exactly that combination of actions. And yet, a computer program must execute commands sequentially. Each command can have consequences which must be handled or built upon as the steps of the transaction proceed. The following examples demonstrate a simple bank transfer from one account to another account. The first portion of the example is a non-transactional approach to the process, with each step being accomplished as it is executed. The second portion of the example is a transactional approach to the process, with the steps being grouped together and only being executed if all the steps can be performed successfully.

Non-Transactional Executions

Remove \$1000 from account #10001

Write to database

Deposit \$1000 into account #10243

Write to database

Transactional Executions

Remove \$1000 from account #10001

Deposit \$1000 into account #10243

Write to database

324

In a transaction, either all of the data manipulation steps must be carried out or action must be taken to permanently retain those operations that did succeed (or disregard those operations that did succeed). If there was a problem with the deposit of the funds (the second execution), the transactional approach should terminate before any permanent action takes place against the data. However, the non-transactional approach does not have this option and would still remove the funds from the first account (would perform the action against the data) and terminate on the second action, causing the process to be half completed.

Non-Transactional Executions

Remove \$1000 from account #10001

Write to database

Deposit \$1000 into account #10243

Transactional Executions

Remove \$1000 from account #10001

Deposit \$1000 into account #10243



InnoDB storage engine

The way that transactions are implemented with the InnoDB storage engine, a transaction could also be considered to be a test environment within which the viability of each data manipulation step is assessed, in the proper order and without affecting the database, until the database is assured that all steps can be performed successfully. The results of these tests can then be used to update the database quickly without performing them again.



12.2.1 ACID

325

Transactional systems often are described as being ACID compliant, where “ACID” stands for the following properties:

- **Atomic** - All the statements execute successfully or are canceled as a unit.
- **Consistent** - A database is in a consistent state when a transaction begins is left in a consistent state by the transaction.
- **Isolated** - One transaction does not affect another.
- **Durable** - All the changes made by a transaction that completes successfully are recorded properly in the database. Changes are not lost.

Transactional processing provides stronger guarantees about the outcome of the database operations, but also requires more overhead in CPU cycles, memory, and disk space. Transactional properties are essential for some applications and not for others, and you can choose which ones make the most sense for your applications.

Financial operations typically need transactions, and the guarantees of data integrity outweigh the cost of additional overhead. On the other hand, for an application that logs web page accesses to a database table, a loss of a few records if the server host crashes might be tolerable.

MySQL offers some storage engines that support transactions and some that do not. This allows you to choose for the engine that best matches your application requirements. Storage engines will be covered later in the course.



12.3 Transaction Control Statements

326

The following statements are used to explicitly control transactions:

- **START TRANSACTION** or **BEGIN** - Explicitly begins 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

NOTE: In MySQL, transactions are supported only for those tables that use a transactional storage engine (like InnoDB). These statements will have no noticeable effect on tables managed by a non-transactional storage engine.

12.3.1 The autocommit mode

327

The autocommit mode determines how and when new transactions are started.

Autocommit enabled

If autocommit is enabled, a single SQL statement implicitly starts a new transaction by default. The transaction is automatically committed if the statement executes successfully, permanently saving any changes caused by executing the statement. If the statement does not execute successfully, the transaction is automatically rolled back, undoing any changes that would have been the result of executing the statement.

Autocommit being mistaken

Enabling autocommit wraps each statement affecting transactional tables in its own transaction. This effectively prevents one from doing multiple statements inside one transaction. This means that the ability to **COMMIT** or **ROLLBACK** multiple statements as unit is lost. Sometimes, this is mistaken with not having transactions at all. However, this is not the case: when autocommit is enabled, each statement is still executed atomically.

The difference between enabling autocommit and not having transactions at all can be easily demonstrated when comparing the effect of a constraint violation while inserting multiple rows. On a non-transactional table, like MyISAM, the statement terminates as soon as the error is encountered, leaving the rows inserted so far in the table. For an InnoDB table, all rows inserted so far are withdrawn from the table, causing no net effect at all.

When the autocommit mode is enabled, transactions can still be started explicitly using the **START TRANSACTION** statement. In this case, the transaction can span multiple statements. Such an explicit transaction will not commit automatically after statement execution and can thus span multiple statements. When autocommit is not enabled, these transactions are the default. They are discussed in more detail in the next section.

Autocommit disabled

328

If autocommit is not enabled, transactions span multiple statements by default. In this case, the transaction can be explicitly committed or rolled back using the **COMMIT** and **ROLLBACK** statements respectively. After the transaction terminates, a new transaction is implicitly started.



When a statement that is executed within the transaction does not complete successfully, any changes that would have been caused by that statement are undone, but the transaction as a whole does not terminate. The transaction remains open in this case, and can still be either committed or rolled back as a whole.

329

Controlling the autocommit mode

The autocommit mode can be controlled through the server variable **AUTOCOMMIT**. The variable is available on the session level. The following statement would disable autocommit for the current session:

```
mysql> SET AUTOCOMMIT = OFF;
```

This is equivalent to:

```
mysql> SET SESSION AUTOCOMMIT = OFF;
```

Alternatively, the variable notation may be used, in which case the values 1 and 0 must be used to respectively enable or disable the autocommit mode:

```
mysql> SET @@autocommit := 0;
```

The variable notation may also be used to find out whether autocommit is currently enabled or disabled:

```
mysql> SELECT @@autocommit;
+-----+
| @@autocommit |
+-----+
|          0   |
+-----+
```

330

Changing the default autocommit setting

By default, autocommit is enabled. This means that autocommit should be disabled when transactions that span multiple statements are required. Like described in the previous section, this can be achieved by issuing a **SET** statement. Alternatively the server configuration can be changed to always automatically disable autocommit.

To ensure that autocommit is disabled by default you can start the server with the **init-connect** option containing a statement to disable autocommit:

```
init-connect="SET AUTOCOMMIT := OFF"
```

By adding a line like this to the `my.cnf` or `my.ini` option file, new connections will first execute this statement, disabling autocommit. However, this solution does not work for users with the **SUPER** privilege (including `root`).



12.3.2 Statements causing an Implicit COMMIT

331

The **COMMIT** statement always explicitly commits the current transaction. Other transaction control statements such as **START TRANSACTION** and **SET AUTOCOMMIT:=1** also have the effect of implicitly committing the current transaction.

Apart from these transaction control statements, other types of statements may have the effect of implicitly committing (and thereby terminating) the current transaction. These statements have the same effect as if a **COMMIT** would be issued prior to executing the actual statement. In addition, these statements are themselves non-transactional, meaning they cannot be rolled back if they succeed.

Generally speaking the data definition statements (**ALTER**, **CREATE**, **DROP**), data access and user management statements (**GRANT**, **REVOKE**, **SET PASSWORD**) and locking statements (**LOCK TABLES**, **UNLOCK TABLES**) have this effect. (There are a number of exceptions, and not all of these statements cause an implicit commit in all versions of the server, but it is advisable to treat all non-DML statements as if they cause an implicit commit).

There are also a few DML statements that cause an implicit commit: **TRUNCATE TABLE**, **LOAD DATA INFILE**.

Transaction Demo1: ROLLBACK

332

The following session demonstrates the effect of the **ROLLBACK** statement:

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (#.## sec)
mysql> SELECT Name FROM City WHERE ID=3803;
+-----+
| Name   |
+-----+
| San Jose |
+-----+
1 row in set (#.## sec)
mysql> DELETE FROM CITY WHERE ID=3803;
Query OK, 1 row affected (#.## sec)
mysql> SELECT Name FROM City WHERE ID=3803;
Empty set (#.## sec)
mysql> ROLLBACK;
Query OK, 0 rows affected (#.## sec)
mysql> SELECT Name FROM City WHERE ID=3803;
+-----+
| Name   |
+-----+
| San Jose |
+-----+
1 row in set (#.## sec)
```



333

12.3.3 Finding a Storage Engine that supports transactions

To make sure that you have a transactional storage engine compiled into your server, and whether it is available at runtime, use the **SHOW ENGINES** statement:

```
mysql> SHOW ENGINES\G
***** 1. row *****
Engine: MyISAM
Support: YES
Comment: Default engine as of MySQL 3.23 with great performance
***** 2. row *****
Engine: MEMORY
Support: YES
Comment: Hash based, stored in memory, useful for temporary tables
***** 3. row *****
Engine: InnoDB
Support: DEFAULT
Comment: Supports transactions, row-level locking, and foreign keys
```

The value in the Support column is YES or NO to indicate that the engine is or is not available. If the value is DISABLED then the engine is present but turned off. The value DEFAULT indicates the storage engine that the server uses by default. (The engine designated as DEFAULT should be considered available).





InLine Lab 12-A

In this exercise you will use the transaction control statements. This will require a mysql command line client and access to a MySQL server.

Step 1. Find a storage engine

1. Determine if any transactional storage engines are available on your system, and which is the default engine:

```
mysql> SHOW ENGINES\G
```

Returns a list of all available storage engines. The DEFAULT engine is indicated in the **Support** column.

Step 2. Verify autocommit mode

1. Check the current setting for **AUTOCOMMIT**:

```
mysql> SELECT @@AUTOCOMMIT;
```

Returns the number 1 (for on), or 0 (for off).

Step 3. Enabling Autocommit

1. Set the **AUTOCOMMIT** mode to on:

```
mysql> SET AUTOCOMMIT=1;
```

2. Check the value to confirm.

```
mysql> SELECT AUTOCOMMIT;
```

Shows **OK**. Check confirms on condition by returning the number 1.

Step 4. Using the innodb engine

1. Execute the following statements;

```
mysql> SHOW CREATE TABLE City;
mysql> ALTER TABLE City ENGINE=InnoDB;
mysql> SHOW CREATE TABLE City;
```

The table now uses the InnoDB engine which supports transactions

Step 5. Using explicit transactions

1. Explicitly start a new transaction

```
mysql> START TRANSACTION;
```

Shows **OK**.



Step 6. Delete a row

1. Select the row that contains the city named 'Manta':

```
mysql> SELECT * FROM City WHERE Name='Manta';
```

2. Delete this row associated with 'Manta':

```
mysql> DELETE FROM City WHERE Name='Manta';
```

1. Confirm that the row is now gone;

```
mysql> SELECT * FROM City WHERE Name='Manta';
```

SELECT confirms the existence of the city of Manta. The delete returns **OK**, indicating that 1 row was affected. The second **SELECT** shows that the row is now empty.

Step 7. Using ROLLBACK

1. Roll back the open transaction:

```
mysql> ROLLBACK;
```

2. Confirm that the row is now back to its' original state.

```
mysql> SELECT * FROM City WHERE Name='Manta';
```

The cancellation is confirmed with an **OK**. The result shows that the row is now back.

Step 8. Another transaction

1. Start a new transaction:

```
mysql> START TRANSACTION;
```

2. Again, delete the row associated with 'Manta'.

```
mysql> DELETE FROM City WHERE Name='Manta';
```

The delete returns **OK**.



Step 9. Using COMMIT

1. Commit the transaction:

```
mysql> COMMIT;
```

2. Confirm that the row is now gone.

```
mysql> SELECT * FROM City WHERE Name='Manta';
```

Shows **OK**. The final **SELECT** shows that the row does not exist anymore, and that this delete cannot be undone.



334

12.4 Isolation Levels

As mentioned earlier, multiple pending transactions may be exist simultaneously within the server (at most one transaction per session). This has the potential to cause problems: If one client's transaction changes data, should transactions for other clients see those changes or should they be isolated from them?

The transaction isolation level determines the level of visibility between transactions—that is, the ways in which simultaneous transactions interact when accessing the same data. This section discusses the problems that can occur and how transactional storage engines (like InnoDB) implement isolation levels. Note that isolation level choices vary among database servers, so the levels as implemented by InnoDB might not correspond exactly to levels as implemented in other database systems.

12.4.1 Consistency issues

When multiple clients are accessing data from the same table concurrently, the following consistency issues can occur:

- Dirty reads
- Non-repeatable reads
- Phantom reads

Dirty reads

A dirty read occurs when a transaction reads the changes made by another uncommitted transaction.

Suppose that transaction T1 modifies a row. If transaction T2 reads the row and sees the modification even though T1 has not committed it, that is a dirty read. One reason this is a problem is that if T1 rolls back, the change is undone but T2 does not know that.

Non-repeatable reads

A non-repeatable read occurs when the same read operation yields different results when it is repeated at a later time within the same transaction.

NOTE: If one and the same transaction causes the modifications leading to different results for an identical read operation, it is not considered to be a non-repeatable read. A non-repeatable read occurs when another transaction commits changes causing the read operation to be non-repeatable

Suppose that a transaction T1 reads some rows. Then, after that, a second transaction T2 changes some of those rows and commits the changes. If T1 is able to see the changes made in transaction T2 when retrieving the rows again; the initial read proved to be non-repeatable. This is a problem because T1 does not get a consistent result from the same query.

Phantom reads

A phantom is a row that appears that was not visible before within the same transaction. Suppose that transactions T1 and T2 begin, and T1 reads some rows. If T2 inserts a new row and T1 sees that row when it repeats the same read operation, a phantom read has occurred (the new row being the phantom row).



335

12.4.2 Four Levels

InnoDB implements four isolation levels that control the extent to which changes made by transactions are noticeable to other simultaneously occurring transactions:

- **READ UNCOMMITTED** allows a transaction to see uncommitted changes made by other transactions. This isolation level allows dirty reads, non-repeatable reads, and phantom reads to occur.
- **READ COMMITTED** allows a transaction to see changes made by other transactions only if they've been committed. Uncommitted changes remain invisible. This isolation level allows non-repeatable reads and phantoms to occur.
- **REPEATABLE READ** ensures that if a transaction issues the same **SELECT** twice, it gets the same result both times, regardless of committed or uncommitted changes made by other transactions. In other words, it gets a consistent result from different executions of the same query. In some database systems, **REPEATABLE READ** isolation level allows phantoms, such that if another transaction inserts new rows in the interval between the **SELECT** statements, the second **SELECT** will see them. This is not true for InnoDB; phantoms do not occur for the **REPEATABLE READ** level.
- **SERIALIZABLE** completely isolates the effects of one transaction from others. It is similar to **REPEATABLE READ** with the additional restriction that rows selected by one transaction cannot be changed by another until the first transaction finishes.

The essential difference between **REPEATABLE READ** and **SERIALIZABLE** is that with **REPEATABLE READ**, one transaction cannot modify rows another has modified, whereas with **SERIALIZABLE**, one transaction cannot modify rows if another has merely even read them.

Isolation levels are relevant only within the context of simultaneously executing transactions. After a given transaction has committed, its changes become visible to any transaction that begins after that.

With multi-versioning, each transaction sees a view of the contents of the database that is appropriate for its isolation level. For example, with a level of **REPEATABLE READ**, the snapshot of the database that a transaction sees is the state of the database at its first read. One property of this isolation level is that it provides consistent reads: A given **SELECT** yields the same results when issued at different times during a transaction. The only changes the transaction sees are those it makes itself, not those made by other transactions. For **READ COMMITTED**, on the other hand, the behavior is slightly different. The view of the database that the transaction sees is updated at each read to take account of commits that have been made by other transactions since the previous read.

336

Isolation level	dirty reads	non-repeatable reads	phantom reads
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not possible	Possible	Possible
REPEATABLE READ	Not possible	Not possible	Not Possible for InnoDB
SERIALIZABLE	Not possible	Not possible	Not possible



To set the server's default transaction isolation level at startup time, use the `transaction-isolation` option. The option value should be either one of READ-UNCOMMITTED, READ-COMMITTED, REPEATABLE-READ, or SERIALIZABLE. For example, to put the server in READ COMMITTED mode by default, put these lines in an option file:

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

The isolation level may also be set dynamically for a running server using a `SET TRANSACTION ISOLATION LEVEL` statement. The syntax model is:

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL <isolation-level>
```

where `<isolation-level>` is defined as:

```
{READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}
```

The transaction level can be set either at the global or at the session level. Without an explicit specification, the transaction isolation level is set at the session level. For example the following statement will set the isolation level to `READ COMMITTED` for the current session:

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

This is equivalent to:

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

To set the default level for all subsequent connections, use the `GLOBAL` keyword instead of `SESSION`:

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

NOTE: Setting the default transaction isolation level globally applies to all new client connections established from that point on. Existing connections are unaffected.



337

Any client can always modify the transaction isolation level for its own session, but changing the default transaction isolation level globally requires the **SUPER** privilege.

To find out what the current isolation level is, use the `tx_isolation` server variable:

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

When used unprefixed, the session transaction isolation level is returned. Use the global and session prefixes to explicitly obtain the global or session isolation level respectively:

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

NOTE: The server variable may also be used to set the transaction isolation level. The same isolation levels are valid as in the `SET TRANSACTION ISOLATION LEVEL` syntax, except that it must be a string representation (rather than plain keywords) and that the words that define an isolation level must be separated by a single hyphen (dash or minus sign) rather than whitespace.



338

Transaction Demo2: Concurrency and Isolation

The procedure below shows the effect of transaction isolation for two transactions occurring on two different simultaneous mysql sessions:

NOTE: This example assumes that all tables are using the InnoDB storage engine and that autocommit is enabled for both sessions.

Session #1	Session #2
<pre>mysql> PROMPT s1></pre>	
<pre>s1> SET GLOBAL TRANSACTION -> ISOLATION LEVEL READ -> COMMITTED;</pre>	
<pre>s1> SELECT @@tx_isolation;</pre>	
<p>gives:</p> <pre>+-----+ @@tx_isolation +-----+ READ-COMMITTED +-----+</pre>	
	<pre>mysql> PROMPT s2></pre>
	<pre>s2> START TRANSACTION;</pre>
	<pre>s2> INSERT INTO City -> (Name, CountryCode, -> population) -> VALUES ('Sakila', 'SWE', 1);</pre>



	Session #1	Session #2
339	<pre>s1> SELECT Name, CountryCode -> FROM City -> WHERE Name = 'Sakila';</pre> <p>Gives an empty set as the current isolation level prevents this transaction from seeing uncommitted changes</p>	
		<pre>s2> COMMIT;</pre>
	<pre>s1> SELECT Name, CountryCode -> FROM City -> WHERE Name = 'Sakila';</pre> <p>now gives:</p> <pre>+-----+-----+ Name CountryCode +-----+-----+ Sakila SWE +-----+-----+</pre>	
	<pre>s1> PROMPT</pre>	<pre>s2> PROMPT</pre>



12.5 Locking

12.5.1 Locking Concepts

340

Locking is a mechanism that prevents problems from occurring with simultaneous data access by multiple clients. Locks are managed by the server: It places a lock on data on behalf of one client to restrict access by other clients to the data until the lock has been released. The lock allows access to data by the client that holds the lock, but places limitations on what operations can be done by other clients that are contending for access. The effect of the locking mechanism is to serialize access to data so that when multiple clients want to perform conflicting operations, each must wait its turn.

Not all types of concurrent access produce conflicts, so the type of locking that is necessary to allow a client access to data depends on whether the client wants to read or write:

- If a client wants to read data, other clients that want to read the same data do not produce 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.
- If a client wants to write data, all other clients must wait until the write has finished, regardless of whether those clients want to read or write.

In other words, a reader must block writers, but not other readers. A writer must block both readers and writers. Read locks and write locks allow these restrictions to be enforced. Locking makes clients wait for access until it is safe for them to proceed. In this way, locks prevent data corruption by disallowing concurrent conflicting changes and reading of data that is in the process of being changed.

341

12.5.2 Locking Reads

InnoDB supports two locking modifiers that may be added to the end of **SELECT** statements:

The **LOCK IN SHARE MODE** clause locks each selected row with a shared lock. It is a shared lock, this means that no other transactions can take exclusive locks but other transactions can also use shared locks. As normal reads do not lock anything they aren't affected by the locks.

The **FOR UPDATE** clause locks each selected row with an exclusive lock, preventing others from acquiring any lock on the rows, but allows reading the rows.

In the **REPEATABLE READ** isolation level, you can add **LOCK IN SHARE MODE** to **SELECT** operations to force other transactions to wait for your transaction if they want to modify the selected rows. This is similar to operating at the **SERIALIZABLE** isolation level, for which InnoDB implicitly adds **LOCK IN SHARE MODE** to **SELECT** statements that have no explicit locking modifier. If the **SELECT** will select rows that have been modified in an uncommitted transaction, it will lock the **SELECT** until that transaction commits.



LOCK IN SHARE MODE:

For example, you might want to add a new row into the `City` table, and make sure that the child row has a parent in table `Country`. The following example shows how to implement referential integrity in your application code.

Suppose that you want to add a new city to the country of Australia in the `City` table. To confirm that the parent (Australia) exists in the child table, you could select the corresponding country code:

```
mysql> SELECT * FROM City WHERE CountryCode='AUS';
+----+-----+-----+-----+
| ID | Name      | CountryCode | District       | Population |
+----+-----+-----+-----+
| 130 | Sydney    | AUS         | New South Wales | 3276207   |
| 131 | Melbourne | AUS         | Victoria        | 2865329   |
| 132 | Brisbane  | AUS         | Queensland     | 1291117   |
| 133 | Perth     | AUS         | West Australia | 1096829   |
...

```

Can you safely add the child row to table `City`? No, because some other user may simultaneously delete the parent row from the table `Country` without you being aware of it.

The solution is to perform the `SELECT` in a locking mode using `LOCK IN SHARE MODE`:

```
mysql> SELECT * FROM Country WHERE Code='AUS' LOCK IN SHARE MODE\G
***** 1. row *****
Code: AUS
Name: Australia
Continent: Oceania
Region: Australia and New Zealand

```

Performing a read in with `LOCK IN SHARE MODE` means that we read the latest available data, and set a shared mode lock on the rows we read. After we see that the preceding query returns the parent ‘AUS’, we can safely add the child record to the `City` table and commit our transaction.



FOR UPDATE:

We have an integer counter field in a table `child_codes` that we use to assign a unique identifier to each child added to table `child`. Using a consistent read or a shared mode read to read the present value of the counter is not good because two users of the database may then see the same value for the counter, and a duplicate-key error occurs if two users attempt to add children with the same identifier to the table.

In this case, one way to implement the reading and incrementing of the counter is to first update the counter by incrementing it by 1:

```
mysql> SELECT counter_field INTO @@counter_field FROM child_codes
-> FOR UPDATE;
```

.... and then read:

```
mysql> UPDATE child_codes SET counter_field = @@counter_field + 1;
```

342

Transaction Demo 3: Deadlock

Session #1	Session #2
<pre>mysql> PROMPT s1> s1> START TRANSACTION;</pre>	
<p>and then</p> <pre>s1> UPDATE Country -> SET name='Sakila' -> WHERE code='SWE';</pre>	<pre>mysql> PROMPT s2> s2> START TRANSACTION;</pre> <p>and then</p> <pre>s2> UPDATE Country SET -> name='World Cup Winner' -> WHERE code='ITA';</pre>



Session #1	Session #2
<pre>s1> DELETE FROM Country -> WHERE code='ITA';</pre> <p>This will hang, waiting for lock</p>	
	<pre>s2> UPDATE Country -> SET population=1 -> WHERE code='SWE';</pre> <p>deadlock detected:</p> <pre>ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction</pre>
DELETE stops hanging and executes successfully: <pre>Query OK, 1 row affected (##.## sec)</pre>	
<pre>s1> PROMPT</pre>	<pre>s2> PROMPT</pre>





Quiz

In this exercise you will answer questions pertaining to Transactions.

1. What is a transaction?

2. What are the implications of being in non-autocommit mode?

3. What does the acronym ACID stand for?

4. Name the four transaction isolation levels

5. Which three types of inconsistencies can occur in case two simultaneous transactions access the same table?

6. A Serializable level allows non-repeatable reads. (True or False)

7. The two locking modifiers that InnoDB supports are:

8. Both locking types will lock selected _____ to prevent problems that may occur when there is simultaneous access of table data.





343

Further Practice

In this exercise, you will exercise working with transactions using the `world` database.

1. Make sure the Storage Engine of the City table is InnoDB.
2. Start a new transaction: How many rows does the City table have? Delete all rows of the City table.
3. If the transaction is rolled back, how many rows will the table have? Roll back the transaction and see.
4. Start a new transaction (t1).
5. In t1, SELECT all rows from the City table where the id > 4070.
6. Open up another client connection to the MySQL Server and start another transaction (t2).
7. In t2, SELECT all rows from the City table where the id > 4070.
8. In t2, INSERT a new row to the City table.
9. Go back to t1 and issue the SELECT again.
10. What do you see? Which isolation levels cannot you be using?
11. Go to t2 and COMMIT the transaction.
12. Return to t1 and issue the select again.
13. Do you now see the row? What isolation level are you running?
14. ROLLBACK t1 and issue the SELECT again, is the row now visible?
15. Repeat steps 4-14 but change the isolation level of t1 to Repeatable Read before you start the transaction. When can you now see the row?
16. Repeat steps 4-14 again, now changing the isolation level of t1 to Read Committed. When is the row visible now?



344

12.6 Chapter Summary

This chapter introduced using Transactions in MySQL. In this chapter, you learned to:

- Use transaction commands to run multiple SQL statements concurrently
- Describe and use the ACID transaction rules
- Isolate one transaction from another



13 JOINS

13.1 Learning Objectives

346

This chapter describes using the join operation to use multiple tables in a single query. In this chapter, the following things will be covered:

- The concept of joining tables
- Construction and properties of the Cartesian product
- The syntax and application of different join types.
- Using qualified column references and table aliases to avoid ambiguity
- To join a table to itself
- Multi-table **UPDATE** and **DELETE** statements



347

13.2 What is a Join?

The SQL statements described earlier in this training guide were always based on a single table. However, not all questions can be answered using just one table. When it is necessary to draw on data that is stored in multiple tables, you may use a *join*. A *join* is an operation on two tables that combines rows from multiple tables into new rows, thereby constructing a single new result table.

13.2.1 The limitation of single table queries

348

Before explaining exactly what a join is, it is necessary to understand why it may be necessary to work with multiple tables in a single query. Consider the **Country** and the **City** tables in the **world** database. The **City** table contains rows that represent cities. The following query returns all rows representing the cities called 'London':

```
mysql> SELECT * FROM City WHERE Name = 'London';
+----+-----+-----+-----+
| ID | Name | CountryCode | District | Population |
+----+-----+-----+-----+
| 456 | London | GBR           | England   | 7285000 |
| 1820 | London | CAN           | Ontario   | 339917  |
+----+-----+-----+-----+
```

Because only the **City** table was used in the query, only data about cities can be retrieved. It could be argued that the **CountryCode** column in the result set also provides some data from the respective country wherein these cities reside. However, although it can be derived which countries correspond to the codes 'GBR' and 'CAN', there aren't any columns that give any details about the countries themselves.

To obtain data concerning countries, it is necessary to look in the **Country** table. So, it would be necessary to do a separate query to retrieve some data from the countries with the country codes 'GBR' and 'CAN':

```
mysql> SELECT Code, Name, Continent, Population
    -> FROM Country WHERE Code IN ('GBR', 'CAN');
+----+-----+-----+-----+
| Code | Name          | Continent     | Population |
+----+-----+-----+-----+
| CAN  | Canada        | North America | 34261700 |
| GBR  | United Kingdom | Europe       | 65585740 |
+----+-----+-----+-----+
```

Although in the end, the data about both cities and countries was obtained, it was accomplished using two separate queries. Like the first query, the second query only uses one table. In this case, it used the **Country** table, and the result thus contains only data about countries, but not about cities.

With this approach, it is never possible to obtain a result that contains rows that combine the data from both tables. For example, if it is necessary to obtain a list of city names along with the name of the country wherein that city resides, it would be necessary to have rows that contain the **Name** column from the **City** table as well as the **Name** column from the **Country** table. To obtain such a result, a query must be written that draws data from both the **City** as well as the **Country** table to create rows in the result using columns from both tables.



SQL offers a number of powerful operations to combine rows from multiple tables in this manner. All these operations are some special case of a *join operation*. By joining two tables, rows are combined from multiple tables and can choose columns from all of the joined tables to make new composite rows, combining the columns from multiple tables.

13.2.2 Combining two simple tables

Before discussing the exact SQL syntax to join tables, let's first consider how exactly the **Country** and **City** table can be combined. To not make things overly complicated, lets create a few simplified versions of the **City** and **Country** tables.

349

The following statement creates a simplified version of the **City** table containing only the rows for the cities called 'London' and only the columns **Name**, **CountryCode**, and **Population**

```
mysql> CREATE TABLE SimpleCity AS
->   SELECT Name AS CityName, CountryCode, Population AS CityPop
->   FROM City WHERE Name Like 'London';
```

And the contents are:

```
mysql> SELECT * FROM SimpleCity;
+-----+-----+-----+
| CityName | CountryCode | CityPop |
+-----+-----+-----+
| London   | GBR        | 7285000 |
| London   | CAN        | 339917  |
+-----+-----+-----+
```

Likewise, the following statement creates a simplified **Country** table containing only the rows for the country codes 'GBR' and 'CAN' and only the columns **Code**, **Name**, and **Population**:

```
mysql> CREATE TABLE SimpleCountry AS
->   SELECT Code, Name AS CountryName, Population AS CountryPop
->   FROM Country WHERE Code IN ('CAN', 'GBR');
```

with the contents:

```
mysql> SELECT * FROM SimpleCountry;
+-----+-----+-----+
| Code | CountryName | CountryPop |
+-----+-----+-----+
| CAN  | Canada      | 34261700 |
| GBR  | United Kingdom | 65585740 |
+-----+-----+-----+
```

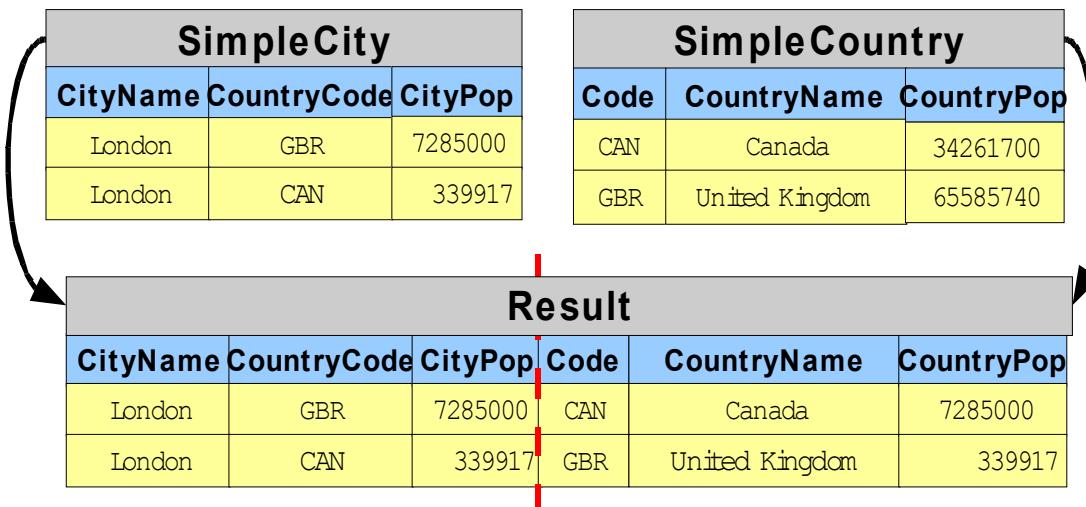


Let's consider what would happen if we would combine the contents of both tables. Remember, the goal is to obtain a result that shows the city names along with the country names so we have to somehow combine the rows as well as the columns from both tables.

350

Glueing columns together

Combining means: to somehow make one thing out of multiple parts. So, the first thing to consider is this; is there somehow to 'glue' the columns from the **SimpleCountry** table to the **SimpleCity** table:



Well, at least now the desired combination of the columns **CityName** and **CountryName** (along with all the other columns from both tables) is achieved. However, it is also clear that this is not exactly correct. The 'GBR' row from the **SimpleCity** half of the table is combined with the 'CAN' row of the **SimpleCountry** half of the table. It could have been the other way around too, as the rows have no intrinsic order.

So, although 'glueing together' the tables could yield the desired combination of columns, there is something wrong about the way the rows are combined. This could have been realized earlier if the **SimpleCity** and **SimpleCountry** table would've had a different number of rows.

Relational databases prevent mismatch of rows

If you imagine that the **SimpleCity** would have had more rows than the **SimpleCountry** table, then not all the rows in the result table would have been complete. Some rows would have had values for all columns from both the **SimpleCity** as well as the **SimpleCountry** table, whereas there would also be a number of rows with only columns for the **SimpleCity** half of the result table. This 'skewed' result set is impossible in a relational database, as the result would not be a proper table.



13.2.3 Cartesian product

351

So how can we combine the rows from the **SimpleCity** and **SimpleCountry** tables in a way that would result in a properly formed table? In the previous example we tried to glue the *columns* of two different tables together. Now, let's try and see what happens if we try to glue the *rows* together instead.

SimpleCity		
CityName	CountryCode	CityPop
London	GBR	7285000
London	CAN	339917

SimpleCountry		
Code	CountryName	CountryPop
CAN	Canada	34261700
GBR	United Kingdom	65585740

Cartesian Product					
CityName	CountryCode	CityPop	Code	CountryName	CountryPop
London	GBR	7285000	CAN	Canada	34261700
London	GBR	7285000	GBR	United Kingdom	65585740
London	CAN	339917	CAN	Canada	34261700
London	CAN	339917	GBR	United Kingdom	65585740

First, let us create *all* possible combinations of rows by combining each individual row from the **SimpleCity** table with all of the rows from the **SimpleCountry** table. As an aid in understanding how two tables can be combined this way, consider the following pseudocode:

```

352      goto_first_row(SimpleCity)
353      while has_rows(SimpleCity) do
354          goto_first_row(SimpleCountry)
            while has_rows(SimpleCountry) do
                new_row= current_row(SimpleCity) + current_row(SimpleCountry)
                add_row(Result, new_row)

                goto_next_row(SimpleCountry)
            end while

            goto_next_row(SimpleCity)
        end while
    
```

352

353

354



Instructor's notes: The Cartesian product is covered in a whole series of slides that show a gradual build up of the result. These slides are meant as a sort of animation – it is not the intention to cover each slide in detail. Rather, the sequence of slides should be scrolled through quite quickly. The intention of the animation is to give attendees a feel for the actual number of operations involved in constructing even a relatively simple Cartesian product

Here, a loop runs through all the rows in the **SimpleCity** table, and for each row from **SimpleCity**, a new loop is started to visit every row in **SimpleCountry**. Then, inside that second loop, the current row from the **SimpleCountry** table is appended to the current row from the **SimpleCity** table. This newly generated composite row is then stored in the imaginary table called “**Result**” which represents the table that results from combining all rows.

Pseudocode purpose

This pseudocode is just an illustration. We do not mean to imply at this point that any RDBMS product uses code like this to combine rows. The purpose of this example is to understand the exact outcome of the process, not to learn how to practically implement it.

The point of this particular pseudocode example is that if some software would use this kind of code, it would in fact have the result of producing all possible combinations of rows.

There will be a number of pseudocode examples in the remainder of this chapter, and none of them relates to any practical implementation or algorithm. The pseudo code listings are always meant as an aid in understanding the result of a particular operation.

355

356

SimpleCity

357

SimpleCountry

358

CityName	CountryCode	CityPop	Code	CountryName	CountryPop
London	GBR	7285000	CAN	Canada	34261700
London	GBR	7285000	GBR	United Kingdom	65585740
London	CAN	339917	CAN	Canada	34261700
London	CAN	339917	GBR	United Kingdom	65585740

359

360

361

The result of this operation would look something like the table shown above. The new table combines all of the columns as well as all of the rows of both tables. Such a result is called the *product* or *cross-product* and is more formally known as the *Cartesian product* (after the French philosopher and mathematician René Descartes).

362

Among the rows of the resulting table, we can already see a few composite rows that list the city names along with their respective country. This is not surprising, as the product operation ensures that we have *all* possible combinations of rows from the underlying tables. The result rows that combine the rows from the **SimpleCity** table with only the corresponding rows from the **SimpleCountry** table *must* be somewhere among the collection of *all* possible combinations of the rows. Another way of putting it is to say that the collection of all rows contains all combinations of corresponding rows as well as all possible combinations of non-corresponding rows.

So, apart from the properly paired rows, we also have a lot of undesirable combinations, but this time we at least have the desired result rows (we did not in our previous example). We will show later in this section how we can discard the undesired rows to yield the final result. However, before we do that we first explore the concept of the Cartesian product in a bit more detail.



13.2.4 General properties of the Cartesian product

The Cartesian product is an essential concept in understanding joins. Therefore it is important to understand a few of the general properties of Cartesian products.

Dimension of the Cartesian product

363

If a Cartesian product is to be created out of 2 tables, we can calculate the dimensions of the result as such:

- The product will have all the columns from both tables, so the total number of columns of the product is the sum of the number of columns of the tables that take part in the product. So, **SimpleCity** has 3 columns, and so has **SimpleCountry**. That's why our product table has $3 + 3 = 6$ columns.
- The product combines each row from one of the tables with all of the rows from the other table, producing every conceivable pair of rows. The resulting number of rows in the product can therefore be calculated by multiplying the number of rows in both tables. So, **SimpleCity** has 2 rows, and **SimpleCountry** has also 2 rows, bringing the total number of rows for the product table to $2 * 2 = 4$ rows.

Order of the tables

364

An important thing to realize about the product operation is that the order of the tables that take part in the product is not of any real consequence from a relational point of view. Although the order of the rows and the order of the columns in the product table is influenced by the order in which we process the tables that take part in the product, this does not really change the information contained in the result table. From a relational point of view, the order of the rows is not significant anyway, and the order of the columns is usually not important as long as we can still identify each individual column. Whether we combine **Country** rows with **City** rows or **City** rows with **Country** rows does not change the meaning of the result; in both cases we end up with a row that combines the data from both tables.

The Cartesian product is yet another table

365

The Cartesian product is itself a table. That is, the product is not a 'real', physical table, but it certainly is a collection of rows structured according to a single collection of columns. For all intents and purposes, the result is conceptually yet another table.

The ability to treat the result of an operation on two tables as another table is very important from a theoretical point of view: it allows a simple language (a calculus) to be built to work with tables. This is somewhat comparable to what we see in regular arithmetic: for example, when we multiply two numbers, the result is again a number, and the same goes for all the other arithmetical operators. This allows us to apply operations like multiplication, addition, division, and subtraction not only to bare numbers, but also to entire multiplications, additions, subtractions, divisions and so on.

Cartesian products of more than two tables

366

What would it look like if we would obtain a Cartesian product of more than two tables? First of all, it's important to realize that we do not literally need to directly calculate the product of more than two tables. We can calculate the product of two tables first, and then calculate the product of the result table with the another table, and so on and so forth. Therefore we will regard the product operation as an operation that processes two tables, resulting in one new table.



Once we realize that the intermediate result of the product of any two tables is yet another table that can take part in yet another product, we can logically skip the intermediate step and consider the product operation as if it processes an arbitrary number of tables all at once. In that sense we can talk about the product of an arbitrary number of tables.

Going deeper in the “nest”

We can physically skip the intermediate result and devise a computer program that literally combines the contents of all tables on a row by row basis, without building an intermediate result that repeatedly takes part in subsequent product operations. Think of the pseudocode example to calculate the product of two tables. All we need to do is to add another nested loop inside the inmost loop. We can repeat this process and theoretically we can keep on adding even deeper nested loops. In the inmost of all loops, the rows from all levels would be combined to a result row.

To make the product operation on more than two tables more tangible, let's define the **SimpleLanguage** table that contains the languages of only the official languages spoken in the countries with the country codes 'GBR' and 'CAN':

```
mysql> CREATE TABLE SimpleLanguage AS
->   SELECT CountryCode, Language
->     FROM CountryLanguage
->   WHERE CountryCode IN ('CAN','GBR') AND IsOfficial = 'T';
```

And the contents are:

```
mysql> SELECT * FROM SimpleLanguage;
+-----+-----+
| CountryCode | Language |
+-----+-----+
| CAN         | English   |
| CAN         | French    |
| GBR         | English   |
+-----+-----+
```

If we would take the product of **SimpleCity** and **SimpleCountry** and then calculate the product of that result table with the **SimpleLanguage** table, we can already predict the dimensions of the final result. We would get:

- All columns from **SimpleCity** (3 columns), **SimpleCountry** (3 columns) and **SimpleLanguage** (2 columns) yields $3 + 3 + 2 = 8$ columns.
- The multiplication of the number of rows in the **SimpleCity** (2 rows), **SimpleCountry** (2 rows) and **SimpleLanguage** (3 rows) tables yields $2 * 2 * 3 = 12$ rows.



367

In the following chart we show the product table. Apart from listing all rows, we've highlighted the first occurrence of each unique row to allow a better understanding of the structure of the resulting Cartesian product:

SimpleCity			SimpleCountry			SimpleLanguage		
CityName	CountryCode	CityPop	Code	CountryName	CountryPop	CountryCode	Language	
London	CAN	339917	CAN	Canada	34261700	CAN	English	
London	CAN	339917	CAN	Canada	34261700	CAN	French	
London	CAN	339917	CAN	Canada	34261700	GBR	English	
London	CAN	339917	GBR	United Kingdom	65585740	CAN	English	
London	CAN	339917	GBR	United Kingdom	65585740	CAN	French	
London	CAN	339917	GBR	United Kingdom	65585740	GBR	English	
London	GBR	7285000	CAN	Canada	34261700	CAN	English	
London	GBR	7285000	CAN	Canada	34261700	CAN	French	
London	GBR	7285000	CAN	Canada	34261700	GBR	English	
London	GBR	7285000	CAN	United Kingdom	65585740	CAN	English	
London	GBR	7285000	CAN	United Kingdom	65585740	CAN	French	
London	GBR	7285000	CAN	United Kingdom	65585740	GBR	English	

As is apparent from this three table example, the results rapidly become very bulky. We won't write out these large products all the time as it becomes quite impractical. However, you are encouraged to practice deriving these product tables a few times to ensure the concept is completely clear.

368

13.2.5 Filtering out undesired rows

Let's get back to our initial Cartesian product created from the **SimpleCity** and **SimpleCountry** tables. Before we took a dive into the details of the Cartesian products, we already acknowledged that our product table contains all the rows we needed to provide the final result, as well as a number of undesired rows.

It is not that hard to see how we can obtain the correct combinations of rows:

SimpleCity			SimpleCountry		
CityName	CountryCode	CityPop	Code	CountryName	CountryPop
London	GBR	7285000	CAN	Canada	34261700
London	GBR	7285000	GBR	United Kingdom	65585740
London	CAN	339917	CAN	Canada	34261700
London	CAN	339917	GBR	United Kingdom	65585740

We simply need to check whether the value in the **CountryCode** column from the **SimpleCity** table matches the value in the **Code** column from the **SimpleCountry** table. Any rows that do not satisfy that condition are combinations of unrelated rows, which can thus be discarded.



13.2.6 Columns

Now that we have selected the proper row combinations, it is a small step to remove all but the CityName and CountryName columns:

SimpleCity			SimpleCountry		
CityName	CountryCode	CityPop	Code	CountryName	CountryPop
London	GBR	7285000	GBR	United Kingdom	65555740
London	CAN	339917	CAN	Canada	34261700

And thus we have obtained the desired result consisting of the city names along with the name of the corresponding country.

369

SimpleCityCountry	
CityName	CountryName
London	United Kingdom
London	Canada

370

13.2.7 Joins and foreign keys

In an earlier chapter we explained that the **CountryCode** column in the **City** table is a foreign key (not necessarily a foreign key constraint) that references the **Code** column in the **City** table, and it is no coincidence that we use this foreign key to determine which row combinations are considered 'correct'. In fact, in the vast majority of cases, tables are joined based on a foreign key.

There is a good reason why join conditions are so often based on foreign keys. In a properly designed database, the data model will be highly normalized in order to achieve maximum maintainability of the data. A normalized data model is the best guarantee that there will be no inconsistencies as a result of adding, removing or modifying data (data anomalies).

However suitable for storage, a normalized structure is usually not suitable for direct consumption by applications and end users. In most cases data needs to be presented in a way that allows humans to easily relate the data to the real-world objects they represent, and in most cases the normalized data is unable to do that. The solution is to join the related pieces of data. When used like this, the join operation is basically used as a means to -temporarily, and only for the purpose of presentation – create an unnormalized result table out of normalized tables, in other words, to *denormalize* the data.

371

Joining and denormalization

We can easily see how the result of joining two normalized (that is to say, 3NF or higher) tables over a foreign key will lead to a result table that is not normalized (that is to say, 2NF or lower).

 **Instructor's notes:** This last step, the removal of the columns has in itself nothing to do with the join operation. We're just mentioning it here for completeness as we started out in an attempt to produce a list of city names along with the corresponding country names

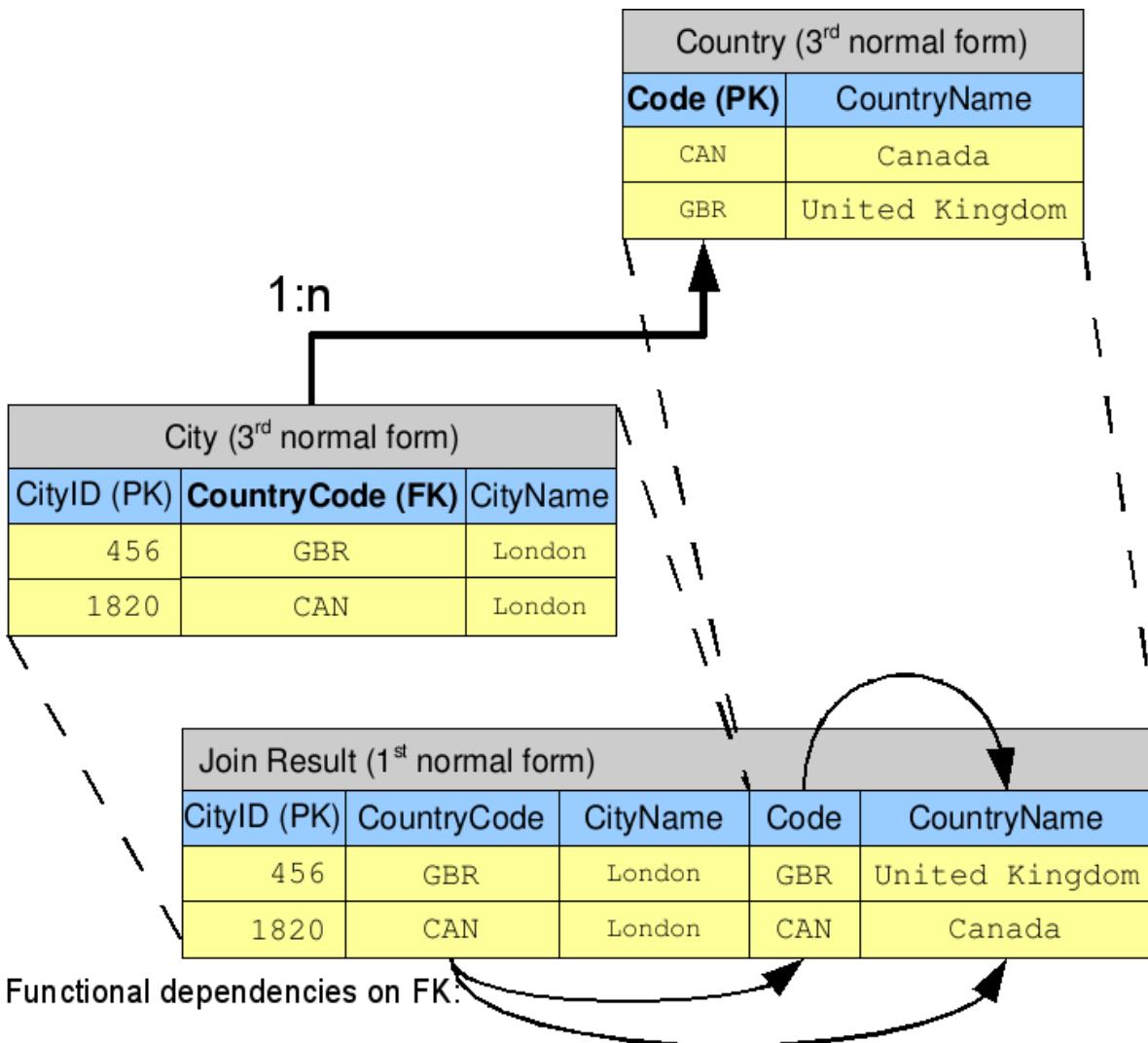
- If we pretend the join result is a real table, and we had to choose a primary key for it, we would create the primary key on the columns that also formed the primary key in the referencing table. (We need to establish the primary key of the join result, because the definition of the normal forms depends on it.) If we joined correctly, we will find exactly one row in the join result for each row in the referencing table, and since the join did not produce any extra rows, the columns that made up the primary key of the referencing table should still be unique in the join result.
- In the join result, the columns that originate from the referenced table are functionally dependent upon the foreign key columns in the referencing table. This functional dependency immediately follows from the fact that we used the foreign key to figure out which rows from the referenced table matched rows in the referencing table. So, if the foreign key columns do not also form a unique or primary key, the columns that originate from the referenced table are dependent upon a non-key. This means that the join result is not in 3NF, but a lower normal form (2NF or 1NF).
- Along the same lines of the previous argument, the columns originating from the referenced table are also dependent upon the columns that made up the primary key in the referenced table. If the columns in the join result that originate from the primary key columns of the referenced table do not also form a unique key in the join result, this again constitutes a functional dependency on non-key columns. Again this means that the join result has a lower normal form than 3NF.

In this sense, the process of joining is like 'pulling in' columns from the referenced table into the referencing table. This is basically the opposite process from normalizing a 1NF or 2NF table to a higher normal form, where tables with non-key functional dependencies are split off and pulled out of the initial table material.

There is nothing wrong with this type of denormalization, as the result is completely derived from the underlying, normalized data stored in the database. This guarantees that the denormalized result and the redundancy that comes along with it is always consistent. Remember that the problem with unnormalized data is not so much the structure or the redundancy itself, it is the fact that redundant storage of data increases the chance for inconsistencies – a problem that is completely avoided when the unnormalized data is completely controlled by putting together pieces of normalized data.



372





InLine Lab 13-A

In this lab you will practice what you've learned so far about the Cartesian product and obtaining related data from different tables.

Step 1. Calculate the dimensions of a cartesian product

- Consider the following excerpts from the **City** and **CountryLanguage** tables:

ID	Name	CountryCode	CountryCode	Language
516	York	GBR	CAN	English
1833	York	CAN	CAN	French
			GBR	English

- Calculate the dimension of the Cartesian product of these tables.

The Cartesian product is formed by making all possible distinct pairs of rows from two tables. The number of columns of the product is therefore the sum of the number of columns. The number of rows is calculated by multiplying the number of rows of both tables:

$$\# \text{columns} = 3 + 2 = 5$$

$$\# \text{rows} = 2 * 3 = 6$$

Step 2. Write out the Cartesian product

- Manually calculate the Cartesian product of these tables, and write down the result. Be sure to write down both the rows as well as the column headings for the product table. The minimize the chance on error, use a systematic approach to generate all row pairs. When done, verify that the resulting table has the dimensions calculated in the previous step.

The Cartesian product is:

ID	Name	CountryCode	CountryCode	Language
516	York	GBR	CAN	French
516	York	GBR	CAN	English
516	York	GBR	GBR	English
1833	York	CAN	CAN	French
1833	York	CAN	CAN	English
1833	York	CAN	GBR	English

Depending on the system you used to generate all possible row pairs, the order of the rows may be different in your solution. However, the important thing is that your result contains all row pairs exactly once.



Step 3. Filter matching rows

1. In the Cartesian product table you produced manually during the previous step of the lab, mark all rows that have matching values in the **CountryCode** columns of the original tables.

The following rows should have been marked:

ID	Name	CountryCode	CountryCode	Language
516	York	GBR	GBR	English
1833	York	CAN	CAN	French
1833	York	CAN	CAN	English



13.3 Joining tables in SQL

SQL offers a number of syntax constructs that allow us to quite literally perform the join process described in the previous section. In this section we'll explore a few of them.

13.3.1 SQL and the Cartesian product

The previous section explained the join operation in terms of an intermediate result table constructed from the combination of all rows and all columns from the joined tables: the Cartesian product.

373

The comma join

One way of obtaining a Cartesian product using SQL is the so-called *comma join*. (There is another SQL syntax to obtain the Cartesian product, namely the **CROSS JOIN** syntax which is discussed later on in this chapter.) To perform the comma join between the **SimpleCity** and **SimpleCountry** tables, we can simply put both table names in the **FROM** clause, separated by a comma:

```
mysql> SELECT * FROM SimpleCity, SimpleCountry;
+-----+-----+-----+-----+-----+-----+
| CityName | CntryCode | CityPop | Code | CntryName | CntryPop |
+-----+-----+-----+-----+-----+-----+
| London   | GBR      | 7285000 | CAN  | Canada    | 34261700 |
| London   | CAN      | 339917  | CAN  | Canada    | 34261700 |
| London   | GBR      | 7285000 | GBR  | United Kingdom | 65585740 |
| London   | CAN      | 339917  | GBR  | United Kingdom | 65585740 |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

NOTE: Note: the column headings are abbreviated to fit the result on the page

The effect of the comma is that both tables are taken together and treated as one result table that contains all columns and all possible pairs of rows from the two tables separated by the comma. The columns of the result are in order of the tables, and the rows are constructed by taking each row from the **SimpleCity** table and combining it with all rows from the **SimpleCountry** table.



374

13.3.2 Using a WHERE clause for the join condition

In the previous section, we explained that the Cartesian product contains a number of rows that correctly pair the **SimpleCity** rows with their corresponding **SimpleCountry** rows. Apart from these desired combinations, there are many undesirable combinations too, that is, **SimpleCity** rows that are paired to a **SimpleCountry** row that does not represent the country in which the city resides. In an earlier chapter we learned that we use a **WHERE** clause to preserve only desired rows (and discarding others).

If we were to set up a **WHERE** clause to retain those **SimpleCity** rows only with their corresponding **SimpleCountry** rows, we should require that the value in the **CountryCode** column of the **SimpleCity** table is equal to the **Code** value from the **SimpleCountry** table:

```
mysql> SELECT *
    -> FROM SimpleCity, SimpleCountry
    -> WHERE CountryCode = Code;
+-----+-----+-----+-----+-----+
| CityName | CntryCode | CityPop | Code | CntryName      | CntryPop |
+-----+-----+-----+-----+-----+
| London   | CAN       | 339917 | CAN  | Canada        | 34261700 |
| London   | GBR       | 7285000 | GBR  | United Kingdom | 65585740 |
+-----+-----+-----+-----+-----+
```

375

Of course, this is just an ordinary **WHERE** clause, allowing us to test any possible condition, including conditions that are true in case the paired rows happen to be related. In this example, the comparison expression

```
CountryCode = Code
```

ensures we end up with only combinations of *corresponding* **SimpleCity** and **SimpleCountry** rows. Such a condition is called the *join condition*.

Even though we used the **WHERE** clause to apply a join condition, there is nothing that prevents us from adding other criteria too, even ones that have nothing to do with combining related rows. For example, if we would want to look for those rows where the city population is less than 1 million, we can do:

```
mysql> SELECT *
    -> FROM SimpleCity, SimpleCountry
    -> WHERE CountryCode = Code
    -> AND CityPop < 1000000;
+-----+-----+-----+-----+-----+
| CityName | CntryCode | CityPop | Code | CntryName      | CntryPop |
+-----+-----+-----+-----+-----+
| London   | CAN       | 339917 | CAN  | Canada        | 34261700 |
+-----+-----+-----+-----+-----+
```



So, apart from retaining only corresponding rows, the additional criterion also ensures that the resulting city has a population less than 1 million. The comparison expression:

```
CityPop < 1000000
```

is also a condition, but is not considered to be part of the *join* condition. Generally speaking, a join condition is a condition that expresses the relationship between two tables that are joined. In the vast majority, join conditions take the form of column comparisons using the equals comparison operator, or combinations (using the logical AND operator) of such column-by-column comparisons.

NOTE: In the context of a comma join, the term join-condition is not very well-defined, but usually, if two tables are joined, all expressions that compare columns from the one table to the joined table are together considered to be the join-condition. In the following section we'll see more examples of join conditions.

It is important to realize that the join condition itself is syntactically speaking 'just' a condition like any other. The fact that the condition has the semantics of defining the relationship between the rows in the joined tables is what makes it the join condition.





InLine Lab 13-B

In this lab you will practice what you've learned so far about the SQL syntax to create a Cartesian product and to join tables using a join condition as part of the **WHERE** clause.

Step 1. Recap of the Cartesian product

- Determine the number of columns and rows in the **Country** and **City** tables. Compute the dimension of the Cartesian product of the **Country** and **City** tables.

The **Country** table has 15 columns; the **City** table 5 columns. This means the Cartesian product will have $15 + 5 = 20$ columns. The **Country** table contains 239 rows; the **City** table 4079. This means the Cartesian product will have $239 * 4079 = 974881$ rows.

Step 2. Use SQL to produce a Cartesian product

- Execute a **SELECT** statement to obtain the Cartesian product of the **Country** and **City** tables:

```
mysql> SELECT *
-> FROM Country, City;
```

Observe that it takes some time before the first rows are returned. (Don't wait around until all rows are returned. Instead, use Ctrl + C to abort the query when you get bored seeing the data flow by.)

A huge result is returned, and most probably it'll last too long to wait for. You should be able to verify the rowcount, even if you aborted the query. The seconds indication is the amount time that was required to calculate the product, although the time to send it to the client is most likely way longer than that.

```
974881 rows in set (#.# sec)
```

Step 3. Adding a join condition

- Imagine adding a condition to the query so that it returns **City** rows that match the **Country** row that represents the country in which the city resides. Will this change change the number of rows returned by the query? Will it change the number of columns of the resultset? Assuming that for each row in the **City** table, a matching row exists in the **Country** table, how many rows will be returned?
- Use a **WHERE** clause to add a condition that ensure only corresponding **City** / **Country** rows are returned:

```
mysql> SELECT *
-> FROM City, Country
-> WHERE CountryCode = Code;
```

Adding a condition limits the number of combinations in the result; hence it can only decrease the number of returned rows. The condition does not change the number of columns. If the **Country** table contains a matching row for each **City**, the number of rows in the result equals the number of rows in the **City** table. Executing the query returns matching 4079 **City** / **Country** combinations (= #**City** rows).



376

13.3.3 Qualifying ambiguous column names

When joining tables, there is a potential ambiguity when referring to a particular column. Ambiguity results when two or more tables that are referenced in the same SQL statement have identically named columns. Consider the following example using the `world` database:

```
mysql> SELECT Continent, CountryCode, Name
-> FROM City, Country
-> WHERE CountryCode = Code;
```

If we execute it, we get the following error:

```
ERROR 1052 (23000): Column 'Name' in field list is ambiguous
```

The reason is that both the `City` table as well as the `Country` table contain a column called `'Name'`. The `SELECT` list refers to some column called `'Name'`, but it is impossible to determine which column is meant: `'Name'` could refer to either the `Name` column from the `City` table, or the `Name` column of the `Country` table.

377

In order to resolve the ambiguity, the column name can be *qualified* by prepending the table name, separating the table name and the column name with a period (dot), allowing both `Name` columns to be referred to unambiguously:

```
mysql> SELECT Continent, CountryCode, Country.Name, City.Name
-> FROM City, Country
-> WHERE CountryCode = Code;
```

All column references may be qualified in this manner, even the ones that do not cause any ambiguity. In addition, column references may be qualified in most parts of the SQL statement, such as the `SELECT` list, the `WHERE` clause, the `ORDER BY` clause and so on and so forth.

378

13.3.4 Using table aliases

Inside an SQL statement, tables can be given a so called *alias*. A table alias is like a second table name for local use within the SQL statement that defines the alias. This alias may then be used instead of the full table name to qualify column references. The general syntax for a table reference including an alias is:

```
<table-reference> [AS] <alias>
```

So, a table alias can be created simply by adding the alias behind the table name. Optionally, the `AS` keyword may be used in between the table reference and the alias. To see how this works, consider the following modification of the previous example query:

```
mysql> SELECT Continent, CountryCode, co.Name, ci.Name
-> FROM City AS ci, Country AS co
-> WHERE CountryCode = Code;
```



Using aliases may be advantageous for a number of reasons (not necessarily related to joining tables):

379

- Aliases can be chosen to be a much shorter name than the original table name, relieving the burden of typing long table names
- Consistent use of aliases makes queries more resilient in case a table is renamed. In the query, only the table names need to be adjusted, leaving all qualified column references be.
- To resolve ambiguity of table names. Whenever a query includes two instances of the same table, an alias needs to be given to at least one of these tables in order to refer to either one of the tables unambiguously.
- To clarify the role or purpose of the table in the query.

To illustrate this last point, consider the following query:

```
mysql> SELECT Capital.Name, Country.Name  
-> FROM Country, City AS Capital  
-> WHERE Capital = Capital.ID;
```

By using the **Capital** alias for the **City** table, it is much clearer how this particular instance of the **City** table is related to the **Country** table.

380

Common Alias Error

One thing to keep in mind is that if a table has been given a table alias, the alias must be used instead of the table name when qualifying a column. Consider this modification of the previous query:

```
mysql> SELECT Country.Name, City.Name  
-> FROM Country, City AS Capital  
-> WHERE Capital = Capital.ID;
```

Here, the second expression in the **SELECT** list attempts to refer to the **Name** column of the **City** table. However, the **City** table has been given the table alias **Capital**, locally renaming that particular instance of the **City** table. This results in an error:

```
ERROR 1054 (42S22): Unknown column 'City.Name' in 'field list'
```

This type of error can be hard to spot, so beware of it.





InLine Lab 13-C

In this lab you will modify the query from the previous lab to return only specific columns

Step 1. Using qualified columns

1. Change the query from the previous lab to return only the city and country name rather than all possible columns. Use the table name to qualify the column references in the **SELECT** list:

```
mysql> SELECT City.Name, Country.Name  
-> FROM City, Country  
-> WHERE CountryCode = Code;
```

A list of city names along with the name of the country in which the city resides is returned.

Step 2. Use table aliases

1. Add the table alias Town for the **City** table:

```
mysql> SELECT City.Name, Country.Name  
-> FROM City AS Town, Country  
-> WHERE CountryCode = Code;
```

What happens if you try to execute this query? An error occurs:

```
ERROR 1054 (42S22) : Unknown column 'City.Name' in 'field list'
```

Can you explain the error and describe a way to solve it whilst maintaining the table alias?

Step 3. Qualify the columns using the table alias

1. Qualify all column references to **City** columns with the new table alias:

```
mysql> SELECT Town.Name, Country.Name  
-> FROM City AS Town, Country  
-> WHERE Town.CountryCode = Code;
```

The query can be executed again, returning the same result as before.



381

13.4 Basic JOIN syntax

We just witnessed that the combination of a comma join and an ordinary **WHERE** clause is capable of delivering the desired result. One of the problems with that approach is that the join condition is 'just' another **WHERE** condition. In this case, the join condition was easy to spot but as more tables are joined in this manner, and as more 'ordinary' (non-join) conditions are added to the **WHERE** clause, it becomes increasingly difficult to understand exactly how the joined tables are related.

To circumvent this problem, SQL also defines an explicit **JOIN** syntax, which uses the **JOIN** keyword instead of a comma to denote a join operation.

NOTE: It is important to realize that the term "join" can mean several things. On the one hand, it denotes an operation that may be applied to two tables. In the context of SQL, it may also denote the **JOIN** keyword, or the specific flavor of the join operation as performed when using that **JOIN** keyword in SQL statements. In this guide we will always simply write "join" to denote the former usage, and **JOIN** to denote the latter.

The confusion is caused by the fact that it is perfectly possible to write a SQL statement that performs a join, without actually ever writing the **JOIN** keyword: the comma-join. To add even more to the confusion, people often use a phrase like "to write a join" to express that they are writing an SQL statement that performs a join, however, this does not automatically mean that they are "writing a **JOIN**". Similarly, the terms "the join syntax" and "the **JOIN** syntax" are different things – "the join syntax" would mean: all of the SQL syntax constructs that allow a join to be performed, whereas "the **JOIN** syntax" would mean all syntax constructs that include the **JOIN** keyword.

Here is the general syntax model for the **JOIN** syntax:

```
<table-reference> [<join-type>] JOIN <table-reference>
ON <join-condition>
```

NOTE: This is a simplified syntax model that leaves out a number of details. For example, the **ON** clause is optional in some cases, and in many cases, the **USING** clause can appear instead of an **ON** clause. All these elements are discussed later on in this chapter, but for now it is necessary to focus only on the fact that the **JOIN** syntax keeps the joined tables and the join condition together and separates the ordinary **WHERE** clauses from the join-condition.

A **<table-reference>** appears on both sides of the **JOIN** keyword. This is actually quite like the earlier example of the comma join, where to tables where separated by a comma. In addition, the **JOIN** syntax allows the **ON** keyword, followed by the join-condition. This way, the join-condition is kept near the joined tables. This is different from the comma join: the comma join does not allow an **ON** keyword, and any join-conditions are 'just' **WHERE** conditions with the 'side effect' of retaining only combinations of related rows from the joined tables. Using the **JOIN** keyword and the **ON** clause for the join-condition allows a more explicit way to express the relationship between the joined tables, separating the join conditions from any other 'normal' conditions.



If we rewrite the previous comma join example using the **JOIN** syntax, we get:

```
mysql> SELECT *
-> FROM SimpleCity JOIN SimpleCountry
-> ON CountryCode = Code;
```

If we would have additional criteria, they would be nicely separated from all the lines that make up the **JOIN** proper:

```
mysql> SELECT *
-> FROM SimpleCity JOIN SimpleCountry
-> ON CountryCode = Code
-> WHERE CityPop < 1000000;
```

So, the **JOIN** syntax allows us to more explicitly denote the intent of join operations.

13.4.1 Non-join conditions in the ON clause

We just argued that the **ON** clause can be used to specify the join condition. However, syntactically **ON** is just like **WHERE**: it will allow almost any type of logical expression, even if it contains conditions that have nothing to do with the joined tables. For example, this:

```
mysql> SELECT *
-> FROM SimpleCity JOIN SimpleCountry
-> ON CountryCode = Code
-> AND CityPop < 1000000;
```

is perfectly legal SQL. The expression `CityPop < 1000000` cannot be regarded as a true join condition, as it does nothing to relate the joined tables to each other. It is simply a condition that is to be applied to the **SimpleCity** table, and does not touch the **SimpleCountry** table at all.

In this example, the result happens to be identical regardless of whether the `CityPop < 1000000` condition is placed in the **ON** or the **WHERE** clause. However, we will see later on examples of join operations where that placement does matter.

Generally speaking, it is advisable to reserve the **ON** clause only for proper join conditions. That is, the **ON** clause is best used to only compare columns from the joined tables to each other. Any additional criteria should appear in the **WHERE** clause.





InLine Lab 13-D

In this lab you will use the basic **JOIN** syntax to rewrite the comma join query constructed in the previous lab.

Step 1. Rewriting comma joins

1. Rewrite the query from the previous lab to use the **JOIN** keyword:

```
mysql> SELECT Town.Name, Country.Name  
      -> FROM     City AS Town JOIN Country  
      -> ON       Town.CountryCode = Code;
```

Verify that the query still returns the same result.

The comma separating the tables is replaced by the keyword **JOIN**; the **WHERE** keyword is replaced by the keyword **ON**.

Step 2. Adding an extra condition

1. Change the query to return only a result row in case the country is situated in South America

```
mysql> SELECT Town.Name, Country.Name  
      -> FROM     City AS Town JOIN Country  
      -> ON       Town.CountryCode = Code  
      -> WHERE    Continent = 'South America';
```

A **WHERE** clause is added, containing the condition to require the country is situated in South America. The condition in the **WHERE** clause is separate from the join condition in the **ON** clause.



382

13.5 Inner joins

The join methods we discussed so far, the comma join and the usage of the **JOIN** keyword, are both examples of so-called *inner* joins. The defining characteristic of an inner join operation on two tables is that the result consists of only those combinations of rows for which the join condition is satisfied.

When using the **JOIN** keyword without an explicit keyword to denote the join-type, **INNER** is implied. Consider the following statement using the **INNER JOIN** syntax:

```
mysql> SELECT *
    -> FROM SimpleCity INNER JOIN SimpleCountry
    -> ON CountryCode = Code;
```

The **INNER** keyword could just as well have been omitted, because **INNER** is implied when the join type is not explicitly specified. So this is equivalent to the earlier example:

```
mysql> SELECT *
    -> FROM SimpleCity JOIN SimpleCountry
    -> ON CountryCode = Code;
```

We argued earlier that this is also equivalent to the comma-join example discussed before (provided the join-conditions are equivalent). So, all these examples represent inner join operations. In fact, there are many more variants of the join syntax that would qualify as an inner join. Instead of discussing each and every one of them, we will first describe the outer join operation, after which it will be easier to understand other inner join variations.

383

13.5.1 Inner join pseudocode

At this point, it's good to look back at the pseudo code example we used to explain the construction of the Cartesian product. We can express the inner join operation in pseudocode as follows:

```
goto_first_row(SimpleCity)
while has_rows(SimpleCity) do

    goto_first_row(SimpleCountry)
    while has_rows(SimpleCountry) do

        if SimpleCity.CountryCode = SimpleCountry.Code then
            new_row= current_row(SimpleCity) + current_row(SimpleCountry)
            add_row(Result, new_row)
        end if

        goto_next_row(SimpleCountry)
    end while

    goto_next_row(SimpleCity)
end while
```



This pseudocode is in many ways very similar to the pseudocode example we used earlier to illustrate the process for calculating a Cartesian product. In both cases, we use nested loops to generate all combinations of rows from the **SimpleCity** and **SimpleCountry** tables. The difference is that in this case, we do not automatically generate a result row for each combination; rather we create a result row only in case the join condition is satisfied. This condition is tested inside the inner loop.

384

13.5.2 Omitting the join condition

We already explained that in for some join types, the join condition is optional. This is the case for these variants of the **INNER JOIN** and **JOIN** syntax. Omitting the join condition from an inner join operation has essentially the effect of creating a Cartesian product. That is, this statement:

```
mysql> SELECT *
->   FROM SimpleCity INNER JOIN SimpleCountry;
```

is equivalent to this statement:

```
mysql> SELECT *
->   FROM SimpleCity, SimpleCountry;
```

Because the effect of omitting the join condition is so radically different from the concept of an inner join operation, it doesn't make much sense to consider this a true inner join operation, despite the usage of the **INNER JOIN** phrase.

In most cases, it does not make much sense to deliberately omit the join condition. However, it is allowed syntax which means you may accidentally forget the join condition. Similarly, you may make the mistake of omitting the **ON** keyword and join condition, but list the relevant join condition in the **WHERE** just like you would for a comma join. In such a case, you should probably promote your ordinary **WHERE** clause to a proper join condition and list it in the **ON** clause where it belongs.





InLine Lab 13-E

In this exercise you will use the methods covered in this chapter for the **INNER JOIN** keywords. This will require a MySQL command line client and the **world** database.

Step 1. Finding countries that speak 'Lithuanian'

- Using the **INNER JOIN** keywords, find all the cities in the **world** where the '**Lithuanian**' language is spoken. List the language and the city names:

```
mysql> SELECT City.Name, Language
-> FROM CountryLanguage INNER JOIN City
-> USING (CountryCode)
-> WHERE Language = 'Lithuanian';
```

Shows all the cities where Lithuanian is spoken.

Step 2. Find information about the city of San Antonio

- Retrieve the country name and city district that has a city named 'San Antonio'

```
mysql> SELECT Country.Name, District FROM Country
-> INNER JOIN City
-> ON CountryCode = Code
-> WHERE City.Name = 'San Antonio';
```

Returns list of one country and its district;

Name	District
United States	Texas



385

13.6 Outer joins

In our discussion of inner joins, we explained that the defining characteristic of an inner join is that its result contains only those rows for which the join condition could be satisfied. In the examples we covered so far, we used this mainly to match rows in a referencing table (**SimpleCity**) to the rows in the referenced table (**SimpleCountry**). Because of this, our inner join results never contained any of the non-matching row pairs that we got in the Cartesian product of **SimpleCity** and **SimpleCountry**. So far, we could always find exactly one row in the **SimpleCountry** table for each row from **SimpleCity**, and thus, the result of the inner join always contained exactly one row for each row from **SimpleCity**.

In this section, we will consider the case where not all rows from one of the joined tables can be matched to a row in the other table. With an inner join, such a row would be lost from the result, because the join condition cannot be satisfied. Here we learn a construct called the outer join which allows us to retain such a row.

13.6.1 Again: Two simple tables

So far, we joined cities with countries to find out in which country a particular city is situated. There is another relationship between countries and cities: a city can be the capital city of a country. In the world database, this relationship is realized by the **Capital** column in the **Country** table. The **Capital** column is a foreign key that references the **ID** column in the **City** table. This allows us to lookup a country's capital by finding whichever city's value in the **ID** column is equal to the value found in the **Capital** column for that particular country.

Let's recreate the **SimpleCountry** table to include this relationship:

386

```
mysql> CREATE TABLE SimpleCountry
    -> AS
    -> SELECT Code, Name AS CountryName, Capital
    -> FROM Country
    -> WHERE Code IN ('CAN', 'GBR');
```

with the contents:

Code	CountryName	Capital
CAN	Canada	1822
GBR	United Kingdom	456



387

We need to recreate the **SimpleCity** table likewise to include the **ID** column:

```
mysql> CREATE TABLE SimpleCity
-> AS
-> SELECT ID as CityID, Name AS CityName, CountryCode
-> FROM City
-> WHERE Name Like 'London';
```

And the contents are:

CityID	CityName	CountryCode
456	London	GBR
1820	London	CAN

388

Finding the capital city

Now we can write a query that joins the **SimpleCountry** table to the **SimpleCity** table in order to list all countries together with their capital city. So, we can write a join like this:

```
mysql> SELECT CountryName, CityName
-> FROM SimpleCountry INNER JOIN SimpleCity
-> ON Capital = CityID;
```

Here, we see a simple inner join like discussed before, but instead of looking for just any city that corresponds to a country, we want to find only the capital of each country, and thus we have modified the join condition accordingly: for each row from the **SimpleCountry** table we now take the value in the **Capital** column and use it to find a city which has that value in its **CityID** column.

NOTE: This query lists the **SimpleCountry** table first, and then the **SimpleCity** table, which is the opposite order as compared to the previous examples. However, this has no significance for this particular query. It is done only to more closely resemble the task of reporting all countries with their capital. In the prior examples, the main focus of the question was cities which is why we listed **SimpleCities** as the first table.

When we execute this query, we get the following result:

CountryName	CityName
United Kingdom	London



389

Interestingly, only the row for the United Kingdom is returned, whereas we know that we have a country row for Canada too. However, if we write out the Cartesian product and closely examine the columns in the join condition it is immediately clear why we only see one row:

SimpleCountry			SimpleCity		
Code	CountryName	Capital	CityID	CityName	CountryCode
CAN	Canada	1822	456	London	GBR
GBR	United Kingdom	456	456	London	GBR
CAN	Canada	1822	1820	London	CAN
GBR	United Kingdom	456	1820	London	CAN

Even though our **SimpleCity** table contains two rows, only the city with `CityID` 456 (London in Great Britain) matches the value in the `Capital` column of a row in the **SimpleCountry** table. We explained that the inner join operation retains only the rows that satisfy the join condition, discarding any other rows, and this explains why only Great Britain with London is returned in the result.

NOTE: The city with `CityID` 1822, capital of Canada is missing because we created our **SimpleCity** table by only including cities called 'London' – In the original world database the capital of Canada is present in the **City** table.



13.6.2 Retaining rows even when no match is found

390 Although correct, this result of the previous query is not exactly what we intended. We wanted to see a list of *all* countries with their capital city. But we didn't really intend to exclude those countries for which we happened to be unable to find the capital. So we need to rephrase our request: We'd like to see a list of *all* countries, and *if possible*, its capital city.

391 So, instead of only retaining those rows for which the join condition is satisfied, we would also like to retain the rows from **SimpleCountry** in case none of the rows matches the condition. This is achieved by an so called *outer join* operation. Before we discuss the SQL syntax for outer joins in detail, let's first consider the pseudo-code to would achieve this result:

```

goto_first_row(SimpleCountry)
while has_rows(SimpleCountry) do

    has_no_related_rows = TRUE

    goto_first_row(SimpleCity)
    while has_rows(SimpleCity) do

        if SimpleCountry.Capital = SimpleCity.ID then
            has_no_related_rows = FALSE
            new_row= current_row(SimpleCountry) + current_row(SimpleCity)
            add_row(Result, new_row)
        end if

        goto_next_row(SimpleCity)
    end while

    if has_no_related_rows then
        new_row= current_row(SimpleCountry) + new_null_row(SimpleCity)
        add_row(Result, new_row)
    end if

    goto_next_row(SimpleCountry)
end while

```

When comparing this pseudocode to the pseudocode example given for the **INNER JOIN** operation, we again notice a lot of similarities. We see nested loops for the two tables, and inside the inner loop a test for the join condition to create a result row.

This example however has some additional statements. First of all, we maintain the variable `has_no_related_rows` to see if the inner loop added any rows to the result. For each row of **SimpleCountry**, the variable is initialized to TRUE, and if the join condition is satisfied in the inner loop, it is set to FALSE, marking the fact that there is at least one related row in the **SimpleCity** table.



After the inner loop, the value of the variable is tested. If we did find at least one related row, we do nothing at all. However, if none of the rows in the **SimpleCity** table satisfied the join condition, we add one row to the result. This row is composed the current row from the **SimpleCountry** table as usual. The part of the result row that would normally be occupied by a related **SimpleCity** row, is now filled with a **NULL**-row, that is, a row that contains only **NULL** values.

The effect of this process is that the result will always contain at least one row for each row that was processed in the outer loop (in this case, for each row from **SimpleCountry**). Note that that is not the case when using the inner join. In the case of the inner join, rows are only added in case the join condition is satisfied in the inner loop, so some rows encountered in the outer loop may not appear in the final result.

For the outer join, the result always contains at least one row for each row processed in the outer loop. In case the join condition was never satisfied by the inner loop, the result row has all **NULL**s for the part of the result row corresponding to the inner loop. When the join condition is satisfied at least once for a particular in the outer loop, the result will contain only the combinations of rows that satisfy the join condition, just as it is the case for the inner join operation.

We will now discuss two variants of the outer join syntax, the **LEFT** and the **RIGHT** outer join.

13.6.3 Left outer join

392

Below follows the model for the **LEFT OUTER JOIN** syntax:

```
<left-table> LEFT [OUTER JOIN] <right-table>
ON <join-condition>
```

Here, the **<left-table>** and **<right-table>** are simply table references such as we saw in the **INNER JOIN** syntax. However, we need different denotations for the tables appearing on the left and right side of the **JOIN** keyword because the outer join operation does not treat the tables the same.

The **LEFT OUTER JOIN**:

- Returns all rows that match the **<join-condition>**
- Retains all unmatched rows from **<left-table>**
- Substitutes **NULL** for the columns of the **<right-table>** in case no match is found for a particular row from **<left-table>**.

393

Here is an example of the **LEFT OUTER JOIN** syntax. Think of this statement as the SQL variant of the outer join pseudo-code example:

```
mysql> SELECT CountryName, CityName
-> FROM SimpleCountry LEFT OUTER JOIN SimpleCity
-> ON Capital = CityID;
```



Compare this to the previous **INNER JOIN** example and notice that instead of the **INNER** keyword, we now have **LEFT OUTER** instead. The **LEFT** keyword denotes that the table that appears on the left side of the **JOIN** keyword is the table that is considered to be the 'outer' table. The result will always contain at least one row for each row in the **SimpleCountry** table (which appears on the left of the **JOIN** keyword), even if the join condition is not satisfied. Note that this relates directly to our pseudo-code example where the **SimpleCountry** table was processed in the outer loop.

The **OUTER** keyword is optional, and we will usually omit it. So, the previous example is equivalent to the following statement:

```
mysql> SELECT CountryName, CityName
-> FROM SimpleCountry LEFT JOIN SimpleCity
-> ON Capital = CityID;
```

Here is the result:

CountryName	CityName
Canada	NULL
United Kingdom	London

Note that both rows from the **SimpleCountry** table are present in the result. For the **SimpleCity** table, only the matching row is present. The part of the result row that originates from the **SimpleCity** table is filled with **NULL** values for the **SimpleCountry** row that does not match any of the **SimpleCity** rows. In the **INNER JOIN** example, this row was not present at all, because the inner join retains only the rows that satisfy the join condition.



13.6.4 Right join

We could have obtained an identical result using the **RIGHT JOIN** syntax.

The **RIGHT OUTER JOIN**:

394

- Returns all rows that match the *<join-condition>*
- Retains all unmatched rows from *<right-table>*
- Substitutes **NULL** for the columns of the *<left-table>* in case no match is found for a particular row from *<right-table>*. The rewritten statement would look like this:

```
mysql> SELECT CountryName, CityName
-> FROM SimpleCity RIGHT JOIN SimpleCountry
-> ON CityID = Capital;
```

395

Here, we used the **RIGHT** keyword instead of **LEFT**. Consequently, the table names are now reversed as compared to the **LEFT JOIN** example.

An important thing to realize about **RIGHT** and **LEFT** joins is that they are not fundamentally different. That is, they do not presume a different class of operation, they merely express a syntactical difference. It is always possible to rewrite a **LEFT JOIN** with a **RIGHT JOIN** and vice versa without changing the result of the operation. For this reason, we prefer to keep things simple and will therefore stick to the **LEFT JOIN**.

13.6.5 The join condition of outer join operations

396

The join condition for outer join operations is not special in itself: just like we saw with the inner join, the join condition can be any valid boolean expression. However, there are a few things that need to be pointed out.

The join condition cannot be omitted

Recall that MySQL allows you to omit the **ON** clause for the **INNER JOIN** syntax. Since we mentioned that, we should point out that this is not the case for any of the outer join syntaxes. The **RIGHT** and **LEFT JOIN** syntax requires a join condition, and omitting the **ON** clause is considered to be a syntax error.

To be clear – it is just as well that the join condition cannot be omitted. In our discussion of the **INNER JOIN**, We already pointed out that it does not make sense to leave out the join condition, and at least the outer join syntax does not allow one to accidentally omit the join condition.

```
SELECT CountryName,  
      CityName  
FROM SimpleCountry LEFT JOIN SimpleCity  
ON Capital = CityID
```



```
SELECT CountryName,  
      CityName  
FROM SimpleCity RIGHT JOIN SimpleCountry  
ON CityID = Capital
```



The order of the columns

In the **RIGHT JOIN** example, we reversed the order of the tables. We also reversed the join condition. In the **LEFT JOIN** example, it was `Capital = CityID` and here we changed it to `CityID = Capital`. However, this has no influence whatsoever on the evaluation of the condition: it is essentially still the same condition.

The reason for reversing the join condition is a matter of style and convention. The **JOIN . . ON** clause is easier to understand if the order of appearance of the columns in the join condition matches the order of appearance of their respective tables.

You can write the columns in any order you like – it does not matter for how the join condition is evaluated. However, you are strongly advised to use a particular style that allows you to immediately see which columns originate in which tables.

Non-join conditions in the ON clause

397

In our discussion of the inner join operation we showed that syntactically, any condition can be written in the **ON** clause. In that particular example, the placement of a particular condition in either the **ON** or the **WHERE** clause did not affect the result in any way. For outer join operations, it can make a big difference depending on whether a condition is placed in the **ON** or the **WHERE** clause.

The result of a join is always such that it is as if any conditions that appear in the **ON** clause are applied when joining the tables; after all tables are joined, the conditions in the **WHERE** clause are applied.

NOTE: It is tempting to say that the conditions in the **WHERE** clause are always applied before the conditions in any **ON** clause. However, in reality, the query optimizer decides how and when conditions are applied. The optimizer may estimate that applying particular **WHERE** conditions in an earlier stage may be more efficient. However, the result is always such that it is as if the **WHERE** conditions are applied after the **ON** conditions.

To illustrate the importance of the placement of conditions in outer joins, consider this query:

398

```
mysql> SELECT City.Name, Country.Name, Country.IndepYear
-> FROM City LEFT JOIN Country
-> ON City.ID = Country.Capital
-> WHERE City.Name IN ('Kabul', 'Oranjestad', 'Qandahar');
```

The query retrieves the cities called 'Kabul', 'Oranjestad', and 'Qandahar', and if the city is the capital of its country, the name of the country as well as the year of independence. Here is the result:

Name	Name	IndepYear
Kabul	Afghanistan	1919
Qandahar	NULL	NULL
Oranjestad	Aruba	NULL

So, Kabul is the capital of Afghanistan which became independent in 1919; Qandahar is not a capital of any country, and Oranjestad is the capital of Aruba, which is not an independent country.



Now, suppose that we want to see the country data only in case the country is independent. It is not hard to see that we need to add a condition of the form:

Country.IndepYear **IS NOT NULL**

399

But where should we add this condition? We have two possibilities: we can either add it to the join condition in the **ON** clause, or to the **WHERE** clause. First, let's try to add it to the **WHERE** clause:

```
mysql> SELECT City.Name, Country.Name, Country.IndepYear
    -> FROM City LEFT JOIN Country
    -> ON City.ID = Country.Capital
    -> WHERE City.Name IN ('Kabul', 'Oranjestad', 'Qandahar')
    -> AND Country.IndepYear IS NOT NULL;
```

In this case, the tables are joined, and the join result is filtered to retain only the rows where the year of independence is **NOT NULL**. Here is the result:

Name	Name	IndepYear
Kabul	Afghanistan	1919

When we look at the initial result before we added this condition, it is not hard to understand this result: from all cities we obtained initially, only Kabul is the capital of an independent country. However, this is not what we intended. We basically want the original set of cities and only show the country data in case the country is independent. The current query does not meet these requirements: it simply retrieves the capitals of independent countries. If we wanted to do that, we would not have needed an outer join to find the matching countries.

Let's see what happens if we move the condition to the **ON** clause:

```
mysql> SELECT City.Name, Country.Name, Country.IndepYear
    -> FROM City LEFT JOIN Country
    -> ON City.ID = Country.Capital
    -> AND Country.IndepYear IS NOT NULL
    -> WHERE City.Name IN ('Kabul', 'Oranjestad', 'Qandahar');
```



This time, we do get the right result, that is “all cities, and if they are the capital of an independent country, the country name and year of independence”:

Name	Name	IndepYear
Kabul	Afghanistan	1919
Qandahar	NULL	NULL
Oranjestad	NULL	NULL

In this case, no independent country was found to be the capital of Oranjestad. However, because of the left join, this did not discard Oranjestad from the set – it merely generated **NULL** values for the **Country** columns.

Sometimes, it can be hard to get this right the first time round. It is important to always check the exact requirements of the query against the results, and of course also to the query itself.

400

13.6.6 Choosing between inner and outer joins

It is important to realize how to choose between using an inner versus an outer join. Outer joins are usually slower than inner joins, so one should not blindly settle for outer joins. Usually, once a query is written using an outer join, measures have to be taken to deal with any **NULL** values that may appear in the result set as a result of introducing an outer join. Therefore, one should by default choose for an inner join and use an outer join only when there is a specific reason to do so. In this section, we'll describe a few typical cases to guide the choice between an inner vs an outer join.

401

Nullable columns in the join condition

Whenever the columns that appear in a join condition can be nullable, one must consider what should happen to the rows that have a **NULL** value in those columns. If it is acceptable if the result of joining the tables does not contain these rows, one can use an inner join. Otherwise, one must use an outer join.

For example, suppose you want to produce a list of all countries together with the name of the capital. If we examine the structure of the **Country** table, we notice that the foreign key column **Capital** is a nullable column:

```
mysql> DESC Country;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key  | Default | Extra |
+-----+-----+-----+-----+-----+
| Code  | char(3) | NO   | PRI  |          |       |
| Name  | char(52) | NO   |       |          |       |
| .     | .       | .    | .    |          |       |
| .     | .       | .    | .    |          |       |
| Capital | int(11) | YES  |       | NULL    |       |
| Code2  | char(2)  | NO   |       |          |       |
+-----+-----+-----+-----+-----+
```



This means that if we want to produce the list, we should use an outer join to look up the city using the `Capital` foreign key:

```
mysql> SELECT Country.Name,
->           City.Name
->      FROM    Country LEFT JOIN City
->      ON        Capital = City.ID;
```

If the `Capital` column is `NULL`, there will be no matching row in the `City` table (remember that nothing is equal to `NULL`, not even `NULL` itself). If we would've used an inner join, we would have lost those countries that happen to have a `NULL` in the `Capital` column. The outer join on the other hand will retain the row, and substitute `NULL` values for all `City` columns.

Joining to an outer joined table

402

Related to the case of having nullable columns in the join condition is the matter of joining against the result of an outer join. Consider a simple case where a table is outer joined to another table. The result of the outer join will contain a number of row pairs of both tables. Due to the fact that the outer join operation provides a `NULL` row in case no matching row is found, some columns in the intermediate result may be nullable, even though the underlying table columns are not nullable. In almost all these cases, joining against this outer joined table requires another outer join.

For example, suppose you want to make a list of all cities, and if the city happens to be a capital, the name of the country, and for that particular country the languages that are spoken in that country. (Admittedly this is a slightly unfocussed and unrealistic example, but it can serve as an illustration).

So, we want a list of all cities, and if the city happens to be a capital, we need to access the country data too. This means we need an outer join from the `City` table to the `Country` table over the `Capital` column:

```
mysql> SELECT      City.Name,
->           Country.Name AS `Capital of`,
->      FROM        City
->      LEFT JOIN   Country
->      ON          City.ID = Country.Capital;
```

So far, so good. This gives us a list of all the cities, and, where applicable, the country name of which it is the capital. Now, we need to join the `Country` table to the `CountryLanguage` table using the `CountryCode` foreign key in the `CountryLanguage` table.

The `CountryCode` column in the `CountryLanguage` table references the `Code` column in the `Country` table. The `Code` column makes up the primary key of the `Country` table and is thus by definition not nullable. The `CountryCode` column in the `CountryLanguage` table is also not nullable.

403

Now, normally it would be no problem to use an inner join to join the `Country` and the `CountryLanguage` table, because the columns that make up the join condition are not nullable. However, in this case, we are not exactly joining directly against the `Country` table. Rather, we are joining against an intermediate result set that was constructed by outer joining the `City` table and the `Country` table.



Due to the outer join, the columns originating from the **Country** table may be **NULL** – this will be the case for all cities that do not happen to be a capital. If we would use an inner join, the result would by definition be an empty set in these cases, because a direct comparison to **NULL** is automatically not TRUE. So, the solution is to use an outer join when joining the **CountryLanguage** table:

```
mysql> SELECT      City.Name,
->                  Country.Name AS `Capital of`,
->                  Language
-> FROM          City
-> LEFT JOIN    Country
-> ON            City.ID = Country.Capital
-> LEFT JOIN    CountryLanguage
-> ON            Country.Code = CountryLanguage.CountryCode;
```

Solving 'Not exists' problems

404

One thing to keep in mind is that an inner join can only retrieve matching row pairs in case a match already exists. For example, if we want to obtain a list of the languages that are spoken in a particular country, then an inner join between **Country** and **CountryLanguage** will return the desired result:

```
mysql> SELECT Name, Language
-> FROM   CountryLanguage INNER JOIN Country
-> ON     Code = CountryCode;
```

However, an inner join cannot return a result in case no languages are recorded for a particular country. So, a query to retrieve all countries for which no corresponding languages are stored in the database cannot be solved with an inner join. However, this can be done using an outer join:

```
mysql> SELECT Name
-> FROM   Country LEFT JOIN CountryLanguage
-> ON     Code = CountryCode
-> WHERE  Language IS NULL;
```

In the outer join solution, the result will contain at least one row for each row in the **Country** table. If any corresponding **CountryLanguage** rows exist, those will be retrieved too. However, the **WHERE** clause condition **Language IS NULL** will discard these rows from the result, leaving only those **Country** rows for which no corresponding **CountryLanguage** rows exists.

NOTE: SQL offers other constructs to solve 'Not Exists' problems, such as subqueries. In fact, there is even a special **EXISTS** operator that may be combined with the logical **NOT** operator to form **NOT EXISTS**. Despite a perhaps more natural syntax, the solution using an outer join is valid and for MySQL in particular usually faster than a true subquery. Therefor, it is important to recognize the outer join solution to a Not Exists problem, as it appears quite a few times in practical applications.



Aggregating related rows

405

Related to the not exists problem is the aggregation of detail rows for each row in a referenced table. Suppose that we want to have a list of countries, along with the number of cities that reside in the country. But what if we have a country that does not contain any cities? In some cases, it may be acceptable to omit these countries from the list, but reporting those countries as having zero cities will in many cases be closer to the request. Again, an inner join solution is incapable of reporting combinations that are not there, and is thus also incapable of counting those combinations. For the outer join, this is no problem:

```
mysql> SELECT Country.Name, COUNT(City.ID)
-> FROM Country LEFT JOIN City
-> ON Code = CountryCode
-> WHERE Country.Name = 'Antarctica'
-> GROUP BY Code;
```

Note that `COUNT(City.ID)` is used instead of `COUNT(*)`. The purpose of the query is to count the number of cities. In the case of Antarctica, there are no cities which means the columns from the `City` table will all be `NULL`. The `COUNT()` function does not take `NULL`-values into account which allows the absent cities to be counted as 0.

When solving `COUNT` problems like these one should be careful to apply the `COUNT` function to some column that is `NULL` only as a result of a missing row. If you accidentally count a nullable column, there is no way to distinguish between a case where a matching row is present while the column happens to be `NULL` and a case where no matching row is found and the `NULL`s are present as a result of an outer join. A good way to solve this is to always count a column that is part of the primary key.





InLine Lab 13-F

In this exercise you will use the methods covered in this chapter for the **LEFT/RIGHT JOIN** keywords. This will require a MySQL command line client and the **world** database.

Step 1. Find information about the 'Eastern Africa' region using a LEFT JOIN

- Find all the countries and their corresponding capitals in the 'Eastern Africa' region (use a **LEFT JOIN**):

```
mysql> SELECT Country.Name AS Country,
->           City.Name AS Capital
->     FROM Country LEFT JOIN City
->       ON Capital = ID
-> WHERE Region = 'Eastern Africa';
```

Returns a list of **Country** names and **Capital** names. There should be 20 rows in the set.

Country	Capital
Burundi	Bujumbura
Djibouti	Djibouti
Eritrea	Asmara
Ethiopia	Addis Abeba
Kenya	Nairobi
Comoros	Moroni
Madagascar	Antananarivo
Malawi	Lilongwe
Mauritius	Port-Louis
Mayotte	Mamoutzou
Mozambique	Maputo
Réunion	Saint-Denis
Rwanda	Kigali
Zambia	Lusaka
Seychelles	Victoria
Somalia	Mogadishu
Tanzania	Dodoma
Uganda	Kampala
Zimbabwe	Harare
British Indian Ocean Territory	NULL

20 rows in set (#.# sec)



Step 2. Find information about the 'Eastern Africa' region using a RIGHT JOIN

- Find all the countries and their corresponding capitals in the 'Eastern Africa' region. To base the comparison on the **City** table use a **RIGHT JOIN** (*Hint:* Use the up arrow key to retrieve the previous commands and then make the necessary change):

```
mysql> SELECT Country.Name AS Country,
->           City.Name AS Capital
->     FROM Country RIGHT JOIN City
->       ON Capital = ID
->   WHERE Region = 'Eastern Africa';
```

There should be 19 rows in the set due to the missing '**NULL**' row from the **City** comparison.

Country	Capital
Burundi	Bujumbura
Djibouti	Djibouti
Eritrea	Asmara
Ethiopia	Addis Abeba
Kenya	Nairobi
Comoros	Moroni
Madagascar	Antananarivo
Malawi	Lilongwe
Mauritius	Port-Louis
Mayotte	Mamoutzou
Mozambique	Maputo
Réunion	Saint-Denis
Rwanda	Kigali
Zambia	Lusaka
Seychelles	Victoria
Somalia	Mogadishu
Tanzania	Dodoma
Uganda	Kampala
Zimbabwe	Harare

19 rows in set (#.# sec)



13.7 Other types of joins

406

Apart from the important distinction between inner and outer joins, there are a number of other classifications. We describe a few here:

- Joins on identically named columns:
 - **NATURAL** join
 - Joins with **USING** (named columns join)
- The **CROSS JOIN** (explicit Cartesian product)
- Join condition categories
 - The equi join
 - The non-equi join
 - The **BETWEEN...AND** join
- The autojoin or self join

Natural join

The **NATURAL JOIN** is like a normal join operation with two extra characteristics:

407

- The join condition cannot be *explicitly* specified. Instead, the join condition is inferred by requiring equality for all pairs of columns that have an identical name.
- If the **SELECT** list uses the all-columns wildcard (*), then only one result column is produced for each identically named column pair

The syntax for the **NATURAL JOIN** is thus:

```
<table-reference> NATURAL [<outer-join-type>] JOIN <table-reference>
```

The following statement illustrates the **NATURAL JOIN** syntax:

```
mysql> SELECT *
      -> FROM SimpleLanguage NATURAL JOIN SimpleCity;
```



408

The **SimpleLanguage** and **SimpleCity** tables both have a **CountryCode** column that is used as a foreign key referencing **SimpleCountry**. Therefore, a join condition is inferred consisting of an equals comparison of the **CountryCode** column from both tables: Likewise, from the * a **SELECT** list is inferred consisting of all columns from **SimpleLanguage** and all columns from **SimpleCity**, eliminating the duplicate **CountryCode** column. This natural join statement is equivalent to this explicit syntax:

```
mysql> SELECT SimpleLanguage.CountryCode, Language, CityID, CityName
-> FROM SimpleLanguage INNER JOIN SimpleCity
-> ON SimpleLanguage.CountryCode = SimpleCity.CountryCode;
```

And the result is:

CountryCode	Language	CityID	CityName
CAN	English	1820	London
CAN	French	1820	London
GBR	English	456	London

At a glance this seems like a very powerful join syntax. It is certainly appealing to not specify the join condition and explicit **SELECT** list and obtain more or less the right result intuitively. Also, the statement will be to some extent immune for schema changes: if columns are added or removed from the set of columns that makes up the relationship, the join will remain working as expected (as long as the new columns are identically named of course).

However, the problem is that the **NATURAL JOIN** only works well if the database is designed so that all foreign key columns get names identical to the columns in the referenced table, and this is not always the case. In fact, this can be a source of hard to spot problems. For example, if a column is added to table, and a related table already contained a column that happened to have the same name as the added column, any statements relating these tables through a **NATURAL JOIN** will most likely be broken after adding the new column.

One of the cases where the merit of the **NATURAL JOIN** immediately breaks down is when you have two foreign keys referencing the same table. In this case, it is impossible to name all of the foreign key columns after their respective referenced column, because that would result in duplicate column names in the referencing table. So, the **NATURAL JOIN** would allow the tables to be related via one of the foreign keys. If we want to relate the tables over the other foreign key, we still have to resort to conventional explicit join syntax. This means that we are essentially using two different syntactical constructs to achieve similar things.

409

Joins with USING (named columns join)

The effect of the **USING** syntax somewhat resembles the effect of the **NATURAL JOIN** syntax in that it allows a join condition to be inferred based on column names. The standard name for this syntax is the *named columns join*. The syntax model for the **USING** syntax is shown below:

```
<table-reference> [<join-type>] JOIN <table-reference>
USING (<column-name1>[, <column-name2>, ..., <column-nameN>])
```



The column names listed between the parentheses after the **USING** keyword must be present in both joined tables. These columns are used to implicitly apply a join condition that compares a column from the table left of the **JOIN** keyword with the identically named column appearing at the right side of the **JOIN** keyword using the equality comparison.

So, the following statement:

410

```
mysql> SELECT Language, Name
-> FROM SimpleLanguage INNER JOIN SimpleCity
-> USING (CountryCode);
```

is equivalent to:

```
mysql> SELECT Language, Name
-> FROM SimpleLanguage INNER JOIN SimpleCity
-> ON SimpleLanguage.CountryCode = SimpleCity.CountryCode;
```

Like the **NATURAL JOIN** syntax, the **USING** syntax also affects the way the all columns wildcard in the **SELECT** list is processed. If the **USING** syntax is used, the columns listed in the **USING** list appear only once in the **SELECT** list. That is, for each column specified in the **USING** clause, a single column will appear in to be **SELECT**-ed, not all instances of the related columns in all of the joined tables.

The CROSS JOIN

411

The **CROSS JOIN** is a join operation that computes a Cartesian product. The general syntax for a **CROSS JOIN** is:

```
<table-reference> CROSS JOIN <table-reference>
```

Because the **CROSS JOIN** is intended especially to obtain a Cartesian product, the syntax model does not contain a join-condition. This makes most sense: applying a condition to a **CROSS JOIN** would be functionally equivalent to an **INNER JOIN**. It would not really make sense to use the **CROSS JOIN** syntax to do that; one should use the **INNER JOIN** syntax instead.

NOTE: In MySQL, the **CROSS JOIN** syntax is actually equivalent to the **INNER JOIN** syntax. That is, the **CROSS JOIN** syntax will accept and process an **ON** clause with a join condition. Conversely, the **INNER JOIN** syntax allows the **ON** clause and join condition to be omitted.

However, that fact that you can do this, does not mean that it is necessarily a good idea. For maximum clarity, as well as portability, you are strongly advised to always include an **ON** clause in an **INNER JOIN** clause, and to always omit the **ON** clause in a **CROSS JOIN** clause.

The **CROSS JOIN** has the same effect as the comma join in the sense that they both produce a Cartesian product from the tables appearing on the left and right side of the phrase. However, there is a difference in precedence. Of all different join syntax symbols, the comma has the lowest precedence. All other join syntax phrases have the same precedence.



The equi-join

412

The term equi join denotes any join operation that relates the joined tables by doing an equality comparison of column pairs of the joined tables. So, the term equi-join conveys something about the join condition, rather than the way the rows from the joined tables are combined.

It does not matter how exactly the comparison is denoted: the **NATURAL JOIN**, and the named columns join (with **USING**) are all particular forms of the equi join, even though they do not literally include an equals operator (=) in the join condition.

The equi join is by far the most used type of join. In this chapter, every join we show so far that used a join condition was an equi join.

The non-equi-join

Of course, if the equi join is any join that uses a join condition that performs an equality comparison of columns, then the non-equi join must be any join that does not use an equality comparison. For example, suppose we would want to look for all cities that are **not** the capital city of their respective country:

```
mysql> SELECT CountryName, CityName
-> FROM SimpleCity INNER JOIN SimpleCountry
-> ON CountryCode = Code
-> AND CityID != Capital;
```

Here, the join condition does not consist only of equality comparisons: the expression `CityID != Capital` explicitly requires the `CityID` column not to be equal to the `Capital` column. Hence, the entire join is considered to be a non-equi join, even though the join condition contains another expression that does test for equality.

The BETWEEN...AND join

The **BETWEEN...AND** join is a special class of non-equi join that uses the **BETWEEN...AND** comparison operator in its join condition.

This type of join is often used to lookup data that is dependent upon a particular range of values. For example, assume that employee bonuses are defined for particular salary ranges. The following query then looks up the bonus per employee:

```
mysql> SELECT Employee.ID, Bonus.Amount
-> FROM Employee INNER JOIN Bonus
-> ON Employee.Salary BETWEEN
->           Bonus.LowerSalaryBound
->           AND Bonus.UpperSalaryBound;
```

This construct is interesting in that the tables `Employee` and `Bonus` are related to each other (there is a one to many relationship from `Bonus` to `Employee`) although there is no foreign key (and consequently, no foreign key constraint).



The autojoin or self-join

The terms *auto-join* is interchangeable with *self-join*. It is an ordinary join between two instances of the same table. The classical example is a hierarchy of employees. The **EMPLOYEE** table has an **ID** (which identifies the employee) and an optional **BOSSID** column which is a foreign key to the **ID** column of the employee table itself. Assuming such a structure, the following query retrieves the names of the employees and the name of their respective boss:

```
mysql> SELECT Employee.LastName, Boss.LastName
-> FROM Employee INNER JOIN Employee AS Boss
-> ON Employee.BossID = Boss.ID;
```

In virtually all cases, the auto-join requires at least one of the tables to receive a table alias to prevent ambiguity. Here, the alias is chosen to expressly denote the role of the second **Employee** table in the query.

13.8 Joins in UPDATE and DELETE statements

413 In MySQL, the join operation is not confined to **SELECT** statements. Joins can also appear in **DELETE** and **UPDATE** statements. Such an **UPDATE** or **DELETE** statement can be used to modify the contents of multiple tables in a single statement.

UPDATE and **DELETE** statements that contain a join are usually called *multi-table UPDATE* and **DELETE** statements respectively. Multi-table **UPDATE** and **DELETE** statements are a non-standard feature.

13.8.1 Multi-table UPDATE syntax

414 The syntax for the multi-table **UPDATE** syntax is shown below:

```
UPDATE <joined-tables>
SET    <column-assignments>
[WHERE <condition>]
```

Here, the **<joined-tables>** part can be any form of join operation discussed in the earlier sections of this chapter. The **<column-assignments>** is a comma separated list of column assignments. One such value assignment has the following form:

```
<column-reference> = <value-expression>
```

The **<column reference>** must be reference to a column of one of the tables used in the **<joined-tables>** part. The **<value-expression>** can be any valid value expression, such as a literal, a function call, another column reference and so on.

The **WHERE** clause is optional, and has the usual effect of limiting the operation to only those rows that satisfy the **<condition>**.



415

For example, the following statement will add 20,000 to the population of 'Rio de Janeiro' as well as to the total population of 'Brazil':

```
mysql> UPDATE City
-> INNER JOIN Country
-> ON CountryCode = Code
-> SET City.Population = City.Population + 20000,
-> Country.Population = Country.Population + 20000
-> WHERE Country.Name = 'Brazil'
-> AND City.Name = 'Rio de Janeiro';
```

Order by an limit

The **ORDER BY** and **LIMIT** clauses can be used in the single table **UPDATE** syntax. However, they are not allowed in the multi-table **UPDATE** syntax. Any attempt to use an **ORDER BY** or **LIMIT** clause in a multi-table **UPDATE** statement results in a syntax error.

416

13.8.2 Multi-table DELETE syntax

There are two alternative multi-table **DELETE** syntaxes. Here is a model of the first form of the syntax:

```
DELETE <table-list>
FROM <joined-tables>
[WHERE <condition>]
```

Here follows a model of the second form of the syntax:

```
DELETE
FROM <table-list>
USING <joined-tables>
[WHERE <condition>]
```

In both cases, the **<table-list>** is a comma-separated list of table names from which rows will be deleted. The **<joined-tables>** part and **WHERE** clause together specify exactly which rows are to be deleted.

The **<joined-tables>** part can be any form of join operation discussed in the first sections of this chapter. The **WHERE** clause is optional and has the usual effect of singling out only those rows for which the **<condition>** is satisfied.

The tables that appear in the **<table-list>** must appear somewhere in the **<joined-tables>** part. However, not all tables that make up the **<joined-tables>** part need to appear in the **<table-list>**.



For example, the following statement removes the country with the code 'NLD' as well as all its cities and languages from the `world` database:

```
mysql> DELETE      Country, City, CountryLanguage
      -> FROM        Country
      -> LEFT JOIN   City
      -> ON          Code = City.CountryCode
      -> LEFT JOIN   CountryLanguage
      -> ON          Code = CountryLanguage.CountryCode
      -> WHERE       Code = 'NLD';
```

Table aliases

The `<table-list>` need not be a list of literal table names; rather, it is a list of references to the tables used to construct the `<joined-tables>` part. If the `<joined-tables>` introduces table aliases to refer to a particular table, then these aliases must be used in the `<table-list>` as well, not the literal table names.

The following example is a modified version of the previous statement, illustrating the usage of table aliases in the multi-table `DELETE` syntax:

417

```
mysql> DELETE      Co, Ci, Cl
      -> FROM        Country           AS Co
      -> LEFT JOIN   City              AS Ci
      -> ON          Code = City.CountryCode
      -> LEFT JOIN   CountryLanguage   AS Cl
      -> ON          Code = CountryLanguage.CountryCode
      -> WHERE       Code = 'NLD';
```

Listing only particular tables

As explained earlier, the `<table-list>` need not contain all tables used in the `<joined-tables>` part. For example, the following statement deletes the cities and languages for the country with the code 'NLD', but retains the corresponding row in the `Country` table:

```
mysql> DELETE      Ci, Cl
      -> FROM        Country           AS Co
      -> LEFT JOIN   City              AS Ci
      -> ON          Code = City.CountryCode
      -> LEFT JOIN   CountryLanguage   AS Cl
      -> ON          Code = CountryLanguage.CountryCode
      -> WHERE       Code = 'NLD';
```

Note that here, the `<table-list>` only lists the `Ci` and `Cl` table aliases but is missing the `Co` alias that does appear in the `<joined-tables>` part.



An attempt to list a table in the *<table-list>* that does not appear in the *<joined-tables>* part leads to a syntax error. For example, this statement:

```
mysql> DELETE      Country
      -> FROM        City
      -> LEFT JOIN   CountryLanguage
      -> ON          City.CountryCode = CountryLanguage.CountryCode
      -> WHERE       City.CountryCode = 'NLD';
```

causes the following error:

```
ERROR 1109 (42S02): Unknown table 'country' in MULTI DELETE
```

This is to be expected, as the **Country** table is listed in the *<table-list>*, although it never appeared in the *<joined-tables>* part.

Order by an limit

The **ORDER BY** and **LIMIT** clauses can be used in the single table **DELETE** syntax. However, they are not allowed in the multi-table **DELETE** syntax. Any attempt to use an **ORDER BY** or **LIMIT** clause in a multi-table **DELETE** statement results in a syntax error.

13.8.3 Using multi-table UPDATE and DELETE statements?

There are a few cases in which multi-table **UPDATE** and **DELETE** statements may be advantageous:

418

- Operations on multiple tables can be packed into one statement, decreasing the number of roundtrips to the server. There are other ways to achieve this, such as stored routines. However, multi-table delete and update statements have been supported in MySQL since version 4.0
- Sometimes, poorly performing single table **UPDATE** and/or **DELETE** statements involving subqueries can be rewritten to an equivalent multi-table statement. In many cases, performance can increase considerably as compared to a statement containing a subquery.
- Logical grouping of related statements. Sometimes it's possible to group multiple **UPDATE** or **DELETE** statements that need to be executed as one unit into a respective multi-table **UPDATE** or **DELETE** statement. This can for example be used to emulate a cascading update or delete rule like you have in foreign key constraints.



However, when using multi-table **DELETE** and **UPDATE** statements, there are a few things to look out for:

419

- Multi-table **DELETE** and **UPDATE** statements do not automatically guarantee atomic execution. That is, the operations on the rows of the join result are implemented as individual operations on the rows that together make up the result row in the underlying tables. If an error occurs while performing one such individual row operation, the corresponding rows from the other tables that make up the row in the join result are not automatically undone. Atomicity is guaranteed as usual if all tables used in the multi-table **UPDATE** or **DELETE** statement are transactional.
- Multi-table **DELETE** and **UPDATE** statements do not support **ORDER BY** and **LIMIT**. This is a problem in case you need the row-level operations to be executed in a particular order to prevent errors due to constraint violations.
- Cascading delete and update rules for InnoDB foreign key constraints may interfere with the order in which the rows are accessed by the multi-table statement
- The multi-table syntax is non-standard. This may be a problem in case SQL portability is important.





InLine Lab 13-G

In this exercise you will use the methods covered in this chapter for the **UPDATE** and **DELETE** with multiple tables. This will require a MySQL command line client and the **world** database.

Step 1. Review exiting data

1. Execute the following statement to prepare for an update;

```
mysql> SELECT * FROM City WHERE Name='Casablanca'\G  
mysql> SELECT * FROM Country WHERE Code='MAR'\G
```

Note the **Code** and **CountryCode** information as well as the country **Name** and city **Name**, as they are now;

```
***** 1. row *****  
ID: 2485  
Name: Casablanca  
CountryCode: MAR  
District: Casablanca  
Population: 2940623  
1 row in set (#.# sec)
```

... and ...

```
***** 1. row *****  
Code: MAR  
Name: Morocco  
Continent: Africa  
...  
Capital: 2486  
Code2: MA  
1 row in set (#.# sec)
```



Step 2. Update the country name that contains the city of Casablanca

1. Make an update so that the name of the country associated with the city of 'Casablanca', will also be named 'Casablanca'.

```
mysql> UPDATE Country INNER JOIN City
-> ON Country.Code = City.CountryCode
-> SET Country.Name = City.Name
-> WHERE City.Name = 'Casablanca';
```

2. Confirm the country name change:

```
mysql> SELECT * FROM Country WHERE Code='MAR'\G
***** 1. row ****
      Code: MAR
      Name: Casablanca
    Continent: Africa
...
      Capital: 2486
      Code2: MA
1 row in set (#.## sec)
```

Step 3. Delete the Country and City record that contains the city of Casablanca

1. Make an update so that the name of the country associated with the city of 'Casablanca', will also be named 'Casablanca'.

```
mysql> DELETE Country, City
-> FROM Country INNER JOIN City
-> ON Country.Code = City.CountryCode
-> WHERE City.Name = 'Casablanca';
```

2. Confirm the country and city associated with 'Casablanca' was deleted:

```
mysql> SELECT * FROM Country WHERE Code='MAR'\G
Empty Set (#.## sec)

mysql> SELECT * FROM City WHERE Name='Casablanca'\G
Empty Set (#.## sec)
```





Further Practice

In this exercise, you will execute multiple table **Joins** using the **world** database tables.

420

1. What languages are spoken in the country of Sweden?
2. List the name of the country which has the largest number of cities.
3. List the name of the countries that have one or more cities with more than 7 000 000 inhabitants.
4. Which countries do *not* have any spoken languages?
5. Display a list of countries without a corresponding capital in the **City** table.
6. List the countries where more than 80% of the population lives in the cities of the **City** table.
7. Using one statement, delete the country ‘Zambia’, including all its cities and languages.



13.9 Chapter Summary

421

This concludes the chapter on joining tables in MySQL. The following things were covered:

- The concept of joining tables
- Construction and properties of the Cartesian product
- The syntax and application of different join types.
- Using qualified column references and table aliases to avoid ambiguity
- To join a table to itself
- Multi-table **UPDATE** and **DELETE** statements



14 SUBQUERIES

14.1 Learning Objectives

423

This chapter introduces MySQL subqueries (sometimes referred to as nested queries). In this chapter, you will learn:

- Nest a query inside another query
- Place the subquery accurately within a query according to the type of table results required
- Understand and use the proper category of subquery per need
- Employ proper SQL syntax when placing subqueries within a statement
- Convert subqueries into joins



14.2 Overview

424

A subquery is a query (that is, a **SELECT** expression) that appears between parenthesis as part of another SQL statement (as opposed to being itself a statement).

The following example illustrates a simple subquery. We use the two tables `Country` and `CountryLanguage` from the `world` database to find the languages spoken in Finland:

```
mysql> SELECT Language          -- outer SELECT expression
      -> FROM CountryLanguage
      -> WHERE CountryCode = (
      ->     SELECT Code           -- left parenthesis - starts subquery
      ->         FROM Country
      ->     WHERE Name = 'Finland'
      -> );                      -- right parenthesis - ends subquery
```

In this example, the subquery is the parenthesized **SELECT** expression (`FROM Country`) that appears in the **WHERE** clause of the initial **SELECT** expression (`FROM CountryLanguage`) that makes up the 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*, *containing query* or *enclosing query*. For example, the query `SELECT...FROM CountryLanguage WHERE ...` is the outer query. This term is most appropriate when the subquery appears as part of another **SELECT** expression or as part of a **INSERT**, **UPDATE** or **DELETE** statement.

If the subquery is part of an expression, then this expression can likewise be referred to as the *outer expression*, *containing expression* or *enclosing expression*. In the example, the equality comparison `CountryCode = (SELECT ... 'Finland')` is the enclosing expression.

To understand the example statement and its result, it's best to pretend the statement is executed in two steps.

425

First, consider the subquery. The subquery retrieves the value of the `Code` column from the `Country` table for the country called "Finland". This effectively evaluates to a single country code, 'FIN':

```
mysql> SELECT Code FROM Country WHERE Name = 'Finland';
+---+
| Code |
+---+
| FIN |
+---+
```

(As we shall see later on, subqueries are not limited to evaluating to a single value, but for now this illustrates the principle of executing a nested **SELECT** expression as part of a larger SQL statement.)



When we know the result of the subquery, it can be substituted by its result so that the outer query can effectively be reduced to:

```
mysql> SELECT Language
-> FROM CountryLanguage
-> WHERE CountryCode = 'FIN';
```

This brings us to the result of the initial subquery example: all languages that are spoken in the country with the name Finland:

Language
Estonian
Finnish
Russian
Saame
Swedish

Why use a subquery?

Subqueries allow data from multiple tables to be combined in a single query. There are other methods that are capable of combining data from multiple tables such as joins, but subqueries are generally considered to be more straightforward and easier to read.

Similar to the English language subordinate clause, a subquery can be read from the inside out. In other words, you can focus on the subquery results first, then on the outer query. This adds a definite structure to SQL statements, which is conveniently reflected in the name of the language, “Structured Query Language”.

14.3 Types of subqueries

A Subquery can be categorized according to what function its result set has with respect to the containing expression. This allows us to discern the following types of subqueries:

- *Scalar subqueries*: the subquery is treated as single value
- *Row subqueries*: the subquery is treated as a single row
- *Table subqueries*: the subquery is treated as a (readonly) table that contains zero or more rows

The term *column subquery* is used to indicate a special case of a table subquery that selects only a single column. However, they do not represent a truly different class of subquery.



14.3.1 Scalar subqueries

426

Scalar subqueries act as simple, singular value expressions (scalars). To evaluate a scalar subquery, its query expression is executed and expected to retrieve at most one row having exactly one column. If the row is retrieved, its column value is treated as a scalar value. If no row is retrieved, the subquery evaluates to the **NULL** value. So, a scalar subquery always evaluates to a single value expression, even if the parenthesized query expression fails to retrieve a row.

We've already seen an example of a scalar query in the introductory section, where a subquery was used to retrieve the country code of Finland for use in the **WHERE** clause of the containing query. Scalar subqueries are by no means confined to the **WHERE** clause.

As far as the containing expression is concerned, scalar subqueries have the same status as literals, function calls, column references and the like. Therefore, they can appear pretty much throughout the enclosing SQL statement, for example in the **SELECT** list, as arguments to functions and as operands to all the regular arithmetic and comparison operators. For example, the following query retrieves each country and the population as a percentage of the entire world population:

```
mysql> SELECT Country.Name,
->           100 * Country.Population /
->           (SELECT SUM(Population) FROM Country) AS pct_of_world_pop
->     FROM Country;
```

Here the subquery (**SELECT SUM(Population) FROM Country**) is evaluated to calculate the total population world-wide, and this is then used to calculate the population percentage for each individual country.

A semantical error occurs when the result set of the parenthesized **SELECT** expression of a scalar subquery happens to contain more than one column. In this cases, a runtime error occurs, explaining the problem:

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

Another semantical error occurs when the parenthesized **SELECT** expression of a scalar subquery happens to retrieve more than one row. This also leads to a runtime error indicating the problem:

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

On the other hand, no such problem occurs if the parenthesized query expression happens to yield the empty set. In that case, the subquery is considered to evaluate to **NULL**:

```
mysql> SELECT (SELECT Name FROM world.Country LIMIT 0);
+-----+
| (SELECT Name FROM world.Country LIMIT 0) |
+-----+
| NULL |
+-----+
```



428

One particular case where scalar subqueries are very useful is when calculating an aggregates of multiple unrelated details. For example, if we want to calculate per country the number of cities as well as the number of languages, we can use a query like this:

```
mysql> SELECT Name,
->       (SELECT COUNT(*) FROM City
->        WHERE CountryCode = Code) AS Cities,
->       (SELECT COUNT(*) FROM CountryLanguage
->        WHERE CountryCode = Code) AS Languages
->     FROM Country;
```

14.3.2 Row subqueries

429

Row subqueries are treated as a single row containing at least 2 columns. A row subquery is a special case of a so-called *row-constructor*. (A row constructor is an expression that evaluates to a single row having two or more columns.)

If the parenthesized **SELECT** expression retrieves a single row, the row subquery evaluates to that row. If the **SELECT** expression does not retrieve any rows, the subquery evaluates to a single row having **NULL** column values. So even though the **SELECT** expression retrieved an empty set, the entire parenthesized expression still evaluates to one row.

Row subqueries (and other row constructors) can be used as operands for the equality operators `=`, `<>`, `!=`, `<=>` and the comparison operators `<`, `>`, `>=`, and `<=`, provided both operands are row constructors having the same number of columns. When used with row constructors, these operators are evaluated by performing a pairwise application of the operator on the corresponding individual (scalar) column values.

For the equality operators (`=` and `<=>`), all pairwise comparisons need to evaluate to true in order for the expression to be true. In other words, two rows are equal only if all corresponding column values are equal. For the inequality operators (`!=` and `<>`), only one pairwise comparison needs to be true in order for the expression to be true. In other words, two rows are not equal unless all corresponding column values are equal, so if one one column pair is not equal, the rows are not equal.

For the comparison operators (`>`, `<`, `>=` and `<=`), it is slightly more complicated. Here, the order of the columns has significance (in addition to the column values). For example, the row `(SELECT 2, 1)` is considered to be larger than the row `(SELECT 1, 100)` because 2 is larger than 1. The comparison for the second column pair (1 and 100 respectively) is not significant, because the first row was already found to be larger than the second one. If the columns would have been reversed, it would have been the opposite: `(SELECT 1, 2)` is considered to be smaller than `(SELECT 100, 1)` because 1 is smaller than 100.



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

430

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

In the same way, two row subqueries can be compared to each other:

```
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   |
+-----+
```

The following example illustrates that a row subquery always evaluates to one row, even if the parenthesized query expression evaluates to the empty set:

431

```
mysql> SELECT (NULL, NULL) <=> (SELECT ID, Name FROM City LIMIT 0);
+-----+
| (NULL, NULL) <=> (SELECT ID, Name FROM City LIMIT 0) |
+-----+
|                                         1   |
+-----+
```

However, if the parenthesized expression of a row subquery happens to retrieve more than one row, a runtime error occurs:

```
mysql> SELECT ('London', 'GBR') = (SELECT Name, CountryCode
   ->                               FROM City);
ERROR 1242 (21000): Subquery returns more than 1 row
```

An error also occurs if the compared row constructors do not have the same number of columns:

```
mysql> SELECT (456, 'London', 'GBR') = (SELECT Name, CountryCode
   ->                               FROM City);
ERROR 1241 (21000): Operand should contain 3 column(s)
```



(The row literal is the first operand, and its structure has set the expectation for that of the second operand, the subquery. Because the subquery selects only two columns, the error message indicates that it expected another column.)

14.3.3 Table subqueries

432

Table subqueries act as (readonly) tables. These table subqueries evaluate to a result set containing zero or more rows with one or more columns. They can appear in two different contexts:

- In the `FROM` clause of an enclosing query.
- As right-hand operands to the logical operators `IN` and `EXISTS`, or as right hand operator to a regular comparison operator (`=`, `!=`, `<>`, `<`, `>`, `<=` or `>=`) quantified with `ALL`, `ANY` and `SOME`.

It is important to realize that the number of selected columns or the number of returned rows does not affect the subquery's status as a table subquery. The fact that the subquery appears in the `FROM` clause, or that it is used as an operand with one of these operators causes it to be treated as a (readonly) table. By definition, this is what makes the subquery a table subquery. There is for example no such thing as a scalar subquery or a row subquery in the `FROM` clause.

Subqueries in the `FROM` clause

433

The result set of a subquery in the `FROM` clause is treated in the same way as results retrieved from base tables or views that are referred to in the `FROM` clause. The following example illustrates the usage of a subquery in the `FROM` clause:

```
mysql> SELECT *
-> FROM (SELECT Code, Name
->         FROM Country
->         WHERE IndepYear IS NOT NULL) AS IndependentCountries;
```

When the statement is executed, the subquery will be executed first, and its result set will be temporarily stored and used when executing the remainder of the statement. The subquery result is discarded after the statement is executed.

In the example, a table alias is used to provide a temporary name for the subquery result. In MySQL, this table alias is required for all subqueries that appear in the `FROM` clause. Omitting the alias will result in an error:

```
ERROR 1248 (42000): Every derived table must have its own alias
```

434

Subqueries in the `FROM` clause are especially useful for calculating aggregates of aggregates. In the following query, we find the average of the sums of the population of each continent:

```
mysql> SELECT AVG(cont_sum)
-> FROM (
->         SELECT Continent, SUM(Population) AS cont_sum
->         FROM Country
->         GROUP BY Continent
->     ) AS t;
```



The table subquery is evaluated separately from the outer query, calculating the **SUM** of the Population per continent. Then, the resulting rows (one for each continent) are aggregated again by the outer query because of the application of the **AVG** function.

14.4 Table subquery operators

435

In this section, we'll discuss the syntax and usage of table subqueries as right hand operand for a number of comparison operators. Only specific operators can accept a table subquery as right hand operand:

- The logical operators **IN** and **EXISTS**
- The regular comparison operators **=**, **!=**, **<**, **>**, **>=** and **<=** (but not **<=>**), provided that one of the quantifiers **ALL**, **ANY** or **SOME** is used to specify how the operator must be applied to the subquery.

Just like the regular comparison operators, all these return a boolean result. There is nothing particularly special about these comparison operators apart from the fact that they require a table subquery at their right hand side.

Although these operators can be used in any place throughout the statement where one can normally place a scalar expression, the usage of a table subquery as a right hand operand for these operators is confined mostly to the **WHERE** clause.

436

14.4.1 The IN operator

A table subquery may be used as the right hand operand for the **IN** operator. In this case, the **IN** operator evaluates to true if there is at least one occurrence in the result set derived from the subquery that is equal to the left hand operand. If there is known to be no occurrence equal to the left hand operand in the result set formed by the right hand operand, **IN** evaluates to false.

If the table subquery selects only a single column (column subquery) the left hand operand must be a scalar value expression. The following example illustrates this usage:

```
mysql> SELECT *
->   FROM City
-> WHERE CountryCode IN (
->   SELECT Code
->     FROM Country
->   WHERE Continent = 'Asia'
-> );
```

The previous example query selects all cities located in Asian countries. First, the subquery retrieves all possible country codes for Asian countries, and with the **IN** operator, the outer query on the `City` table checks if the `CountryCode` of the city matches one of these Asian country codes.

If the table subquery selects more than one column, the left-hand operand must be a row constructor. In this case, the left hand operand row is checked for equality using pairwise column comparisons with the rows in the result set.



437

The following example illustrates pairwise column comparison usage:

```
mysql> SELECT *
-> FROM City
-> WHERE (CountryCode, Name) IN (
->           SELECT Code, Name
->           FROM Country
->           WHERE Continent = 'Asia'
->         );
```

The subquery is executed to retrieve the codes and names for all Asian countries. The outer query retrieves cities and compares their country code and city name to each of the country code and name rows. This way the statement returns only those `City` rows for which the name is equal to the name of its respective country.

Using NOT IN

The `IN` operator can be negated by prefixing the `IN` keyword with the `NOT` keyword. To all intents and purposes this can be considered to be a distinct `NOT IN` operator. `NOT IN` evaluates to TRUE in case the result set is known not to contain an entry that is equal to the left operand. If the result set does contain an entry equal to the left hand operand, `NOT IN` evaluates to FALSE.

438

The IN operator and the NULL value

There are cases where it may be impossible to determine whether the result set formed by the right hand operand contains an occurrence that is equal to the left hand operand. In this case, `IN` evaluates to `NULL`. There are two different scenarios in which `IN` can evaluate to `NULL`:

- If the left hand operand is `NULL` and the result set formed by the right hand operand is not empty, `IN` evaluates to `NULL`. Because `NULL` cannot be compared to any value, it cannot be determined whether it occurs in the result set derived from the subquery. However, If the result set is empty, then `IN` always evaluates to false because by definition, there can be no occurrence equal to the left operand in this case.
- If the result set is not empty and no occurrence is found equal to the left operand, then `IN` will evaluate to `NULL` in case the result set contains at least one row for which at least one of the columns is `NULL`.

439

14.4.2 The EXISTS Operator

The `EXISTS` operator accepts a single right hand argument which must be a table subquery. If the subquery result set contains at least one row, then the `EXISTS` expression evaluates to true. It returns false in all other cases.

For example, the following query can be used to retrieve all cities that are the capital of some country:

```
mysql> SELECT *
-> FROM City
-> WHERE EXISTS (
->   SELECT NULL
->   FROM Country
->   WHERE Capital = ID);
```



Here, the outer query retrieves cities. In the **WHERE** clause, the **EXISTS** operator is used to find out if there is a country of which the city happens to be the capital.

Note that the result of the **EXISTS** operator is in no way influenced by the actual values selected by the table subquery. In the example query, the table subquery selects the **NULL** value, but it could just as well have been a literal value or a column expression: as far as the **EXISTS** operator is concerned, it simply doesn't matter. The only thing that affects the evaluation of the **EXISTS** operator is whether the query expression that makes up the table subquery retrieves a row, regardless of the selected columns or their values.

440

Using NOT EXISTS

The effect of **EXISTS** can be negated by placing the **NOT** keyword before the **EXISTS** keyword. The **NOT EXISTS** construct evaluates to TRUE only if the argument subquery evaluates to the empty set, and is FALSE otherwise.

The following query illustrates how **NOT EXISTS** may be used to find all the countries where no English is spoken:

```
mysql> SELECT *
      -> FROM Country
      -> WHERE NOT EXISTS (
      ->     SELECT NULL
      ->     FROM CountryLanguage
      ->     WHERE CountryCode = Code
      ->     AND Language = 'English');
```

Here, the outer query retrieves all countries. For each **Country** row, the corresponding rows from **CountryLanguage** are retrieved in the subquery

441

14.4.3 ALL, ANY and SOME

ALL, **ANY** and **SOME** are so-called *quantifiers*. They are not themselves operators; rather, they are used in conjunction with the regular comparison operators ($=$, \neq , $<$, $>$, \leq , and \geq) and specify how to apply the operator to compare a singular left hand operand to the result set returned by the table subquery used as the right hand operand.

The need for the quantifiers becomes clear when we consider the normal usage of the comparison operators. Normally, these operators are used to compare a single item that forms the left hand operand to another single item that makes up the right operand. In our discussion of row subqueries, we have seen already that it is perfectly possible that the operands are composite: the fact that the operands themselves consist of multiple parts does not change the fact that the operator still compares only one such item with one other such item.

In our discussion of scalar and row subqueries we have seen a few examples of the error that occurs when the parenthesized query expression that makes up the subquery happens to retrieve more than one row:

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



The problem is that the = operator cannot be used directly to compare the scalar value 'Finland' to the set of values returned by the subquery (`SELECT Name FROM world.Country`). However, we can imagine repeatedly applying the = operator (and the left hand operator) to a number of items returned by the subquery. This is exactly the purpose of the quantifiers `ALL`, `ANY` and `SOME`: they specify how to repeatedly apply the operator to compare the singular operand at the left hand side to the multiple values returned by the subquery at the right side of the operator. The effect of the different quantifiers is described below:

442

- `ALL` indicates that the comparison is true if the operator is applied to all items in the subquery result set and returns true in all cases.
- `ANY` indicates that the operator must be applied to the items in the subquery result set until at least one comparison returns true. False is returned if none of comparisons returned true.
- `SOME` is an alias of `ANY`, and has exactly the same effect. However, in some cases the statement is easier to understand for the human reader when the `SOME` keyword is used instead of the `ANY` keyword.

Quantifiers are used by placing them between the operator and the subquery:

```
<left-hand-operand> <comparison-operator> <quantifier> <table-subquery>
```

For example, the following example uses `ANY` to see if 'Finland' is a valid country name:

```
mysql> SELECT 'Finland' = ANY (SELECT Name FROM world.Country);
```

The query above will return 1 (that is, `TRUE`), because there is at least one row returned by the subquery that has the value 'Finland' in the name column. Conversely, substituting `ANY` with `ALL` will return 0 (that is, `FALSE`), because the value 'Finland' is not simultaneously equal to the names of all countries.

443

Another example:

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

The previous query retrieves countries only when the country's population exceeds the population of all of the cities.



SOME versus ANY

We just mentioned that **SOME** is an alias for **ANY**. As far as query execution is concerned, **SOME** and **ANY** are completely interchangeable. However, there are cases where **SOME** is found to be more readable than **ANY**. Consider the following query:

```
mysql> SELECT *
->   FROM City
-> WHERE Population > ANY (
->       SELECT Population
->         FROM Country
->     );
```

If we attempt to explain in plain English what this query does, we may end up with something like:

“Retrieve all cities that have a population larger than any country's population”

You may feel uncomfortable reading the English rendering of the query: how in the world can the number of inhabitants of a part of a country (like a city) ever contain more inhabitants than the country itself – let alone all of the countries?!

It may become even more mysterious when you execute the query and see its result:

ID	Name	CountryCode	District	Population
1	Kabul	AFG	Kabol	1780000
2	Qandahar	AFG	Qandahar	237500
.....
4078	Nablus	PSE	Nablus	100231
4079	Rafah	PSE	Rafah	92020

4079 rows in set (#.## sec)

We see an entirely unanticipated result – *all* of the cities are returned.

The paradox is not that hard to solve though. If we take a closer look at how the **ANY** quantifier is defined, we must conclude that our English rendering of the query is dead wrong. Instead of requiring that the city population is larger than any country's population, we should read the query as:

“For each city, check if there is any country at all for which the city's population exceeds that of the country”



A few countries don't even have cities, and their population is recorded as 0. So, in the `world` database, one can always find at least one country that has a population that is lower than that of any arbitrary city. For these occasions, `SOME` may make the query slightly more readable:

```
mysql> SELECT *
-> FROM City
-> WHERE Population > SOME (
->           SELECT Population
->             FROM Country
->           );
```

“Retrieve all cities that have a population larger than some country's population”

or:

“Retrieve all cities with a population larger than that of some countries”

444

Alternatives to ANY and ALL

We just illustrated that `ANY` can sometimes be confusing. However, `ALL` is not without issues either. In our discussion of the `ALL` quantifier we described a query using `> ALL` that retrieves countries whose population exceeds the population of all of the cities:

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

We could have chosen slightly different words to describe the effect of this particular query:

“Retrieve all countries that have a population larger than any city's population”

So first we had a SQL query with `> ANY` for which the English translation included the word “any” but which led us to believe something was done with *all* rows returned by the subquery; this time we have a SQL query with `> ALL` that is translated to English using the word “any” and correctly implies something is done with *all* rows returned by the subquery. Because queries with `ANY` and `ALL` can lead this kind of confusion, many people like to rewrite queries in order to avoid the quantifiers.



The following list shows the alternatives for a number of quantified operator expressions:

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

Comparing row constructors and table subqueries

MySQL offers only support for the quantifiers to compare scalar left hand operands to column subqueries. In theory, there is no reason why the quantifiers could not be applied for comparing row constructors to table subqueries. However, at present only scalar left hand operators are supported in MySQL.

445

14.5 Correlated and non-Correlated Subqueries

Subqueries are either correlated or non-correlated. The distinction between correlated and non-correlated subqueries is based on whether the parenthesized `SELECT` expression that makes up the subquery contains expressions derived from the containing statement. This categorization of subqueries is independent of the distinction between scalar, row and table subqueries discussed in the previous sections, which has to do with the form and function of the subquery result set.

446

14.5.1 Non-Correlated Subqueries

A subquery is *non-correlated* if the parenthesized query expression does not refer to an expression that depends on the enclosing statement. The subquery contained by the following example `SELECT` statement is a typical example of a non-correlated subquery:

```
mysql> SELECT *
    -> FROM City
    -> WHERE CountryCode IN (
    ->         SELECT Code
    ->         FROM Country
    ->         WHERE Continent = 'Europe'
    ->     );
```



Because the subquery is not dependent upon any expressions derived from outside the parenthesized query expression, its result can be computed independently from the remainder of the statement. The rows from the `City` table must each be tested against the subquery, but the result set of the subquery does not need to be rebuilt for each `City` row. Instead, the subquery result can be computed just once, and reused when processing the rows for the outer query.

Because a non-correlated subquery is self-contained, you can execute the query expression as a standalone query:

```
mysql> SELECT Code  
-> FROM Country  
-> WHERE Continent = 'Europe';
```

The fact that the query expression of the subquery is itself a valid `SELECT` statement proves the subquery did not contain references to expressions outside the scope of the subquery.

447

14.5.2 Correlated Subqueries

A subquery is a *correlated* subquery if it contains one or more expressions that are derived from a part of the statement that appears outside of the parenthesis that demarcate the subquery. Here's a typical example of a correlated subquery:

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

The parenthesized query contains the expression `Code`. However, the `City` table that is queried by the subquery does not contain a `Code` column. To resolve whatever the expression `Code` refers to, the parts of the statement that appear outside the subquery's parenthesis must be examined.

In this case, the remainder of the statement is the outer query of the subquery. The outer query queries the `Country` table which indeed contains a `Code` column. So, the `Code` expression used in the subquery must refer to the `Code` column of the `Country` table that stems from the outer query.

The value of the `Code` column may be different for each row of the `Country` table. Because the subquery requires the `CountryCode` column of the `City` table to be equal to the `Code` column of the `Country` table, the result of the subquery is potentially different too for each distinct value of the `Code` column. So, for each `Country` row, the subquery must be reevaluated to get the result that corresponds to the current value of the country's `Code` column.



Because the subquery can be evaluated only when the value of the Code column is known already, the subquery is said to be dependent upon the outer query. As a direct consequence, the query expression that makes up the subquery cannot be executed as a standalone query:

```
mysql> SELECT NULL
-> FROM City
-> WHERE CountryCode = Code;
ERROR 1054 (42S22): Unknown column 'Code' in 'where clause'
```

448

Scope

Let's examine another subquery example:

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

Here, the subquery requires the Name column to be equal to 'Belgium'. Both the City table used in the outer query as well as the Country table used in the subquery contain a Name column. However, this ambiguity does not pose a problem because column references are resolved using the nearest possible scope. So, in this case, the Name column appears in the subquery. The nearest possible source for the Name column would be a table appearing in the **FROM** list of the subquery's query expression. In this case, the **FROM** list only contains the Country table which does indeed contain a Name column. This means that there is no need to look further: the Name column can be resolved in the local scope of the parenthesized query expression. Therefore, the previous query is equivalent to:

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

Correlated Subqueries in the FROM clause

Correlated subqueries can appear in any context where a subquery is valid, with one exception. A subquery that directly appears in the **FROM** clause cannot be a correlated subquery. Any attempt to use a correlated subquery results in an error:





InLine Lab 14-A

In this exercise you will use the methods covered in this chapter for placing subqueries in a **SELECT** column designation and the **FROM** clause. This will require a MySQL command line client and the world database.

Step 1. Correlated subquery

1. Perform a query with a nested, correlated query (subquery) that retrieves the country names in the world that are located in the 'Nordic Countries' regions, with the number of cities per country. The outer query is done for you below. You must create the subquery and run the completed statement:

```
mysql> SELECT Country.Name, (SELECT count(*)
->                               FROM City
->                               WHERE CountryCode = Country.Code) AS CityCount
-> FROM   Country
-> WHERE  Region = 'Nordic Countries';
```

Returns list of the seven countries in the Nordic Region and the number of cities associated with each country:

Name	CityCount
Tórshavn	1
Longyearbyen	1
Reykjavík	1
Oslo	5
Stockholm	15
Helsinki [Helsingfors]	7
København	5



Step 2. Non-correlated subquery

1. Perform a query with a nested, non-correlated query (subquery) that retrieves the average of the continental populations. The outer query is done for you below. You must create the subquery and run the completed statement.

```
mysql> SELECT AVG(Totals)
->   FROM (SELECT SUM(Population) AS Totals
->          FROM Country
->         GROUP BY continent) AS temp;
```

Returns a single value of the average of the sum of the continental populations:

+-----+
avg(totals)
+-----+
868392778.5714
+-----+





InLine Lab 14-B

In this exercise you will use the methods covered in this chapter for placing subqueries in the **WHERE** clause. This will require a MySQL command line client and the world database.

Step 1. Nested Query

1. Perform a query with a nested query (subquery) that retrieves the cities in the world with populations larger than New York city, in order by population:

```
mysql> SELECT      Name
->   FROM      City
->   WHERE      Population > (SELECT Population
->                                FROM      City
->                                WHERE     Name = 'New York')
->   ORDER BY  Population;
```

Returns list of the country names with populations larger than that of New York:

Name
Moscow
...
Mumbai (Bombay)

9 rows in set (#.# sec)

Step 2. Distinct languages removed

1. Retrieve the distinct languages spoken on the continent of 'Africa':

```
mysql> SELECT DISTINCT Language
->   FROM      CountryLanguage
->   WHERE     CountryCode IN (
->           SELECT Code
->             FROM  Country
->            WHERE Continent='Africa'
->       );
```

Returns a list of languages spoken in Africa:

Language
Ambo
Chokwe
...



14.5.3 Other Subquery Uses

Use of subqueries is not limited to **SELECT** statements. It is often useful to delete rows based on whether they match rows in another table. Also, you can update rows in one table using the contents of rows in another table.

For example, this statement will remove (from the `City` table) all the cities located in countries where the life expectancy is less than 70 years;

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

In the following example, a subquery is used in the **SET** clause of an **UPDATE** statement to set the `CountryPopulation` to the **SUM** of the Population of its corresponding cities:

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

14.6 Converting Subqueries to Joins

449

In general, MySQL can optimize join queries better than subqueries. Therefore, a join is sometimes more efficient than a subquery. For instance, if a **SELECT** written as a subquery takes a long time to execute, you can try writing it as a join to see if it performs better.

14.6.1 Rewriting IN and NOT IN

Specifically, a subquery that finds matches between tables often can be rewritten as a join.

Rewriting IN to INNER JOIN

Consider for example a query that finds all countries where a particular language is spoken:

```
mysql> SELECT Name
      -> FROM Country
      -> WHERE Code IN (
      ->           SELECT CountryCode
      ->             FROM CountryLanguage
      ->           WHERE Language = 'Spanish'
      ->       );
```



This can be rewritten to an **INNER JOIN** using the following steps:

- Move the table used in the subquery to the **FROM** clause of the outer query using **INNER JOIN**:
- Move the **IN** comparison and the subquery's **SELECT** list from the **WHERE** clause to the **ON** clause of the join
- Rewrite the **IN** to an equals (=) operator
- Move the subquery's **WHERE** clause to the **WHERE** clause of the join query

```
mysql> SELECT      Name
      -> FROM        Country
      -> INNER JOIN  CountryLanguage
      -> ON          Code = CountryCode
      -> WHERE       Language = 'Spanish';
```

450

When executing these queries, you will notice that the results are identical. In some cases, the join query may return multiple copies of the same row as compared to the original query containing the **IN** subquery (although the collection of distinct rows is identical). As an example, consider the previous query, but omit the **WHERE Language = 'Spanish'** criterion:

```
mysql> SELECT Name
      -> FROM  Country
      -> WHERE Code IN (
      ->           SELECT CountryCode
      ->           FROM   CountryLanguage
      ->         );
```

In case of the subquery, each **Country** row is returned at most once: the **IN** subquery only tests if the value of the country's **Code** column happens to be recorded in the **CountryLanguage** table, but this does not change the fact that each individual **Country** row is processed only once. In case of the join query however, the rows from the **Country** and **CountryLanguage** tables are *combined*, causing each **Country** row to appear just as often as there are corresponding **CountryLanguage** rows. Of course, this is not apparent from the query result, because that only contains the country's **Name** column in the **SELECT** list and does thus not reveal that the rows are different with regard to the **Language** column.

To rewrite such a subquery to a join, follow the previously described process, but also add the **DISTINCT** keyword to the **SELECT** list in order to eliminate the duplicate rows:

```
mysql> SELECT DISTINCT Name
      -> FROM  Country
      -> INNER JOIN CountryLanguage
      -> ON    Code = CountryCode;
```



451

Rewriting NOT IN to an outer join

In much the same way as we can rewrite a subquery with `IN` to an `INNER JOIN`, we can rewrite a subquery using `NOT IN` to a `LEFT` or `RIGHT JOIN`. Consider the following query that uses `NOT IN` and a subquery to find cities that are not the capital city of any country:

```
mysql> SELECT City.*  
-> FROM City  
-> WHERE ID NOT IN (  
->         SELECT Capital  
->         FROM Country  
->     );
```

This can be rewritten to a `LEFT JOIN` using the following steps:

- Move the table used in the subquery to the `FROM` clause of the outer query using `LEFT JOIN`:
- Move the `NOT IN` comparison and the subquery's `SELECT` list from the `WHERE` clause to the `ON` clause of the join
- Rewrite the `NOT IN` to an equals (=) operator
- Add a condition to the `WHERE` clause of the join query to require that there is no corresponding row in the joined table

```
mysql> SELECT      City.*  
-> FROM      City  
-> LEFT JOIN  Country  
-> ON        ID = Capital  
-> WHERE     Capital IS NULL;
```



452

14.6.2 Limitations to Rewriting Subqueries to Joins

Some subqueries can't be rewritten to joins. Other subqueries can be rewritten to joins, but may result in a query that performs worse than the original query using a subquery.

Aggregating aggregates using FROM clause subqueries

A subquery in the `FROM` clause can be conveniently used to compute an aggregate. Because the result is essentially treated as just another table, the outer query can again apply an aggregate function to the subquery result. This device allows one to calculate aggregates on aggregate data.

Earlier, we discussed a query that uses this method to calculate the average continent population. The following query is yet another example, this time using the `GROUP_CONCAT` aggregate function to obtain a hierarchical overview of countries throughout the world:

```
mysql> SELECT      GROUP_CONCAT('\n', Continent, Regions
->                      ORDER BY CAST(Continent AS CHAR(15))
->                      SEPARATOR '')                                AS Continents
-> FROM (
->     SELECT      Continent,
->                 GROUP_CONCAT('\n ', Region, Countries
->                         ORDER BY Region SEPARATOR '')           AS Regions
->                 FROM (
->                     SELECT      Continent, Region,
->                         GROUP_CONCAT('\n   ', Name
->                             ORDER BY Name SEPARATOR '')            AS Countries
->                         FROM        Country
->                         GROUP BY Continent, Region
->                     )
->                 GROUP BY Continent
->             )                                              AS Countries
->         )                                              AS Regions\G
```

Executing this yields a result like this:

```
Africa
Central Africa
Angola
Cameroon
.....
.....
Eastern Africa
British Indian Ocean Territory
Burundi
.....
Asia
Eastern Asia
...
```



Reporting aggregates of distinct child tables

Sometimes it's necessary to calculate multiple aggregates on different tables. For example, one may need to calculate the number of languages and the number of cities per country. In this particular case, it turns out that it actually is possible to do this without any subqueries at all:

```
mysql> SELECT      Country.Name,
->           COUNT(DISTINCT City.ID)      AS NumberOfCities,
->           COUNT(DISTINCT Lang.Language) AS NumberOfLanguages
->     FROM        Country
->   LEFT JOIN    City
->     ON          Country.Code = City.CountryCode
->   LEFT JOIN    CountryLanguage AS Lang
->     ON          Country.Code = Lang.CountryCode
->   GROUP BY    Country.Name;
```

However, the problem with this type of query is that it implies a (partial) Cartesian product. The **City** and **CountryLanguage** rows are both related to country, but not (directly) to each other. This means that for each country, the number of resulting rows in the intermediate result before the **GROUP BY** takes place is as large as the number of its cities times the number of its languages. For this reason, we had to use `COUNT(DISTINCT ...)` rather than just `COUNT()`; the **DISTINCT** keyword ensures that per country, only unique **City** and **Language** occurrences are counted.

There are two ways to rewrite this using subqueries, and often these will be faster than the equivalent join query. The first way to rewrite this is to replace each **JOIN** with a correlated subquery in the **SELECT** list:

```
mysql> SELECT Country.Name,
->           (SELECT COUNT(ID)
->             FROM City
->            WHERE CountryCode = Code) AS NumberOfCities,
->           (SELECT COUNT(Language)
->             FROM CountryLanguage
->            WHERE CountryCode = Code) AS NumberOfLanguages
->     FROM Country;
```

Note that the **LEFT JOINS** are removed, and that the aggregate **DISTINCT** functions are each replaced by a scalar subquery. The condition that was used as join condition is now used to bind the subquery result to the current row of the outer query, thus creating correlated subqueries. You will notice that this performs slightly better than the original JOIN query.



454

The second alternative uses the design of the original query with the **LEFT JOINS**. However, this time we use subqueries in the **FROM** clause to join to the **Country** table. The subqueries now pre-aggregate the data per country before joining. This ensures there is at most one row available to join to the **Country** row, thus avoiding the partial Cartesian product that occurred in the original join query.

```
mysql> SELECT      Country.Name,
->           IFNULL(Cities.NumberOfCities, 0),
->           IFNULL(Languages.NumberOfLanguages, 0)
->     FROM        Country
->   LEFT JOIN  (SELECT    CountryCode , COUNT(ID) AS NumberOfCities
->              FROM      City
->              GROUP BY CountryCode) AS Cities
->   ON          Country.Code = Cities.CountryCode
->   LEFT JOIN  (SELECT    CountryCode, COUNT(Language) AS
NumberOfLanguages
->              FROM      CountryLanguage
->              GROUP BY CountryCode) AS Languages
->   ON          Country.Code = Languages.CountryCode;
```

Of the three possible solutions, this one is by far the best in this case.





Quiz

In this exercise you will answer questions pertaining to Subqueries.

1. Where in an SQL statement may a scalar subquery be placed?

2. The following query selects those continents that have countries in which more than 50% of the population speak English. Is this an example of using a correlated subquery? Why or why not?

```
SELECT DISTINCT Continent
FROM Country
WHERE Code IN (SELECT CountryCode
                FROM CountryLanguage
                WHERE Language='English'
                      AND Percentage>50
            ) ;
```

-
-
-
3. The following statement uses a correlated subquery to find the South American country with the smallest population:

```
SELECT * FROM Country c1
WHERE Continent = 'South America'
AND Population = (SELECT MIN(Population) FROM Country c2
                   WHERE c2.Continent = c1.Continent);
```

In the above subquery, **c1** depends on the outer query, because the **c1** table alias is defined in the outer query. Rewrite the statement to use a non-correlated subquery.

-
-
-
4. The equal (=) can be used as a WHERE subquery construct for all subquery result types.
(True or False)





455

Further Practice

In this exercise, you will execute subqueries using the world database tables.

1. Which country has the most populous city in the world? (Use a subquery to determine the city with the maximum population, then retrieve the associated country name.)
2. Name the countries on the European continent, where Spanish is spoken. (Use a subquery to determine the countries (by code) in the world that speak Spanish, then compare it to the codes of the European continent to filter out all other continents.)
3. Find all Countries where French is spoken but not English.
4. List each Country and the population of the largest city for each country.
5. List each Country and the name of the largest city for each country.
6. Find all cities that are larger than all cities in China.
7. Find the largest city in each continent.
8. Find all city names that are present on more than 2 continents.
9. List all country names with duplicate city names.



14.7 Chapter Summary

This chapter introduced MySQL subqueries. In this chapter, you learned:

456

- Nest a query inside another query
- Place the subquery accurately within a query according to the type of table results required
- Understand and use the proper category of subquery per need
- Employ proper SQL syntax when placing subqueries within a statement
- Convert subqueries into joins





457

15 VIEWS

458

15.1 Learning Objectives

This chapter introduces Views in MySQL. In this chapter, you will learn to:

- Define views
- List the reasons for using views
- Create a view
- Check a view
- Alter and remove a view
- Set privileges for views



459

15.2 What Are Views?

A view is a database object that is defined in terms of a query (that is, a **SELECT** expression) that retrieves the data you want the view to produce. Views are sometimes called “virtual tables.” A view can be used to select from regular tables (called “base tables”) or other views. In some cases, a view is updatable and can be used with statements such as **UPDATE**, **DELETE**, or **INSERT** to modify an underlying base table.

Views provide several benefits compared to selecting data directly from base tables:

- Access to data becomes simplified:
 - A view can be used to perform a calculation and display its result. For example, a view definition that invokes aggregate functions can be used to display a summary.
 - A view can be used to select a restricted set of rows by means of an appropriate WHERE clause, or to select only a subset of a table's columns.
 - A view can be used for selecting data from multiple tables by using a join or union.

460

A view performs these operations automatically. Users need not specify the expression on which a calculation is based, the conditions that restrict rows in the **WHERE** clause, or the conditions used to match tables for a join.

- Views can be used to display table contents differently for different users, so that each user sees only the data pertaining to that user's activities. This improves security by hiding information from users that they should not be able to access or modify. It also reduces distraction because irrelevant columns are not displayed.
- Views can assist with structure changes that need to be made to tables to accommodate certain applications. The view can preserve the appearance of the original table structure to minimize disruption to other applications. For example, if you split a table into two new tables, a view can be created with the name of the original table and defined to select data from the new tables such that the view appears to have the original table structure.

461

15.3 Creating Views

To define a view, use the **CREATE VIEW** statement, which has this syntax:

```
CREATE [OR REPLACE] [ALGORITHM = <algorithm_type>]
    VIEW <view_name> [(<column_list>)]
    AS <select_expression>
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

<view_name> is the name to give the view. It can be unqualified to create the view in the default database, or qualified as *<db_name>. <view_name>* to create it in a specific database.

The *<select_expression>* is a query that indicates how to retrieve data when the view is used. The statement can select from base tables or other views. References in the **SELECT** statement to unqualified table or view names are resolved against the database that was the default database at the time the view was created. To select from a table or view in a specific database, refer to it using the *<db_name.table_name>* or *<db_name.view_name>* syntax.



Several parts of the **CREATE VIEW** statement are optional:

- The **OR REPLACE** clause causes any existing view with same name as the new one to be dropped prior to creation of the new view.
- The optional **ALGORITHM** clause specifies the processing algorithm to use when the view is invoked. If specified it must be one of:
 - **MERGE** – To execute the view, the SQL text is expanded into the SQL text of the query that references the view, thus rewriting it.
 - **TEMPTABLE** – To execute the view, the query underlying the view is executed and its results stored in a temporary table. The data is then drawn from this temporary table.
 - **UNDEFINED** – The choice for an algorithm is deferred until execution time. If possible an attempt is made to use the **MERGE** algorithm, otherwise the **TEMPTABLE** algorithm is used.
- **<column_list>** provides names for the view columns to override the default names.
- When the **WITH CHECK OPTION** clause is included in a view definition, all data changes made to the view are checked to ensure that the new or updated rows satisfy the WHERE condition in the view's query expression.. If the condition is not satisfied, the change is not accepted, either in the view or in the underlying base table.

462

Example

The following **CREATE VIEW** statement defines a simple view named CityView that selects the ID and Name columns from the City table. The **SELECT** statement shows an example of how to retrieve from the view:

```
mysql> CREATE VIEW CityView AS SELECT ID, Name FROM City;
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT * FROM CityView;
+----+-----+
| ID | Name |
+----+-----+
| 1  | Kabul |
| 2  | Qandahar |
| 3  | Herat |
```

By default, the names of the columns in a view are the same as the names of the columns retrieved by the query in the view definition. For the CityView view just defined, the query retrieves two columns named ID and Name, which also become the view column names.



To override the default view column names and provide explicit names, include a `<column_list>` clause following the view name in the `CREATE VIEW` statement. If present, this list must contain one name per column selected by the view, separating multiple names with a comma. There are some important reasons to name view columns explicitly:

463

- View column names must be unique within the view. If the columns selected by a view do not satisfy this condition, a list of unique explicit column names resolves name clashes. For example, an attempt to define a view that selects columns with the same name from joined tables fails unless you rename at least one of the columns:

```
mysql> CREATE VIEW v AS
-> SELECT Country.Name, City.Name
-> FROM Country, City WHERE Code = CountryCode;
ERROR 1060 (42S21): Duplicate column name 'Name'
```

You can always rewrite the statement and simply use column aliases to avoid the duplicate column error:

```
mysql> CREATE VIEW v AS
-> SELECT Country.Name AS CountryName, City.Name AS CityName
-> FROM Country, City WHERE Code = CountryCode;
Query OK, 0 rows affected (#.# sec)
```

By using the column list in the `CREATE VIEW` statement, you can do this without changing the actual query expression:

```
mysql> CREATE VIEW v (CountryName, CityName) AS
-> SELECT Country.Name, City.Name
-> FROM Country, City WHERE Code = CountryCode;
Query OK, 0 rows affected (#.# sec)
```

- Explicit view column names make it easier to use columns that are calculated from expressions. By default, the first 64 characters of the expression text are used as column name for such a column. Usually such identifiers are unwieldy. The following example creates a view for which the second column is created from an aggregate expression:

```
mysql> CREATE VIEW CountryLangCount (Name, LanguageCount) AS
-> SELECT Name, COUNT(Language)
-> FROM Country, CountryLanguage WHERE Code = CountryCode
-> GROUP BY Name;
Query OK, 0 rows affected (#.# sec)
```





InLine Lab 15-A

In this exercise you will use the methods covered in this chapter for using **CREATE VIEW**. This will require a MySQL command line client and the `world` database.

Step 1. A view for Countries European Countries

1. Create a new view called ‘europe_view’ with the Code, Name and Population columns from the Country table for only the countries in Europe:

```
mysql> CREATE VIEW europe_view AS  
->   SELECT Code, Name, Population  
->     FROM Country WHERE Continent='europe';
```

Returns an OK for the view creation.

Step 2. Verify

1. Run the **SHOW TABLES** command to see if the new view has been added:

```
SHOW TABLES;
```

The **SHOW TABLES** returns a list of the current tables in the `world` database, including the new `europe_view` view.

Step 3. Retrieve from View

1. Select all the contents of the new `europe_view`:

```
mysql> SELECT * FROM europe_view;
```

The **SELECT** will show a list of all the countries in Europe, including the code, name and population for each:

Code	Name	Population
ALB	Albania	3741320
AND	Andorra	85800
AUT	Austria	8900980
BEL	Belgium	11262900



464

15.4 Updatable Views

A view is updatable if it can be used with statements such as **UPDATE** or **DELETE** to modify the underlying base table. Not all views are updatable. For example, you might be able to update a table, but you cannot update a view on the table if the view is defined in terms of aggregate values calculated from the table. The reason for this is that each view row need not correspond to a unique base table row, in which case MySQL would not be able to determine which table row to update.

The primary conditions for updatability are that there must be a one-to-one relationship between the rows in the view and the rows in the base table, and that the view columns to be updated must be defined as simple column references, not expressions. There are a few other conditions as well, but we will not go into them here.

The following example demonstrates updatability. Create a simple view that contains the rows in the `Country` table for countries in Europe:

```
mysql> CREATE VIEW EuropePop AS
->   SELECT Name, Population FROM Country
-> WHERE Continent = 'Europe';
Query OK, 0 rows affected (#.## sec)
```

465

The `EuropePop` view satisfies the one-to-one requirement, and its columns are simple column references, not expressions such as `col1+1` or `col2/col3`. `EuropePop` is updatable, as demonstrated by the following statements. The example also selects from the base table `Country` to show that the base table is indeed modified by the **UPDATE** and **DELETE** statements that use the view.

```
mysql> SELECT * FROM EuropePop WHERE Name = 'San Marino';
+-----+-----+
| Name      | Population |
+-----+-----+
| San Marino |      27000 |
+-----+-----+
1 row in set (#.## sec)
mysql> UPDATE EuropePop SET Population = Population + 1
-> WHERE Name = 'San Marino';
Query OK, 1 row affected (#.## sec)
Rows matched: 1  Changed: 1  Warnings: 0
mysql> SELECT * FROM EuropePop WHERE Name = 'San Marino';
+-----+-----+
| Name      | Population |
+-----+-----+
| San Marino |      27001 |
+-----+-----+
1 row in set (#.## sec)
```



466

```
mysql> SELECT * FROM Country WHERE Name = 'San Marino';
+-----+-----+-----+
| Name      | Population | Continent |
+-----+-----+-----+
| San Marino |    27001 | Europe     |
+-----+-----+-----+
1 row in set (#.## sec)
mysql> DELETE FROM EuropePop WHERE Name = 'San Marino';
Query OK, 1 row affected (#.## sec)

mysql> SELECT * FROM EuropePop WHERE Name = 'San Marino';
Empty set (#.## sec)

mysql> SELECT * FROM Country WHERE Name = 'San Marino';
Empty set (#.## sec)
```





InLine Lab 15-B

In this exercise you will use the methods covered in this chapter for using updatable views. This will require a MySQL command line client and the world database.

Step 1. Update on Europe view

1. Change the name of the country with code ‘SWE’ to ‘MySQL’ in the ‘europe_view’ view:

```
mysql> UPDATE europe_view SET Name='MySQL' WHERE code='SWE';
```

Returns OK.

Step 2. Verify

1. Select the name of the country with the code of ‘SWE’:

```
mysql> SELECT Name FROM Country WHERE Code='SWE';
```

Returns the new name MySQL;

```
+-----+
| name   |
+-----+
| MySQL  |
+-----+
```

Step 3. DELETE from Europe View

1. Delete the row from europe_view that has the country code of ‘BEL’:

```
mysql> DELETE FROM europe_view WHERE Code = 'BEL';
```

Returns OK.

Step 4. Verify

1. Select the name of the country with the code of ‘BEL’:

```
mysql> SELECT Name FROM europe_view WHERE Code = 'BEL';
```

Returns an empty set, now that the row for BEL (Belgium) has been deleted.



Step 5. Insert a new row

1. Insert a new row into the europe_view view and confirm the insertion:

```
mysql> INSERT INTO europe_view VALUES ('XXX', 'MySQL', 450);
mysql> SELECT Name FROM europe_view WHERE Code = 'BEL';
```

The row is inserted:

name
MySQL



467

15.4.1 Insertable Views

An updatable view is insertable if it also satisfies these additional requirements for the view columns:

- There must be no duplicate view column names.
- The view must contain all columns in the base table that do not have a default value.
- The view columns must be simple column references and not derived columns. A derived column is one that is not a simple column reference but is derived from an expression. These are examples of derived columns:
 - 3.14159
 - col1 + 3
 - UPPER(col2)
 - col3 / col4
 - (<subquery>)

A view that has a mix of simple column references and derived columns is not insertable, but it can be updatable if you update only those columns that are not derived. Consider this view:

```
CREATE VIEW v AS SELECT col1, 1 AS col2 FROM t;
```

This view is not insertable because col2 is derived from an expression. But it is updatable if the update does not try to update col2. This update is allowable:

```
UPDATE v SET col1 = 0;
```

This update is not allowable because it attempts to update a derived column:

```
UPDATE v SET col2 = 0;
```

468

It is possible for a multiple-table view to be updatable with the following restrictions:

- It can be processed with the **MERGE** algorithm
- The view must use an inner join (not an outer join or a **UNION**)
- Only a single table in the view definition can be updated
- Views that use **UNION ALL** are disallowed even though they might be theoretically updatable
- **INSERT** can only work if it inserts into a single table. **DELETE** is not supported

If a table contains an **AUTO_INCREMENT** column, inserting into an insertable view on the table that does not include the **AUTO_INCREMENT** column does not change the value of **LAST_INSERT_ID()**, because the side effects of inserting default values into columns not part of the view should not be visible.

NOTE: For a view to be insertable, it must be updatable.



469

15.4.2 WITH CHECK OPTION

If a view is updatable, you can use the **WITH CHECK OPTION** clause to place a constraint on allowable modifications. This clause causes the conditions in the WHERE clause of the view definition to be checked when updates are attempted:

- An **UPDATE** to an existing row is allowed only if the **WHERE** clause remains true for the resulting row.
- An **INSERT** is allowed only if the **WHERE** clause is true for the new row.

In other words, **WITH CHECK OPTION** ensures that you cannot update a row in such a way that the view no longer selects it, and that you cannot insert a row that the view will not select.

Using the `Country` table created earlier in this section, define the following view that selects countries with a population of at least 100 million:

```
mysql> CREATE VIEW LargePop AS
->   SELECT Name, Population FROM Country
->   WHERE Population >= 100000000
->   WITH CHECK OPTION;
Query OK, 0 rows affected (#.## sec)
mysql> SELECT * FROM LargePop;
+-----+-----+
| Name           | Population |
+-----+-----+
| Bangladesh     | 129155000  |
| Brazil          | 170115000  |
| Indonesia       | 212107000  |
| India           | 1013662000 |
| Japan           | 126714000  |
| China           | 1277558000 |
| Nigeria          | 111506000  |
| Pakistan         | 156483000  |
| Russian Federation | 146934000 |
| United States    | 278357000  |
+-----+-----+
```



The **WITH CHECK OPTION** clause in the view definition allows some modifications but disallows others. For example, it's possible to increase the population of any country in the view:

470

```
mysql> UPDATE LargePop SET Population = Population + 1
      -> WHERE Name = 'Nigeria';
Query OK, 1 row affected (#.## sec)
Rows matched: 1  Changed: 1  Warnings: 0
mysql> SELECT * FROM LargePop WHERE Name = 'Nigeria';
+-----+-----+
| Name | Population |
+-----+-----+
| Nigeria | 111506001 |
+-----+-----+
1 row in set (#.## sec)
```

It is also possible to decrease a population value, but only if it does not drop below the minimum value of 100 million that is required by the views' **WHERE** clause:

```
mysql> UPDATE LargePop SET Population = 99999999
      -> WHERE Name = 'Nigeria';
ERROR 1369 (HY000): CHECK OPTION failed 'world.LargePop'
```



471

15.5 Managing Views

15.5.1 Checking Views

When you define a view, any object referenced by the view (such as a table, view, or column) must exist. However, a view can become invalid if a table, view, or column on which it depends is dropped or altered. To check a view for problems of this nature, use the **CHECK TABLE** statement. The following example shows the output from **CHECK TABLE** after renaming a table that a view depends on:

```
mysql> CREATE TABLE t1 (i INT);
Query OK, 0 rows affected (#.# sec)

mysql> CREATE VIEW v AS SELECT i FROM t1;
Query OK, 0 rows affected (#.# sec)
mysql> RENAME TABLE t1 TO t2;
Query OK, 0 rows affected (#.# sec)

mysql> CHECK TABLE v\G
***** 1. row *****
Table: world.v
Op: check
Msg_type: error
Msg_text: View 'world.v' references invalid table(s) or
          column(s) or function(s)
1 row in set (#.# sec)
```

472

15.5.2 Altering Views

To change the definition of an existing view, use the **ALTER VIEW** statement. **ALTER VIEW** discards the current definition for the view and replaces it with the new definition in the statement. It is an error if the named view does not exist. Syntactically, the only differences from **CREATE VIEW** are that the initial keyword is **ALTER** rather than **CREATE**, and the **OR REPLACE** option cannot be used.

The following statement redefines the LargePop view created in a previous section so that it no longer includes a **WITH CHECK OPTION** clause:

```
ALTER VIEW LargePop AS
SELECT Name, Population FROM Country
WHERE Population >= 100000000;
```

CREATE VIEW can be used instead of **ALTER VIEW**.



473

15.5.3 Dropping Views

To delete one or more views, use the **DROP VIEW** statement:

```
DROP VIEW [IF EXISTS] <view_name> [, <view_name> ... ]
```

It is an error if a given view does not exist. Include the **IF EXISTS** clause to generate a warning instead. (The warning can be displayed with **SHOW WARNINGS**.) **IF EXISTS** is a MySQL extension to standard SQL. If the view v1 exists but the view v2 does not, the following **DROP VIEW** statement results in a warning for the attempt to drop v2:

```
mysql> DROP VIEW IF EXISTS v1, v2;
Query OK, 0 rows affected, 1 warning (#.## sec)
mysql> SHOW WARNINGS;
+-----+-----+
| Level | Code | Message           |
+-----+-----+
| Note  | 1051 | Unknown table 'world.v2' |
+-----+-----+
1 row in set (#.## sec)
```





InLine Lab 15-C

In this exercise you will use the methods covered in this chapter for using checking views. This will require a MySQL command line client and the **world** database.

Step 1. Check View

1. Check the europe_view view:

```
CHECK TABLE europe_view\G
```

Returns the table name europe_view and status;

```
***** 1. row *****
Table: world.europe_view
Op: check
Msg_type: status
Msg_text: OK
1 row in set (#.# sec)
```

Step 2. ALTER VIEW

1. Add the **WITH CHECK OPTION** to the europe_view table:

```
ALTER VIEW europe_view AS SELECT Code, Name, Population
FROM Country WHERE Continent='europe'
WITH CHECK OPTION;
```

Returns OK.

Step 3. Insert

1. Insert a new country into the view :

```
INSERT INTO europe_view VALUES ('XYZ', 'MySQL', 450);
```

The insert fails as a result of the check option;

```
ERROR 1369 (HY000) : CHECK OPTION failed 'world.europe_view'
```



15.6 Obtaining View Metadata

474

15.6.1 Using the information_schema

The `information_schema` database contains a `TABLES` table that contains a row for every table accessible to the user. The `TABLE_TYPE` column of the `TABLES` table indicates its kind and contains the value `VIEW` in case the table is a view. However, most columns in the `TABLES` table pertain to base tables. It contains almost no columns that reveal any information about the views themselves.

The `information_schema` database also contains a `VIEWS` table that contains metadata that is specific to views. For example, to display information about the `world.CityView` view that was created earlier, use this statement:

```
mysql> SELECT * FROM information_schema.VIEWS
      -> WHERE TABLE_NAME = 'CityView'
      -> AND TABLE_SCHEMA = 'world'\G
***** 1. row *****
TABLE_CATALOG: NULL
TABLE_SCHEMA: world
TABLE_NAME: CityView
VIEW_DEFINITION: select `world`.`City`.`ID` AS `ID`, `world`.`City`.`Name` AS
`Name` from `world`.`City`
CHECK_OPTION: NONE
IS_UPDATABLE: YES
```

475

15.6.2 SHOW Statements

MySQL also supports a family of `SHOW` statements that retrieve metadata. To display the definition of a view, use the `SHOW CREATE VIEW` statement:

```
mysql> SHOW CREATE VIEW CityView\G
***** 1. row *****
      View: CityView
Create View: CREATE ALGORITHM=UNDEFINED VIEW `world`.`CityView` AS
select `world`.`City`.`ID` AS `ID`,
`world`.`City`.`Name` AS `Name` from `world`.`City`
```

476

Some statements in MySQL that originally were designed to obtain metadata pertaining to base tables have been extended so that they also work with views:

- `DESCRIBE` and `SHOW COLUMNS`
- `SHOW TABLE STATUS`
- `SHOW TABLES`



By default, **SHOW TABLES** lists only the names of tables and views. MySQL 5 has a **SHOW FULL TABLES** variant that displays a second column. The values in that column are the same as those in the `TABLE_TYPE` column of the `information_schema.TABLES` table: **BASE TABLE** or **VIEW**, indicating what kind of object each name refers to:

```
mysql> SHOW FULL TABLES FROM world;
+-----+-----+
| Tables_in_world | Table_type |
+-----+-----+
| City           | BASE TABLE |
| CityView        | VIEW        |
| Country         | BASE TABLE |
| CountryLangCount | VIEW       |
| CountryLanguage | BASE TABLE |
| EuropePop      | VIEW        |
| LargePop        | VIEW        |
+-----+-----+
```





InLine Lab 15-D

In this exercise you will use the methods covered in this chapter for metadata for views. This will require a MySQL command line client and the world database.

Step 1. Query the VIEWS table

1. Select the europe_view view from the information_schema database:

```
SELECT *
FROM information_schema.VIEWS
WHERE TABLE_NAME = 'europe_view'
AND TABLE_SCHEMA = 'world'\G
```

Returns the table information for europe_view.

Step 2. SHOW CREATE TABLE

1. Show the command used to create the original view of europe_view

```
SHOW CREATE VIEW europe_view\G
```

Result shows the **CREATE VIEW** syntax used to create the current europe_view.

Step 3. DESCRIBE

1. Show the table structure of the europe_view

```
DESCRIBE europe_view;
```

A table showing all the columns and their attributes for europe_view is output.

Step 4. SHOW FULL TABLES

1. List all the tables in the world database and their respective table types;

```
SHOW FULL TABLES FROM world;
```

Returns a list of all world tables and their types.





Further Practice

In this exercise, you will create views.

477

1. Create a view `CountryCapitals` that is a join of the tables `Country` and `City` and contains the columns `Code`, `Country.Name`, `Continent`, `City.Name`, `City.Id`. Is the view updatable? Is it insertable?
2. Create a view `Languages` which contains a row for each Language with the amount of people speaking the language and a list of countries where the language is spoken.
3. Replace the existing view called `europe_view` (created in a prior InLine Lab) and populate it with data from the `Country` database for the `Code`, `Name` and `Continent` when the continent is ‘Europe’, making sure that any updates will be checked before performed.
4. Change the name of the continent with code ‘DEU’ to ‘Asia’ in the new `europe_view` view. Explain the result.
5. You want to make sure that upon **INSERT/UPDATE** each `City` always has a corresponding country in the `Country` table. How can you achieve this?

Hint: The above view will require a subquery to accurately create.

6. Create a view ‘FrenchCont’ that contains the continents that have countries in which more than 20% of the population speak French. Confirm that the new view exists and list all of its contents.



15.7 Chapter Summary

478

This chapter introduced Views in MySQL. In this chapter, you learned to:

- Define views
- List the reasons for using views
- Create a view
- Check a view
- Alter and remove a view
- Explain required privileges for views





16 PREPARED STATEMENTS

16.1 Learning Objectives

480

This chapter introduces Prepared Statements in MySQL. In this chapter, you will learn to:

- List the reasons for using prepared statements
- Using prepared statements with mysql
- Preparing, executing, and de-allocating prepared statements



16.2 Why Use Prepared Statements?

481

MySQL Server supports prepared statements, which are useful when you want to run several queries that differ only in very small details. For example, you can prepare a statement, and then execute it multiple times, each time with different data values.

Besides offering a convenience, prepared statements also provide enhanced performance because the complete statement is parsed only once by the server. When the parse is complete, the server and client may make use of a new protocol that requires fewer data conversions. This usually makes for less traffic between the server and client than when sending each statement individually.

16.3 Using Prepared Statements from the mysql Client

482

In most circumstances, statements are prepared and executed using the programming interface that you normally use for writing applications that use MySQL. However, to aid in testing and debugging, it is possible to define and use prepared statements from within the mysql command-line client. Prepared statements are session-bound, therefore they are not visible in other sessions.

16.3.1 User Defined Variables:

You can store a value in a user-defined variable (also known as *user variable*) and then refer to it later when executing various statement types, including prepared statements. This enables you to pass values from one statement to another. User-defined variables are connection-specific. That is, a user variable defined by one client cannot be seen or used by other clients. All variables for a given client connection are automatically freed when that client exits.

User variables are written as `@var_name`, where the variable name `var_name` may consist of alphanumeric characters from the current character set, “.”, “_”, and “\$”. A user variable name can contain other characters if you quote it as a string or identifier (for example, `@'my-var'`, `@"my-var"`, or `@`my-var``).

User variable names are case sensitive before MySQL 5.0 and not case sensitive in MySQL 5.0 and up.

One way to set a user-defined variable is by issuing a **SET** statement:

```
SET @var_name = expr [, @var_name = expr] ...
```

For **SET**, either `=` or `:=` can be used as the assignment operator. The `expr` assigned to each variable can evaluate to an integer, real, string, or **NULL** value. However, if the value of the variable is selected in a result set, it is returned to the client as a string.

If a user variable is assigned a string value, it has the same character set and collation as the string. The coercibility of user variables is implicit. (This is the same coercibility as for table column values, including data types.)



483

Here is a short example that illustrates the use of a prepared statement. It prepares a statement that determines how many languages are spoken in a given country, executes it (using user defined variables) multiple times, and disposes of it:

```
mysql> PREPARE my_stmt FROM
      -> 'SELECT COUNT(*) FROM CountryLanguage WHERE CountryCode= ?';
Query OK, 0 rows affected (#.## sec)
Statement prepared
mysql> SET @code = 'ESP';

mysql> EXECUTE my_stmt USING @code;
Query OK, 0 rows affected (#.## sec)
+-----+
| COUNT(*) |
+-----+
|        4 |
+-----+
1 row in set (#.## sec)

mysql> SET @code = 'RUS';

mysql> EXECUTE my_stmt USING @code;
Query OK, 0 rows affected (#.## sec)
+-----+
| COUNT(*) |
+-----+
|       12 |
+-----+
1 row in set (#.## sec)

mysql> DEALLOCATE PREPARE my_stmt;
Query OK, 0 rows affected (#.## sec)
```



16.4 Preparing a Statement

484

The **PREPARE** statement is used to define an SQL statement that will be executed later. **PREPARE** takes two arguments: a name to assign to the statement once it has been prepared, and the text of an SQL statement. Prepared statement names are not case sensitive. The text of the statement can be given either as a literal string or as a user variable containing the statement.

The statement may not be complete, because data values that are unknown at preparation time are represented by question mark ('?') characters that serve as parameter markers. At the time the statement is executed, you provide specific data values, one for each parameter in the statement. The server replaces the markers with the data values to complete the statement. Different values can be used each time the statement is executed.

485

The following example prepares a statement named namepop. When executed later with a country code as a parameter value, the statement will return a result set containing the corresponding country name and population from the world database.

```
mysql> PREPARE namepop FROM '
    '> SELECT Name, Population
    '> FROM Country
    '> WHERE Code = ?
    '> ';
Query OK, 0 rows affected (#.# sec)
Statement prepared
```

The message `Statement prepared` indicates that the server is ready to execute the namepop statement. On the other hand, if the server finds a problem as it parses the statement during a **PREPARE**, it returns an error and does not prepare the statement:

```
mysql> PREPARE error FROM
    '-> 'SELECT NonExistingColumn FROM Country WHERE Code = ?';
ERROR 1054 (42S22): Unknown column 'NonExistingColumn' in 'field list'
```

If you **PREPARE** a statement using a statement name that already exists, the server first discards the prepared statement currently associated with the name, and then prepares the new statement. If the new statement contains an error and cannot be prepared, the result is that no statement with the given name will exist.

486

MySQL does not allow every type of SQL statement to be prepared. Those that may be prepared are limited to the following:

- **SELECT** statements
- Statements that modify data: **INSERT**, **REPLACE**, **UPDATE**, and **DELETE**
- **CREATE TABLE** statements
- **SET**, **DO**, and many **SHOW** statements



...and as of mysql version 5.1...

- **ANALYZE TABLE**
- **OPTIMIZE TABLE**
- **REPAIR TABLE**
- **ANALYZE TABLE**
- **CACHE INDEX**
- **CHANGE MASTER**
- **CHECKSUM**
- **CREATE | RENAME | DROP DATABASE**
- **CREATE | RENAME | DROP USER**
- **FLUSH ...**
- **GRANT**
- **REVOKE**
- **KILL**
- **LOAD INDEX INTO CACHE**
- **RESET ...**
- **SHOW BINLOG EVENTS**
- **SHOW CREATE ...**
- **SHOW AUTHORS | CONTRIBUTORS | WARNINGS | ERRORS**
- **SHOW LOGS**
- **SHOW SLAVE ...**
- **INSTALL PLUGIN**
- **UNINSTALL PLUGIN**

A prepared statement exists only for the duration of the session in which it is created, and is visible only to the session in which it is created. When a session ends, all prepared statements for that session are discarded.



487

16.5 Executing a Prepared Statement

After a statement has been prepared, it can be executed. If the statement contains any ‘?’ parameter markers, a data value must be supplied for each of them by means of user variables.

To execute a prepared statement, initialize any user variables needed to provide parameter values, and then issue an **EXECUTE . . . USING** statement. The following example prepares a statement and then executes it several times using different data values:

```
mysql> PREPARE namepop FROM '
    '> SELECT Name, Population
    '> FROM Country
    '> WHERE Code = ?
    '> ';
Query OK, 0 rows affected (#.## sec)
Statement prepared

mysql> SET @var1 = 'USA';
Query OK, 0 rows affected (#.## sec)

mysql> EXECUTE namepop USING @var1;
+-----+-----+
| Name      | Population |
+-----+-----+
| United States | 278357000 |
+-----+-----+
1 row in set (#.## sec)

mysql> SET @var2 = 'GBR';
Query OK, 0 rows affected (#.## sec)

mysql> EXECUTE namepop USING @var2;
+-----+-----+
| Name      | Population |
+-----+-----+
| United Kingdom | 59623400 |
+-----+-----+
1 row in set (#.## sec)

mysql> SELECT @var3 := 'CAN';
+-----+
| @var3 := 'CAN' |
+-----+
| CAN          |
+-----+
1 row in set (#.## sec)
```

488



```
mysql> EXECUTE namepop USING @var3;
+-----+-----+
| Name | Population |
+-----+-----+
| Canada | 31147000 |
+-----+-----+
1 row in set (#.## sec)
```

If you refer to a user variable that has not been initialized, its value is **NULL**:

```
mysql> EXECUTE namepop USING @var4;
Empty set (#.## sec)
```

NOTE: The **USING** clause is not required if there are no wildcards in the statement.

16.6 Deallocating a Prepared Statement

489

Prepared statements are dropped automatically when they are redefined or when you close the connection to the server, so there is rarely any reason to drop them explicitly. However, should you wish to do so (for example, to free memory on the server side), use the **DEALLOCATE PREPARE** statement:

```
mysql> DEALLOCATE PREPARE namepop;
Query OK, 0 rows affected (#.## sec)
```

MySQL also provides **DROP PREPARE** as an alias for **DEALLOCATE PREPARE**.

In addition, if a prepared statement is allocated again, the original prepared statement is “dropped” to allow the new prepared statement to take its place.





Further Practice

In this exercise, you will use prepared statements using the mysql client, using the world database.

490

1. Create a prepared statement called ‘mystmt’ which includes a query for the name from the city table, where the id will be set as equal to a variable.
2. Set the variable equal to 3567.
3. Execute the prepared statement with the set variable.
4. Reset the variable to 3568.
5. Re-execute the prepared statement with the newly set variable.
6. Explicitly remove the newly created prepared statement.



491

16.7 Chapter Summary

This chapter introduced Prepared Statements in MySQL. In this chapter, you learned to:

- List the reasons for using prepared statements
- Using prepared statements with mysql
- Preparing, executing, and de-allocating prepared statements



17 EXPORTING AND IMPORTING DATA

493

17.1 Learning Objectives

This chapter introduces Exporting and Importing in MySQL. In this chapter, you will learn to:

- Import data using SQL
- Export data using SQL
- Export using the ‘mysqldump’ database backup client
- Import using the ‘mysqlimport’ client
- Import data with the SOURCE command



17.2 Export and Import Data Using SQL

494

17.2.1 Export Data Using SELECT with INTO OUTFILE

A `SELECT` statement can be used with the `INTO OUTFILE` clause to write the results set directly into a file. To use `SELECT` in this way, place the `INTO OUTFILE` clause before the `FROM` clause. For example, to write the contents of the `City` table into a file named `City.txt`, issue this statement:

```
mysql> SELECT * INTO OUTFILE 'C:/City.txt' FROM City;
```

The name of the file indicates the location where you want to write it. MySQL interprets the pathname for files located on the server host. For example, given the statement above, the server writes the file into the `C:\` directory. The text file contains all the row data from the `City` table, in the default format:

5	Amsterdam	NLD	Noord-Holland	731200
6	Rotterdam	NLD	Zuid-Holland	593321
7	Haag	NLD	Zuid-Holland	440900
8	Utrecht	NLD	Utrecht	234323
9	Eindhoven	NLD	Noord-Brabant	201843
10	Tilburg	NLD	Noord-Brabant	193238
11	Groningen	NLD	Groningen	172701
12	Breda	NLD	Noord-Brabant	160398
13	Apeldoorn	NLD	Gelderland	153491

495

The filename is given as a quoted string. On Windows, the pathname separator character is ‘\’, but MySQL treats the backslash as the escape character in strings. To deal with this issue, write separators in Windows pathnames either as ‘/’ or as ‘\\’. You can specify the filename as shown above, or as follows:

```
mysql> SELECT * INTO OUTFILE 'C:\\\\City.txt' FROM City;
```

If no file path is specified, MySQL assumes that the file should be placed in the database data directory. For example, the following statement result file would be placed in the `C:\Program Files\MySQL\MySQL Server 5.1\data\world` directory:

```
mysql> SELECT * INTO OUTFILE 'City.txt' FROM City;
```



Use of the `INTO OUTFILE` changes the operation of the `SELECT` statement in several ways:

- The file will be written to the server host, instead of sending the file over the network to the client (file name must not already exist)
- Causes the server to write a new file on the server host (must connect to the server using an account that has FILE privilege)
- File is created with filesystem access permissions that make it owned by the MySQL server but world-readable
- File contains one line per row select by the statement (by default, column values are delimited by tab characters and lines are terminated with newlines)

496

Using Data File Format Specifiers:

`SELECT...INTO OUTFILE` assumes a default data file format in which column values are separated by tab characters and records are terminated by newlines. If you want `SELECT...INTO OUTFILE` to write a file with different separators or terminators, you'll need to indicate the format to use. It is also possible to control data value quoting and escaping behavior.

The format specifiers supported by `SELECT...INTO OUTFILE` enable you to define the general characteristics that apply to all column values: What characters separate column values in data rows, whether values are quoted, and whether there is an escape character that signifies special character sequences.

For `SELECT...INTO OUTFILE`, they follow the output filename. The syntax for format specifiers is as follows:

```
FIELDS
  TERMINATED BY '<string>'
  ENCLOSED BY '<char>'
  ESCAPED BY '<char>'
LINES TERMINATED BY '<string>'
```

The `FIELDS` clause defines the formatting of data values within a line. The `LINES` clause defines the line-ending sequence. In other words, `FIELDS` indicates the structure of column values within records and `LINES` indicates where record boundaries occur.

The `TERMINATED BY`, `ENCLOSED BY`, and `ESCAPED BY` parts of the `FIELDS` clause may be given in any order. You need not specify all three parts. Defaults are used for any that are missing (or if the `FIELDS` clause itself is missing):

- Data values are assumed to be terminated by (that is, separated by) tab characters. To indicate a different value, include a `TERMINATED BY` option.
- Data values are assumed to be unquoted. To indicate a quote character, include an `ENCLOSED BY` option. For `LOAD DATA INFILE`, enclosing quotes are stripped from input values if they're found. For `SELECT...INTO OUTFILE`, output values are written enclosed within quote characters.

A variation on `ENCLOSED BY` is `OPTIONALLY ENCLOSED BY`. This is the same as `ENCLOSED` for `LOAD DATA INFILE`, but different for `SELECT...INTO OUTFILE`: The presence of `OPTIONALLY` causes output value quoting only for string columns, not for all columns.



497

- The default escape character is backslash ('\'). Any occurrence of this character within a data value modifies interpretation of the character that follows it. To indicate a different escape character, include an **ESCAPED BY** option. MySQL understands the following special escape sequences:

- \N - **NULL** value
- \0 - NUL (zero) byte
- \b - Backspace
- \n - Newline (linefeed)
- \r - Carriage return
- \s - Space
- \t - Tab
- \' - Single quote
- \" - Double quote
- \\ - Backslash

All these sequences except \N are understood whether they appear alone or within a longer data value. \N is understood as **NULL** only when it appears alone.

The default line terminator is the newline (linefeed) character. To indicate a line-ending sequence explicitly, use a **LINES** clause. Common line terminators are newline, carriage return, and carriage return/newline pairs. Specify them as follows:

```
LINES TERMINATED BY '\n'  
LINES TERMINATED BY '\r'  
LINES TERMINATED BY '\r\n'
```

Because newline is the default line terminator, it need be specified only if you want to make the line-ending sequence explicit. Newline terminators are common on Unix/Linux systems and carriage return/newline pairs are common on Windows.

The **ESCAPED BY** option controls only the handling of values in the data file, not how you write the statement itself. If you want to specify a data file escape character of '@', you would write **ESCAPED BY '@'**. That doesn't mean you then use '@' to escape special characters elsewhere in the statement. For example, you'd still specify carriage return as the line termination character using **LINES TERMINATED BY '\r'**, not using **LINES TERMINATED BY '@r'**.



Suppose that you want to export the `City` table using the comma-separated values (CSV) format, with values quoted by double quote characters and lines terminated by carriage returns, use this `SELECT...INTO OUTFILE` statement:

```
mysql> SELECT * INTO OUTFILE 'C:/City.csv'  
-> FIELDS TERMINATED BY ','  
-> ENCLOSED BY '\"'  
-> LINES TERMINATED BY '\r'  
-> FROM City;
```

The text file contains all the rows from the `City` table, in the CSV format specified:

```
"5","Amsterdam","NLD","Noord-Holland","731200"  
"6","Rotterdam","NLD","Zuid-Holland","593321"  
"7","Haag","NLD","Zuid-Holland","440900"  
"8","Utrecht","NLD","Utrecht","234323"  
"9","Eindhoven","NLD","Noord-Brabant","201843"  
"10","Tilburg","NLD","Noord-Brabant","193238"  
"11","Groningen","NLD","Groningen","172701"  
"12","Breda","NLD","Noord-Brabant","160398"  
"13","Apeldoorn","NLD","Gelderland","153491"
```





InLine Lab 17-A

In this exercise you will use the **SELECT...INTO OUTFILE** statement. This will require a MySQL command line client and access to the mysql server.

Step 1. SELECT...INTO OUTFILE

- Issue a **SELECT...INTO OUTFILE** statement which will create a text output file of the CountryLanguage table. Put this file in the C:\ directory:

```
mysql> SELECT * INTO OUTFILE 'C:/CountryLanguage.txt' FROM CountryLanguage;
```

Returns an OK. A text file is created in the C:\ directory.

Step 2. Review

- Review the text file. Confirm that all the table rows and columns are represented:

AFG	Pashto	T	52.4
NLD	Dutch	T	95.6
ANT	Papiamento	T	86.2
ALB	Albaniana	T	97.9
DZA	Arabic	T	86.0
ASM	Samoan	T	90.6
AND	Spanish	F	44.6
....			

Step 3. Using FIELDS TERMINATED BY

- Issue a **SELECT...INTO OUTFILE** statement which will create a text output file of the CountryLanguage table in CSV format, with fields ending with a comma, and strings enclosed by double quotes, and lines ending with a return (\r). Put this file in the C:\ directory:

```
mysql> SELECT * INTO OUTFILE 'C:/CountryLanguage_csv.txt'
      -> FIELDS TERMINATED BY ',' ENCLOSED BY '"'
      -> LINES TERMINATED BY '\r'
      -> FROM CountryLanguage;
```

Returns an OK. A text file is created in the current directory (where all course files reside).

Step 4. Review

- Review the text file and note the format. Confirm that all the table rows and columns are represented:

"AFG", "Pashto", "T", "52.4"
"NLD", "Dutch", "T", "95.6"
"AND", "Spanish", "F", "44.6"
...



498

17.2.2 Importing Data Using LOAD DATA INFILE

LOAD DATA INFILE provides an alternative to **INSERT** for adding new records to a table. **INSERT** specifies data values directly in the text of the statement. **LOAD DATA INFILE** reads the values from a separate data file.

For example, assume that the `City` table (used previously to export the ‘`City.txt`’ file) has been emptied of its’ contents and we want to re-populate the table with the exported file. You would issue the following statement:

```
mysql> LOAD DATA INFILE 'C:/City.txt' INTO TABLE City;
```

LOAD DATA INFILE has similar clauses and format specifiers as **SELECT...INTO OUTFILE**.

As shown earlier with **SELECT...INTO OUTFILE**, if no file path is specified the file will be placed in the database data directory. To retrieve the data from the `City.txt` file previously written to the database (Windows-specific path separators shown here):

```
mysql> LOAD DATA INFILE
      -> 'C:\\Program Files\\MySQL\\MySQL Server 5.1\\data\\world\\City.txt'
      -> INTO TABLE City;
```

Importing Tab Delimited or Comma Separated Files:

499

To import a data file, containing tab delimited or comma separated table data, you can use the **LOAD DATA INFILE** command. The characteristics of the file you must know:

- The column value separators
- The line separator
- What are the values enclosed within? (quotes, for example)
- Are the column names specified in the file?
- Is there a header indicating rows of the table to skip before importing?
- The file system where the file resides
- Are privileges required to access the file?
- The order of the columns
- Do the column numbers in file and table match?

Suppose that you want to import the ‘`City.csv`’ table (CSV format created in a previous example), use this **LOAD DATA INFILE** statement:

```
mysql> LOAD DATA INFILE 'C:/City.csv' INTO TABLE City
      -> FIELDS TERMINATED BY ','
      -> ENCLOSED BY '"'
      -> LINES TERMINATED BY '\r';
```



500

Specifying Data File Location:

LOAD DATA INFILE can read data files that are located on the server host or on the client host:

- By default, MySQL assumes that the file is located on the server host. The MySQL server reads the file directly.
- If the statement begins with **LOAD DATA LOCAL INFILE** rather than with **LOAD DATA INFILE**, the file is read from the client host on which the statement is issued. In other words, **LOCAL** means local to the client host from which the statement is issued. In this case, the client program reads the data file and sends its contents over the network to the server. For example:

```
mysql> LOAD DATA LOCAL INFILE 'C:/City.txt' INTO TABLE City;
```

Skipping Data File Lines:

To ignore the initial part of the data file, use the **IGNORE n LINES** clause, where n is an integer that indicates the number of input lines to skip. This clause commonly is used when a file begins with a row of column names rather than data values. For example, to skip the first input line, a statement might be written like this:

```
mysql> LOAD DATA INFILE 'C:/City.txt' INTO TABLE City
      -> IGNORE 2 LINES;
Query OK, 2231 rows affected (#.## sec)
Records: 2231 Deleted: 0 Skipped: 0 Warnings: 0
```

NOTE: The results show that the number of records has gone down to 2231 (originally 2233).

501

LOAD DATA INFILE and Duplicate Records:

When you add new records to a table with an **INSERT** or **REPLACE** statement, you can control how to handle new records containing values that duplicate unique key values already present in the table. You can allow an error to occur, ignore the new records, or replace the old records with the new ones. **LOAD DATA INFILE** affords the same types of control over duplicate records by means of two modifier keywords. However, its duplicate-handling behavior differs slightly depending on whether the data file is on the server host or the client host, so you must take the data file location into account.

When loading a file that's located on the server host, **LOAD DATA INFILE** handles records that contain duplicate unique keys as follows:

- By default, an input record that causes a duplicate-key violation results in an error and the rest of the data file isn't loaded. Records processed up to that point are loaded into the table.
- If you specify the **IGNORE** keyword following the filename, new records that cause duplicate-key violations are ignored and no error occurs. **LOAD DATA INFILE** processes the entire file, loads all records not containing duplicate keys, and discards the rest.
- If you specify the **REPLACE** keyword after the filename, new records that cause duplicate-key violations replace any records already in the table that contain the duplicated key values. **LOAD DATA INFILE** processes the entire file and loads all its records into the table.

IGNORE and **REPLACE** are mutually exclusive. You can specify one or the other, but not both.



For data files located on the client host, duplicate unique key handling is similar, except that the default is to ignore records that contain duplicate keys rather than to terminate with an error. That is, the default is as though the `IGNORE` modifier is specified. The reason for this is that the client/server protocol doesn't allow transfer of the data file from the client host to the server to be interrupted after it has started, so there's no convenient way to abort the operation in the middle.

Information Provided by LOAD DATA INFILE

As `LOAD DATA INFILE` executes, it keeps track of the number of records processed and the number of data conversions that occur. Then it returns to the client an information string in the following format (the counts in each field will vary per `LOAD DATA INFILE` operation):

```
Records: 174 Deleted: 0 Skipped: 3 Warnings: 14
```

The fields have the following meaning:

- Records indicates the number of input records read from the data file. This is not necessarily the number of records added to the table.
- Deleted indicates the number of records in the table that were replaced by input records having the same unique key value as a key already present in the table. The value may be non-zero if you use the `REPLACE` keyword in the statement.
- Skipped indicates the number of data records that were ignored because they contained a unique key value that duplicated a key already present in the table. The value may be non-zero if you use the `IGNORE` keyword in the statement.
- Warnings indicates the number of problems found in the input file. These can occur for several reasons, such as missing data values or data conversion (for example, converting an empty string to 0 for a numeric column). The warning count can be larger than the number of input records because warnings can occur for each data value in a record. To see what caused the warnings, issue a `SHOW WARNINGS` statement after loading the data file.

NOTE: Even if the table is an InnoDB table and you are inside a transaction, a failed `LOAD DATA INFILE` cannot be rolled back.





InLine Lab 17-B

In this exercise you will use the **LOAD DATA INFILE** statement. This statement assumes that you have an empty table. Therefore, the beginning of this lab will consist of creating an empty table for use with the import. This will require access to the mysql server and the world database.

Step 1. Review Table definition

1. Show the original statement used to create the table CountryLanguage table:

```
mysql> SHOW CREATE TABLE CountryLanguage\G
```

Returns **CREATE TABLE** statement with full syntax.

Step 2. Create similar table

1. Use the same syntax as used for that table to create a new table named CountryLanguage2;

```
mysql> CREATE TABLE `CountryLanguage2` (
    ->   `CountryCode` CHAR(3)      NOT NULL DEFAULT '',
    ->   `Language` CHAR(30)       NOT NULL DEFAULT '',
    ->   `IsOfficial` ENUM('T', 'F') NOT NULL DEFAULT 'F',
    ->   `Percentage` FLOAT(4,1)    NOT NULL DEFAULT '0.0',
    ->   PRIMARY KEY (`CountryCode`, `Language`),
    -> ) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

Returns an OK and 0 rows affected.

Step 3. Verify creation

1. Show the tables list and confirm that the new table now exists:

```
mysql> SHOW TABLES;
```

Should now have the new table in the list;

+-----+
Tables_in_world
+-----+
city
country
countrylanguage
countrylanguage2
+-----+



Step 4. Load Data

1. Issue a **LOAD DATA INFILE** statement that loads the 'CountryLanguage.txt' into the table CountryLanguage2:

```
mysql> LOAD DATA INFILE 'C:/CountryLanguage.txt'  
-> INTO TABLE CountryLanguage2;
```

Returns an OK and 6 rows affected.

Step 5. Confirm

1. Select all the row/column data in the new CountryLanguage2 table to confirm that the table is now populated:

```
mysql> SELECT * FROM CountryLanguage2;
```

All the entries from the previous export file (and original table) should now be in the new CountryLanguage2 table.



502

17.3 Export and Import Using MySQL Client Programs

17.3.1 Export Data with ‘mysqldump’

The `mysqldump` utility dumps table contents to files. It can export tables in a variety of ways:

- Dump full structure, including `DROP` and `CREATE` statements to be able to re-create a complete setting
- Dump only data
- Dump only table structure
- Dump in a standard format
- Dump with MySQL specifics for optimized speed
- Can be compressed with a high ratio since dumps are clear text

There are three general ways to invoke `mysqldump` at the shell prompt:

```
shell> mysqldump [options] db_name [tables]
shell> mysqldump [options] --databases db_name1 [db_name2 db_name3...]
shell> mysqldump [options] --all-databases
```

If you do not name any tables following `db_name` or if you use the `--databases` or `--all-databases` option, entire databases are dumped.

It is best (and sometimes required) that you include the user and password in the `mysqldump` options.

503

For example, to dump the full `world` database you give it one argument;

```
shell> mysqldump -uroot -p<password> world
```

To dump only the `City` and `Country` tables in the `world` database, give the tables as arguments after the database;

```
mysqldump -uroot -p<password> world City Country
```

To dump multiple databases `world` and `test` simultaneously;

```
shell> mysqldump -uroot -p<password> --all-databases (or -A)
shell> mysqldump -uroot -p<password> --databases world test
```

To get a list of the options your version of `mysqldump` supports, execute `mysqldump --help`.



504

To dump a database to a text file using `mysqldump` use the redirect operator to indicate the filename and location. For example, to export all the tables and data for the world database:

```
shell> mysqldump -uroot -p<password> world > C:/world_dump.sql
```

As shown in the above example, it is best to include the filepath in the statement. Otherwise the file is placed in the current command line directory. This statement will create a text file containing all the commands necessary to recreate all the tables and data found in world:

```
-- MySQL      dump 10.12
--
-- Host: localhost      Database: world
...
--
-- Table structure for table `city`
--

DROP TABLE IF EXISTS `city`;
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=MyISAM AUTO_INCREMENT=4080 DEFAULT CHARSET=latin1;

--
-- Dumping data for table `city`
--
...

...
```

However, what if you want to export only the `CountryLanguage` table? To do this the command is modified as follows:

```
shell> mysqldump -uroot -p<password> world CountryLanguage >
C:/CountryLanguage.sql
```

NOTE: Existing files (with the same name) will be overwritten.



505

17.3.2 Import Data with mysqlimport

The `mysqlimport` client program loads data files into tables. It provides a command-line interface to the `LOAD DATA INFILE` statement. That is, `mysqlimport` examines the options given on the command line. It then connects to the server and, for each input file named in the command, issues a `LOAD DATA INFILE` statement that loads the file into the appropriate table.

Because `mysqlimport` works this way, to use it most effectively, you should be familiar with the `LOAD DATA INFILE` statement.

Invoke `mysqlimport` from the command line as follows:

```
shell> mysqlimport options db_name input_file ...
```

`db_name` names the database containing the table to be loaded and `input_file` names the file that contains the data to be loaded. You can name several input files following the database name if you like.

`mysqlimport` uses each filename to determine the name of the corresponding table into which the file's contents should be loaded. The program does this by stripping off any filename extension (the last period and anything following it) and using the result as the table name. For example, `mysqlimport` treats a file named `City.txt` or `City.dat` as input to be loaded into a table named `City`. After determining the table name corresponding to the filename, `mysqlimport` issues a `LOAD DATA INFILE` statement to load the file into the table.

Each table to be loaded by `mysqlimport` must already exist, and each input file should contain only data values. `mysqlimport` is not intended for processing dump files that consist of SQL statements.

Invoke `mysqlimport` with the `--help` option to see a complete list of the options.

By default, input files for `mysqlimport` are assumed to contain lines terminated by newlines, with each line containing tab-separated data values. This is the same default format assumed by the `LOAD DATA INFILE` statement.

Here are some of the most common options:

506

- `--lines-terminated-by=string` - Specifies character sequence that each input line ends with--default is `\n`
- `--fields-terminated-by=string` - specifies the delimiter between data values within input lines--default is `\t` (tab)
- `--fields-enclosed-by=char` - Use if values are quoted--common value for char is the double quote character (`"`)
- `--ignore` or `--replace` - Handling of input records that contain unique key values that are already present in the table
- `--local` - allows use of a data file that's located locally on the client host



507

The preceding options give you the flexibility to load input files containing data in a variety of formats. Some examples follow which load an input file named `City.txt` into the `City` table in the `world` database.

```
shell> mysqlimport --lines-terminated-by="\r\n" world City.txt
shell> mysqlimport -fields-terminated-by=,
                  --lines-terminated-by="\r" world City.txt
shell> mysqlimport --fields-enclosed-by=''' world City.txt
```

508

17.3.3 Import Data with the SOURCE Command

New data can be imported into a database with the `mysql` command line client using a SQL script file containing all the SQL statements needed to create tables, etc. To execute a SQL command file use the following command syntax;

```
shell> mysql [<options>] < filename
```

And then use **SOURCE** (at the `mysql` prompt) to repopulate the tables:

```
mysql> SOURCE /path/to/file
```

For example, you can import the `CountryLanguage.sql` file back into the `CountryLanguage` table of the `world` database:

```
shell> mysql -u root world < C:/CountryLanguage.sql
```

And then **SOURCE** the file:

```
mysql> SOURCE C:/CountryLanguage.sql
```





Further Practice

In this exercise, you will use various methods to export and import table data using the mysql client and the DOS command line, with the world database.

509

1. From the system shell make a backup of the world database with mysqldump. View the back-up file.
2. Create another file with the --skip-opt flag for mysqldump and compare the two files. What is the difference?
3. Enter the mysql client and create a new database called world_old
4. Copy the contents of world to world_old with your mysqldump file (from above).
5. Use mysqldump with the --tab=/path option to create a backup of the City table.
6. Copy the contents of the City table into a file (City.txt) using **SELECT INTO OUTFILE**.
7. Compare the files created from steps 5 and 6 above.
8. Create a table called City_short with the following columns: Name CHAR(35), Country CHAR(52), kPopulation INT.
9. Import the data from the 'City.txt' file into the newly created table City_short using **LOAD DATA INFILE** such that: the Name column gets the name of the city, the Country column is the name of the country and the kPopulation column, is the population of the city in thousands (i.e., if the population is 2 000 000 you store 2 000 in the column). Verify that the operation is successful.



17.4 Chapter Summary

510

This chapter introduced Exporting and Importing Data in MySQL. In this chapter, you learned to:

- Import data using SQL
- Export data using SQL
- Export using the ‘mysqldump’ database backup client
- Import using the ‘mysqlimport’ client
- Import data with the SOURCE command



18 STORED ROUTINES

18.1 Learning Objectives

512

This chapter introduces **Stored Routines** in MySQL. In this chapter, you will learn to:

- Define a stored routine
- Differentiate between stored procedures and stored functions
- Create stored routines
- Execute stored routines
- Examine an existing stored routine
- Delete an existing stored routine
- Create stored routines with compound statements
- Assign variables in stored routines
- Create flow control statements
- Declare and use handlers
- Cursor usage and limitations



18.2 What is a Stored Routine?

A **stored routine** is a set of SQL statements that can be stored in the server. Once this has been done, clients do not need to keep reissuing the individual statements but can refer to the stored routine instead. There are two types of stored routines:

513

1. **Stored Procedures** – A series of instructions stored within the database itself that acts upon the instructions but does not return a value. A procedure is invoked using a **CALL** statement, and can only pass back values using output variables.
2. **Stored Functions** – A series of instructions stored within the database itself that returns a single value. A function can be called from inside a statement just like any other function (that is, by invoking the function's name), and can return a scalar value.

Some situations where stored routines can be particularly useful:

514

- **Client Applications** – Stored routines give developers the ability to create a statement in one application (MySQL) that can be utilized in multiple client applications that may require writing programs different programming languages or work on different platforms.
 - **One Application** – Embedding "business logic" in MySQL using stored routines eliminates duplicating the same logic in the programs that access the data. This allows the creation and maintenance of the stored routines involved to be simplified tone location and language which helps minimize the possibility of data corruption from external systems that may fail in validation logic. This is especially important when the client applications are maintained by other individuals in possibly separate organizations.
 - **One Programming Language** – By using stored routines in MySQL, the developer of the "business logic" only has to produce the program in one location using only one programming language. This minimizes the creation time, implementation and maintenance of the program and provides the highest level of data validation and assurance no matter the application that accesses the data.
- **Security** – Stored routines provide an outlet for those programs that require the highest level of security. Banks, for example, use stored procedures and functions for all common operations. This provides a consistent and secure environment, and routines can ensure that each operation is properly logged. In such a setup, applications and users would have no access to the database tables directly, but can only execute specific stored routines.
 - **Minimal Data Access** – With stored routines, end users can be given as little access to the data as possible for them to accomplish their tasks. This ensures that the complete data in the database is secure and only available to those users that require complete access while giving the specific data or results of data manipulation (such as SUM, AVERAGE, COUNT, etc.) to the end users that require only the minimal access to the data.
 - **Single Location Processing** – With stored routines, the database server has direct access to the data it needs to manipulate and only needs to send the final results back to the user. This eliminates the possibility that large amounts of data being processed could be monitored by external or internal sources and only the final result sets of the data are returned.



- **Performance** – Stored routines provide improved performance because less information needs to be sent between the server and the client.

The trade off is that this does increase the load on the database server because more of the work is done on the server side and less is done on the client (application) side. Consider this if many client machines (such as Web servers) are serviced by only one or a few database servers.

- **Function Libraries** – Stored routines allows for libraries of functions in the database server. These libraries would then act as an API to the database.

Stored Routine Issues

515

- **Increased Server Load** - Executing stored routines in the database itself can increase the server load and reduce the performance of the applications. Testing and common sense need to be applied to ensure that the convenience of having logic in the database itself versus the performance required.
- **Limited Development Tools** - There are currently a limited number of development tools to support stored routines in MySQL. This limitation can make writing and debugging a much more difficult process and needs to be considered in the decision process.
- **Limited Language Functionality and Speed** - Even though having logic in the database itself is a huge advantage in many situations, there is definitely limitations on what can be accomplished in comparison to other programming languages. This needs to also be considered to ensure that the best possible solution is being utilized.
- **Limited Debugging/Profiling Capabilities**

18.3 Creating Stored Routines

As stated in the previous chapter, there are two separate types of stored routines that can be utilized in MySQL: **Procedures** and **Functions**.

18.3.1 Creating Procedures

The minimal that is required to create a procedure is:

516

```
CREATE PROCEDURE procedure_name procedure_statement
```

This simple syntax created a procedure with the name identified in the **procedure_name** syntax executing the statement in the **procedure_statement** syntax.

Here is an example of a procedure with a single statement:

```
mysql> CREATE PROCEDURE world_record_count ()  
-> SELECT 'country count ', COUNT(*) FROM Country;
```



18.3.2 Creating Functions

The minimal that is required to create a function is:

```
CREATE FUNCTION function_name RETURNS return_type function_statement
```

This simple syntax created a function with the name identified in the ***function_name*** syntax, the data type of the value to be returned using the **RETURNS *return_type*** syntax, and executing the statement in the ***function_statement*** syntax. A function body needs the **RETURN ()** call to return a scalar value.

The following is an example of a function that takes a parameter, performs an operation using an SQL function, and returns the result:

```
mysql> CREATE FUNCTION ThankYou (s CHAR(20))
->   RETURNS CHAR(50)
->   RETURN CONCAT('Thank You, ',s,'!');
```

Incremental learning

There are many ways to use procedures and functions. This chapter incrementally builds on the details of procedure and functions usage.





InLine Lab 18-A

In this exercise you will create two stored **routines**: a **stored procedure** and a **stored function**.

Step 1. Create and test 'hello world' procedure

1. Using the **world** database, create the following stored procedure:

```
mysql> CREATE PROCEDURE hello()
->   SELECT 'Hello world!';
```

A stored procedure is created that will output ‘Hello world!’ when executed.

2. Enter the following statement:

```
mysql> CALL hello();
```

The following output is produced:

```
+-----+
| Hello world! |
+-----+
| Hello world! |
+-----+
1 row in set (#.## sec)
```

Step 2. Create and test 'hello world' function

1. Create the following stored function:

```
mysql> CREATE FUNCTION hello(input CHAR(20))
->   RETURNS CHAR(50)
->   RETURN CONCAT('Hello, ', input, '!');
```

A stored function is created that when called will return the words Hello and the text value that is defined by the parameter name **input** and the value of **CHAR(20)**. The return value can not be greater than 50 characters as stated with the statement **RETURNS CHAR(50)**.

2. Enter the following statement:

```
mysql> SELECT hello('world');
```

The following output is produced:

```
+-----+
| hello('world') |
+-----+
| Hello, world! |
+-----+
```



18.4 Compound Statements

Most stored routines that developers will want to design in MySQL will require multiple lines of code to accomplish the task. In the introduction to stored routines chapter, the stored routines presented were small applications and did not require the use of multiple statements to accomplish their tasks. MySQL offers a way to write multiple statements in stored routines using **BEGIN ... END** syntax.

```
DELIMITER
```

517 Within the **BEGIN ... END** syntax, statements must be terminated with a semicolon (;). This is due to the fact that this is the same default terminating character for SQL statements, it is important to change the SQL terminating character sequence with the **DELIMITER** statement when using the MySQL command line client or batch processing.

```
DELIMITER //  
...  
DELIMITER ;
```

The first **DELIMITER** statement makes the terminating SQL statement character two forward slashes (//). This change allows the terminating character for the stored routine compound statement to be the semicolon (;). After the stored routine with compound statements is created, the second **DELIMITER** statement returns the terminating SQL character back to a semicolon (;).

two forward slashes (//) is a good terminating character to use .Backward slashes (\) are not recommended due to their use in regular SQL syntax.

```
BEGIN ... END
```

The **BEGIN ... END** syntax can be used in stored routines and triggers. The compound statements within the **BEGIN ... END** syntax can contain one or more statements (there is no limit) or no statements (an empty compound statement is legal).Here is an example of a compound statement stored procedure:

```
mysql> DELIMITER //  
mysql> CREATE PROCEDURE world_record_count ()  
    -> BEGIN  
    ->     SELECT 'country count ', COUNT(*) FROM country;  
    ->     SELECT 'city count ', COUNT(*) FROM city;  
    ->     SELECT 'CountryLanguage count', COUNT(*) FROM CountryLanguage;  
    -> END//  
mysql> DELIMITER ;
```

Using indentation, similar to that used in the above example, will make code easier tread. This can help locate problems in the event that the original code does not work or does not work as expected.

- **BEGIN ... END** blocks can be nested.
- **BEGIN ... END** blocks can have labels, like **WHILE/REPEAT/LOOP**





InLine Lab 18-B

In this exercise you will create two stored routines with **compound statements**.

Note: Due to the amount of typing and possible editing that is required in this chapter, it may be helpful to use a tool like Notepad in Windows or another text application for cut-and-paste operations.

Step 1. Create a stored procedure that interacts with the world database tables

1. Using the **world** database, enter the following statement:

```
mysql> DELIMITER //
```

This statement sets the SQL statement terminating character to two forward slashes (//).

2. Enter the following statements:

```
mysql> CREATE PROCEDURE country_info (IN c_code CHAR(3))  
-> BEGIN  
->   SELECT * FROM City WHERE CountryCode = c_code;  
->   SELECT * FROM Country WHERE Code = c_code;  
->   SELECT * FROM CountryLanguage WHERE CountryCode = c_code;  
-> END//
```

This procedure will display records from each relevant table in the world database for a single country.

3. Enter the following statements:

```
mysql> DELIMITER ;
```

This statement returns the SQL statement terminator back to a semicolon (;).

Step 2. Create a stored procedure that interacts with the world database tables

1. Enter the following statement to call the stored procedure created in step 1:

```
mysql> CALL country_info ('NLD');
```

The **country_info procedure** is executed using **NLD** as the **c_code** input (country code).

2. Call the same stored procedure again, but this time use a different terminating delimiter:

```
mysql> CALL country_info ('NLD')\G
```

The **country_info** procedure is executed again using **NLD** as the country code but now the results from the country table (2nd select statement in the **country_info procedure**) are easier to view. However, the 1st and 3rd select statements were easier to read when using the standard view of results (;).

Due to \G being a client command for MySQL, it is not changed with the **DELIMITER** statement. Thus you can not use \G in compound statements to produce an easier viewable record set for results with a large number of columns.



18.5 Assign Variables

DECLARE

This statement is used to declare local variables and can be utilized in stored routines to initialize user variables. The **DEFAULT** clause can be added to the end of the **DECLARE** statement to identify what the initial value for the user variable should be. If the **DEFAULT** clause is left out, the initial value for the user variable is **NULL**.

518

```
mysql> DELIMITER //
mysql> CREATE FUNCTION add_tax (total_charge FLOAT(9,2))
    -> RETURNS FLOAT(10,2)
    -> BEGIN
    ->     DECLARE tax_rate FLOAT (3,2) DEFAULT 0.07;
    ->     RETURN total_charge + total_charge * tax_rate;
    -> END//
mysql> DELIMITER ;
```

There are some major differences between **local** and **user** variables that should be noted:

- **DECLARING - local** variables have to be explicitly declared while **user** variables do not
- **SCOPE - local** variables are local to the instance of the routines while the **user** variables provide more flexibility in scope by being global and usable to the whole session/connection

SELECT ... INTO

The **SELECT ... INTO syntax** stores query results directly in to user defined variables. These user defined variables can be global and usable to the whole session or local.

Session Variables

519

```
mysql> SELECT SUM(population) FROM Country INTO @WorldPop;
```

... is equivalent to...

```
mysql> SELECT SUM(population) INTO @WorldPop FROM Country;
```

Local Variables

```
mysql> SELECT COUNT(*) FROM City INTO TotalCities;
```

... is equivalent to...

```
mysql> SELECT COUNT(*) INTO TotalCities FROM City;
```

The **SELECT ... INTO** statement is extremely useful in stored functions because a stored function can not return result sets like stored procedures. For Local Variables, it is best practice **to DECLARE** the variables prior to use.



SET

The **SET** statement allows the user to assign a value to a user defined variable using either **=** or **`:=`** as the assignment operator.

520

```
mysql> DELIMITER //
mysql> CREATE FUNCTION final_bill (total_charge FLOAT(9,2),
-> tax_rate FLOAT (3,2))
-> RETURNS FLOAT(10,2)
-> BEGIN
->     DECLARE Bill FLOAT(10,2);
->     SET Bill=total_charge + total_charge * tax_rate;
->     RETURN Bill;
-> END//
mysql> DELIMITER ;
```

With the **SET** statement, a user can set multiple user variables in the same statement using commas to separate each variable.

Warning! Use different prefixes for different type of variables to avoid name clashes. (e.g. `p_foo` for parameters, `l_foo` for local variables).

Variable Scope

It is possible for the same identifier to be used for a routine parameter, a local variable, and a table column. Also, the same local variable name can be used in nested blocks. For example:

521

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE precedence (param1 INTEGER)
-> BEGIN
->     DECLARE var1 INT DEFAULT 0;
->     SELECT 'outer1', param1, var1;
->     BEGIN
->         DECLARE param1, var1 CHAR(3) DEFAULT 'abc';
->         SELECT 'inner1', param1, var1;
->     END;
->     SELECT 'outer1', param1, var1;
-> END//
mysql> DELIMITER ;
```

In such cases the identifier is ambiguous and the following precedence rules apply:

- A routine parameter takes precedence over a table column
- A local variable takes precedence over a routine parameter or table column
- A local variable in an inner block takes precedence over a local variable in an outer block

The behavior that table columns do not take precedence over variables is non-standard.



18.6 Parameter Declarations

Stored routines can include **parameter declarations** which enable the caller to pass values to the routines. For stored procedures, the procedure can also pass values back to the caller at the termination of the procedure.

Stored Procedure Parameters

For stored procedures there are three possible procedure parameter declarations that control the information that will be passed or be retrieved from the procedure:

522

1. **IN** – indicates an input parameter which is passed in from the caller to the procedure. The following syntax displays a stored procedure using an **IN** parameter:

```
CREATE PROCEDURE procedure_name (IN parameter_name parameter_type)
procedure_statement
```

The **parameter_name** is accessible within the procedure statement to retrieve the value passed in by the parameter and the **parameter_type** tell the procedure how this value should be seen by the procedure. Each parameter can be declared to use any valid data type, except for the **COLLATE** attribute. The most common are **INT**, **CHAR** and **VARCHAR**.

IN is the default parameter declaration if no parameter declaration precedes the parameter.

2. **OUT** – indicates an output parameter which is set by the procedure and passed to the process calling it after the procedure terminates. The following syntax displays a stored procedure using an **OUT** parameter:

```
CREATE PROCEDURE procedure_name (OUT parameter_name parameter_type)
procedure_statement
```

The syntax is identical to the **IN** parameter, except any value that is passed from the caller to the procedure through an **OUT** parameter will be ignored by the procedure.

3. **INOUT** – indicates a parameter that can act as an **IN** and an **OUT** parameter. Any value that is passed from the caller to the procedure sets the initial value of the parameter which can be changed by the procedure and passed to the caller after the procedure terminates. The following syntax displays a stored procedure using an **INOUT** parameter:

```
CREATE PROCEDURE procedure_name (INOUT parameter_name parameter_type)
procedure_statement
```

The **INOUT** syntax is identical to the **IN** parameter; however the value passed into the **INOUT** parameter can be utilized by the procedure statement and passed out at the end of the procedure statement making this parameter the most versatile.

Stored Function Parameter

For stored functions there is only one possible procedure parameter declaration: **IN**, and thus is set by default and is not allowed to precede the parameter value.





InLine Lab 18-C

In this exercise you will create two stored **routines** with **parameter declarations**.

Step 1. Create and test a stored procedure with parameter declarations

- Using the **world** database, enter the following to create a stored procedure that will return the country population in percentage to the world population:

```
mysql> DELIMITER //
mysql> CREATE FUNCTION pop_percentage (c_code CHAR(3))
-> RETURNS DECIMAL (4,2)
-> BEGIN
-> DECLARE worldpop, cpop BIGINT;
->     SELECT SUM(population) FROM country INTO worldpop;
->     SELECT population FROM country WHERE CODE = c_code INTO cpop;
->     RETURN cpop / worldpop * 100;
-> END///
mysql> DELIMITER ;
```

This function, **pop_percentage**, will return the country population in percentage to the world population. Within the compound statement there are two select statements: the user-variable **@WorldPop** is assigned the value of the entire world population and the user variable **@CountryPop** is assigned the value of the countries population based on the country code that is sent into the function. The value returned is a decimal that represents the percentage that countries population is to the whole world.

- Enter the following statements to test the stored procedure just created:

```
mysql> SELECT pop_percentage ('CHN');
```

This select statement will execute the **pop_percentage** function using the population numbers for the country of China. The results are listed below:

```
+-----+
| pop_percentage ('CHN') |
+-----+
|          21.02 |
+-----+
1 row in set, 1 warning (#.## sec)
```

This result tells us that China's population makes up 21.02% of the world's population.

Note: The warning is simply a note telling us that the output value was truncated because of the return value being sent to the output was defined at **DECIMAL (4,2)**.



Step 2. Create and test a stored procedure with multiple parameter declarations

- Using the `world` database, enter the following to create a procedure called `param_proc_one` that will display three (3) values passed into it:

```
mysql> CREATE PROCEDURE param_proc_one
->   (IN param1 INT, OUT param2 INT, INOUT param3 INT)
->   SELECT param1, param2, param3;
```

A procedure called `param_proc_one` is created that will display the three (3) values; `param1`, `param2`, `param3`, that will be passed from the caller to the procedure.

- Assign values to three different user variables by entering the following statement:

```
mysql> SET @value1 = 100, @value2 = 200, @value3 = 300;
```

Values are assigned to three (3) user variables for this current session.

- Call the stored procedure `param_proc` using the three variable assigned:

```
mysql> CALL param_proc_one (@value1,@value2, @value3);
```

The three (3) user variables set in step 3 are passed to the `param_proc_one` procedure. The following output is sent to the screen:

```
+-----+-----+-----+
| param1 | param2 | param3 |
+-----+-----+-----+
|    100 |     NULL |     300 |
+-----+-----+-----+
1 row in set (#.# sec)
```

In the case of `param2`, which was declared as an `OUT` parameter, the value is `NULL` because the variable value passed was ignored by the procedure.

Step 3. Create and test a stored procedure that will alter user variables

- Using the `world` database, enter the following to create a procedure called `param_proc_two` that will be used to alter the variables that were created in the last step:

```
mysql> CREATE PROCEDURE param_proc_two
->   (IN param1 INT, OUT param2 INT, INOUT param3 INT)
->   SET param1 = 1 , param2 = 2, param3 = 3;
```

A procedure called `param_proc_two` is created that will set the three (3) values; `param1`, `param2`, `param3`, 1, 2 and 3 respectively.



2. Utilize the three (3) user variable again in the following statement:

```
mysql> CALL param_proc_two(@value1,@value2, @value3);
```

The three (3) user variables assigned earlier are used again with the `param_proc_two` procedure. Due to the design of the procedure, there is no output from the procedure.

3. Enter the following statement to display the values that have been assigned to the three (3) values

```
mysql> SELECT @value1, @value2, @value3;
```

This select statement displays the values that have been assigned to the three (3) values; `@value1`, `@value2`, `@value3`, and demonstrates how `param_proc_two` affected `@value2` and `@value3` but does not change the `@value1` value.

```
+-----+-----+-----+
| @value1 | @value2 | @value3 |
+-----+-----+-----+
| 100    | 2      | 3      |
+-----+-----+-----+
1 row in set (#.## sec)
```



18.7 Execute Stored Routines

As shown previously, the commands for calling stored routines are very similar to other commands in MySQL. A procedure is invoked using a `CALL` statement, and can only pass back values using output variables. A function can be called from inside a statement just like any other function (that is, by invoking the function's name), and can return a scalar value.

In MySQL, a stored procedure or function is associated with a particular database. This has several implications:

523

- **USE Database** - When the routine is invoked, an implicit `USE db_name` is performed (and undone when the routine terminates). `USE` statements within stored routines are disallowed.
- **Qualify Names** – Routine names can be qualified with the database name. This can be used to refer to a routine that is not in the current database. For example, to invoke a stored procedure `p` or function `f` that is associated with the test database, `CALL test.p()` or `test.f()` can be used.
- **Database Deletions** - When a database is dropped, all stored routines associated with it are dropped as well.

MySQL supports the very useful extension that allows the use of regular `SELECT` statements inside a stored procedure. The result set of such a query is simply sent directly to the client.

Multiple result sets

Multiple `SELECT` statements generate multiple result sets, so the client must use a MySQL client library that supports multiple result sets. This is supported by the mysql command line client.



18.8 Examine Stored Routines

With the ability to create a multitude of stored routines within each database, it is imperative to have the ability to be able to review the stored routines syntax that have already been created. There are three different commands that are available in MySQL to accomplish this task:

524

1. **SHOW CREATE PROCEDURE** and **SHOW CREATE FUNCTION**

These statements are MySQL extensions and are similar to SHOW **CREATE TABLE**. These statements will return the exact string that can be used to re-create the named routine. One of the main limitations of these statements is the fact that the user must know the name of the procedure or function and must know if it is a procedure or function before the user can attempt to view the information.

2. **SHOW PROCEDURE STATUS** and **SHOW FUNCTION STATUS**

This statement is a MySQL extension. It returns characteristics of routines, such as the database, name, type, creator, creation and modification dates. These statements have the advantage of being able to display specific routines based on a **LIKE** pattern. If no pattern is specified, the information for all stored procedures or all stored functions is listed, depending on which statement is used.

The disadvantage to these statements is that they do not show the actual syntax of the routines that are in the database, only the status information.

3. **INFORMATION_SCHEMA.ROUTINES** table

The **ROUTINES** table provides information about stored routines (both procedures and functions) and returns the majority of the details that could be found in both the **SHOW CREATE ...** and **SHOW ... STATUS** statements to include the actual syntax used to create the stored routines. Of the three options, this table holds the most complete picture of the stored routines available in the databases.





InLine Lab 18-D

In this exercise you will examine the stored routines in your database using the three (3) different approaches: **SHOW CREATE ...**, **SHOW ... STATUS**, and the **INFORMATION_SCHEMA.ROUTINES** table.

Step 1. Display the details of a stored procedure using SHOW

1. In the MySQL command client, using the world database, type:

```
mysql> SHOW CREATE PROCEDURE param_proc_one\G
```

The one (1) record output displays the procedure name, the sql mode(s) that are used, and the text used to create the procedure.

Hint: (\G) displays the output of that record in row format which can be easier for reading when the length of the columns returned is greater than the width of your window.

DEFINER?

You may notice that the stored procedure starts off with the words **CREATE DEFINER= ...**. A stored routine runs either with the privileges of the user who created it or the user who invoked it. The choice of which set of privileges to use is controlled by the value of the **SQL SECURITY** characteristic. This characteristic can take one of two values:

DEFINER – This value causes the routine to have the privileges of the user who created it, and it is the default – or –

INVOKER – This value causes the routine to run with the privilege of the user who invoked or is running the stored routine. This means the routine has access to database objects only if the invoker (user) already has access to them.

For the purposes of this course, this topic is of a more advanced nature. The MySQL training team offers additional training in creating and utilizing stored procedures if you wish to learn more advanced topics of using these features in your database development.

Step 2. Display the details of a stored function using SHOW

1. Type the following statement:

```
mysql> SHOW CREATE FUNCTION hello\G
```

Similar to the stored procedure output, this output displays the function name, the sql mode(s) that are used, and the text used to create the function.



Step 3. Display the details of all stored procedures associated with a specific database

1. Type the following statement:

```
mysql> SHOW PROCEDURE STATUS WHERE Db = 'world'\G
```

All the procedures associated with the world database are displayed in the MySQL client. The field list includes the database name, the name of the procedure, the type of stored routine, the definer for the routine, the last date the routine was modified, the date the routine was created, the security type that the routine is using and any comments that are associated with the routine.

Note: Without the **WHERE** clause in this statement, all user defined procedures associated with the server would have been displayed.

Step 4. Display the details of all stored functions associated with a specific database

1. Type the following statement:

```
mysql> SHOW FUNCTION STATUS WHERE Db = 'world'\G
```

All user defined functions associated with the server are sent to the MySQL client window. The column list is identical to the **SHOW PROCEDURE STATUS** statement.

Step 5. Utilize the INFORMATION_SCHEMA database

1. Type the following statement to see a complete list of procedures and functions associated with the **world** database:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.ROUTINES  
-> WHERE ROUTINE_SCHEMA = 'world'\G
```

A complete list of procedures and functions associated with the world database are sent to the MySQL command client.

Note: The list of columns is extensive and the details of each of these columns go beyond the scope of this class. However, many of the columns are similar to those that were displayed in the **SHOW CREATE ...** and **SHOW ... STATUS** statements.



18.9 Delete Stored Routines

Deleting, or dropping, a stored routine is an important aspect of managing stored routines and is easily handled. To delete an existing stored procedure, use `DROP PROCEDURE procedure_name`. To delete an existing stored function, use `DROP FUNCTION function_name`. If you are unsure if a stored routine exists, place the `IF EXISTS` clause prior to the `procedure_name` or `function_name`. For example:

525

```
mysql> DROP PROCEDURE proc_1;  
mysql> DROP FUNCTION IF EXISTS func_1;
```





InLine Lab 18-E

In this exercise you will **delete a stored routine** and **recreate** it with new SQL code.

Step 1. Attempt to delete a stored function that does not exist

1. In the MySQL command client, using the **world** database, type:

```
mysql> DROP FUNCTION hello2you;
```

An error is displayed in the MySQL command client window:

```
ERROR 1305 (42000): FUNCTION world.hello2you does not exist
```

2. Type the following statement that includes the clause “**IF EXISTS**”:

```
mysql> DROP FUNCTION IF EXISTS hello2you;
```

This time the statement completes, but a warning has been issued.

3. Type the following statement to display the warning that was issued

```
mysql> SHOW WARNINGS\G
```

The output displays a warning that was issued after executing the statement in step 2:

```
***** 1. row *****
Level: Note
Code: 1305
Message: FUNCTION hello2you does not exist
1 row in set (#.## sec)
```

Step 2. Drop and recreate an existing stored function

1. Type the following to display the function called “**hello**”:

```
mysql> SHOW CREATE FUNCTION hello\G
```

The output displays the function name, the sql mode(s) that are used, and the text used to create the function.

2. Delete the function called “**hello**”:

```
mysql> DROP FUNCTION IF EXISTS hello;
```

The drop statement completes without any warnings.



3. Recreate the function called “**hello**” with the following statement:

```
mysql> CREATE FUNCTION hello (input CHAR(20))
-> RETURNS CHAR(100)
-> RETURN CONCAT('Hello, ', input, '! Welcome to MySQL training.');
```

The hello function has been recreated with new text and a larger possible **CHAR** return (100).

4. Enter the following statement to test the recreated function:

```
mysql> SELECT hello ('Bob');
```

The words 'Hello, Bob! Welcome to MySQL training.' are displayed in the MySQL command client window.

Note: Please feel free to replace the name *Bob* with your own name.



18.10 Flow Control Statements

526

Flow control is defined as the statements and other constructs that control the order in which operations are executed. Most high level programming languages have control flow statements which allow variations in this sequential order. The two common flow controls are:

- **Choices** – statements that are obeyed under certain conditions. In MySQL, these are represented in the **IF** and **CASE** statements.
- **Loops** - statements that are obeyed repeatedly. In MySQL these are represented in the **REPEAT**, **WHILE** and **LOOP** statements.

MySQL SQL statements can utilize many of the same flow control syntax outside of stored routines; however, there are some very notable differences between stored routine flow control statements and SQL syntax flow control statements. The following discussions and examples are designed for using flow control statements in stored routines.

IF

527

In most programming languages the **IF** statement is the most basic of all choice flow controls or conditional constructs.

```
IF (test_condition) THEN  
...  
ELSEIF (test_condition) THEN  
...  
ELSE  
...  
END IF
```



CASE

528

The **CASE** statement for stored routines provides a means of developing complex conditional constructs. The **CASE** choice works on the principle of comparing a given value with specified constants and acting upon the first constant that is matched. If the constants form a compact range then this can be implemented very efficiently as if it were a choice based on whole numbers.

```
CASE case_value WHEN when_value THEN  
...  
ELSE  
...  
END CASE
```

Or:

```
CASE WHEN test_condition THEN  
...  
ELSE  
...  
END CASE
```

Unlike other languages, no **BREAK/CONTINUE** is needed inside **CASE** statements to not execute following **WHEN** branches.

REPEAT

529

The **REPEAT** statement repeats an SQL statement until the search condition that is set becomes true. A **REPEAT** statement is always run at least once.

The label option can be part of the **REPEAT** statements. A label is an identifier, which is attached to a statement by using a colon (:). A **REPEAT** statement can use both a label at the beginning (*mylabel*) and label at the end. An end label is optional but when used cannot be given unless a beginning label also is present. If both are present, they must be the same (in this case *mylabel*).

```
mylabel: REPEAT  
...  
UNTIL test_condition  
END REPEAT mylabel
```



WHILE

530

The statement list within a **WHILE** statement is repeated as long as the search_condition is true. The statement list can consist of one or more statements. The **WHILE** statement can also utilize the label **options**. **WHILE** repeats an SQL statement while a set condition is true.

```
mylabel: WHILE test_condition DO
  ...
END WHILE mylabel
```

LOOP

531

The **LOOP** statement implements a simple loop construct, enabling repeated execution of the statement list, which consists of one or more statements. The statements within the loop are repeated until the loop is exited; usually this is accomplished with a **LEAVE** statement. The **LOOP** statement can also utilize the label options.

```
mylabel: LOOP
  ...
  LEAVE mylabel;
END LOOP mylabel
```

The labels at the beginning and end can *only* be used in loop constructs and **BEGIN ... END blocks**. **Labels** are *not* required.

A **LOOP** is often terminated with a **RETURN** statement in stored functions.

Other Label Flow Control Constructs:

532

As stated earlier, the label option has potential for enhancing the flow control constructs mentioned here. MySQL offers two (2) additional constructs that utilize the label option:

- **LEAVE** – This statement is used to exit any labeled flow control construct within a **LOOP**, **REPEAT** or **WHILE** statement. **LEAVE** can also be used in labeled **BEGIN ... END** compound statements. It can jump to any label name. It does not have to be the label name of the **LOOP**.
- **ITERATE** – This statement simply means “do the **LOOP** (or **REPEAT** or **WHILE**) again.”

Transactions are possible in stored routines, but must use the **START TRANSACTION** keyword as **BEGIN** is already a keyword in stored routines.





InLine Lab 18-F

In this exercise you will make create stored routines using some of the **flow control statements** explained.

Step 1. Create and test a stored function using a CASE flow control statement

1. In the MySQL command client, using the **world** database, type the following statements:

```
mysql> DELIMITER //
mysql> CREATE FUNCTION docase (1st_Num DOUBLE(10,2),2nd_Num DOUBLE(10,2),
-> formula_type CHAR(15))
-> RETURNS CHAR(30)
-> BEGIN
->   CASE formula_type WHEN 'Multiplication' THEN
->     RETURN CONCAT(1st_Num,' * ',2nd_Num,' = ',1st_Num * 2nd_Num);
->   WHEN 'Division' THEN
->     RETURN CONCAT(1st_Num,' / ',2nd_Num,' = ',1st_Num / 2nd_Num);
->   WHEN 'Addition' THEN
->     RETURN CONCAT(1st_Num,' + ',2nd_Num,' = ',1st_Num + 2nd_Num);
->   WHEN 'Subtraction' THEN
->     RETURN CONCAT(1st_Num,' - ',2nd_Num,' = ',1st_Num - 2nd_Num);
->   ELSE
->     RETURN 'Invalid Formula Type';
-> END CASE;
-> END//
-> DELIMITER ;
```

This function demonstrates the capabilities of using the **CASE** flow control statement. In this function, the user will enter in two numbers that must be in a double format with or without decimals. (If decimals are added, the system will truncate any numbers past two digits. If decimals are not added, the system will add two decimals to the number 0.00). The third input is the type of mathematical operation the user wishes to perform on the numbers.

The function will select which mathematical formula to utilize using the **CASE** flow control statement and return the proper mathematical formula to the screen.

Note: In this function, the mathematical formula type is case sensitive.

2. Test the **docase** stored function with the following statement that performs addition:

```
mysql> SELECT docase (10,5,'Addition');
```

This select statement uses the **docase** function created in Step 1 to perform the addition mathematical formula on the numbers 10 and 5.



3. Test the **docase** stored function with the following statement that performs division:

```
mysql> SELECT docase (22.3785, 12.3,'Division');
```

In this statement, the **docase** function truncates the 1st number t22.38 and expands the 2nd number t12.30 and performs division on the two numbers.

4. Test the docase stored function with the following statement which attempts to perform multiplication:

```
mysql> SELECT docase (15, 5,'multiplication');
```

In this statement, the **docase** function gives the following response:

```
Invalid Formula Type
```

The reason this function did not perform a multiplication mathematic formula is because the formula type was all lower case and thus the **ELSE** portion of the **CASE** flow control statement was used.

Step 2. Create and test a stored procedure using a REPEAT flow control statement

1. In the MySQL command client, using the **world** database, type the following statements:

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE dorepeat (p1 INT)
-> BEGIN
->   DECLARE var_x INT;
->   SET var_x = 0;
->   REPEAT SET var_x = var_x + 1;
->   SELECT CONCAT('The x variable is', var_x) AS Repeat_Counter;
->   UNTIL var_x > p1
->   END REPEAT;
->   SELECT CONCAT('The final REPEAT number is ',var_x)
->   AS Repeat_Results;
->   END///
mysql> DELIMITER ;
```

This procedure demonstrates the capabilities of using the **REPEAT** flow control statement.

In this procedure, the user enters an integer that is then passed to the procedure. The procedure counts from 0, incrementing by 1, until it reaches the value of the number entered. The procedure then tells the user what number it ended on and exits the procedure.

2. Test the **dorepeat** stored procedure with the following statement:

```
mysql> CALL dorepeat (10);
```

In this statement, the **dorepeat** procedure outputs the last value that the internal delimiter was set to.



Step 3. Create and test a stored procedure using a WHILE flow control statement

1. In the MySQL command client, using the **world** database, type the following statements:

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE dowhile (p1 INT)
      -> BEGIN
      -> DECLARE var_x INT;
      -> SET var_x = 0;
      -> WHILE var_x < p1 DO
      ->     SET var_x = var_x + 1;
      ->     SELECT CONCAT('The x variable is', var_x) AS While_Counter;
      -> END WHILE;
      -> SELECT CONCAT('The final WHILE number is ', var_x)
      ->     AS While_Results;
      -> END//
mysql> DELIMITER ;
```

The procedure is similar to the **dorepeat** procedure created in an earlier step, but uses the **WHILE** flow control statement.

2. Test the **dowhile** stored procedure with the following statement:

```
mysql> CALL dowhile (10);
```

In this statement, the **dowhile** procedure outputs the last value that the internal delimiter was set to.

Are the outputs from the **dorepeat** and **dowhile** different? If so, why?



Step 4. Create and test a stored procedure using a LOOP flow control statement

1. In the MySQL command client, using the **world** database, type the following statements:

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE doloopif (p1 INT)
      -> BEGIN
      -> DECLARE var_x INT;
      -> SET var_x = 0;
      -> loop_test: LOOP
      ->     IF var_x < p1 THEN
      ->         SET var_x = var_x + 1;
      ->     ELSE
      ->         LEAVE loop_test;
      ->     END IF;
      -> END LOOP loop_test;
      -> SELECT CONCAT('The final LOOP and IF number is ',var_x) AS Results;
      -> END//
mysql> DELIMITER ;
```

This procedure, **doloopif**, is similar to both **dowhile** and **dorepeat**, yet uses the **LOOP** and **IF** flow control statements.

2. Test the **doloopif** stored procedure with the following statement:

```
mysql> CALL doloopif (1000);
```

In this statement, the **doloopif** procedure outputs the last value that the internal delimiter was set to.



533

18.11 Declare and Use Handlers

Certain conditions may require specific handling. These conditions can relate to errors, as well as to general flow control inside a routine. When a condition and handler is used in a compound statement, the scope of the condition and handler is the compound statement itself.

DECLARE Statement Syntax

The **DECLARE** statement is used to define various items local to a routine:

- Local variables
- Conditions and handlers
- Cursors

DECLARE is allowed only inside a **BEGIN . . . END** compound statement and *must be at its start*, before any other statements. Declarations must follow a certain order:

- Variables
- Conditions
- Cursors
- Handlers

DECLARE CONDITION

534

The **DECLARE CONDITION** statement specifies conditions that need specific handling. This statement associates a name with a specified error condition.

```
DECLARE condition_name CONDITION FOR condition_value;
```

The **condition_value syntax** can include errors that are associated with an **SQLSTATE** value using the syntax below. In this example, we are declaring a condition called **null_not_allowed** when the **SQLSTATE 23000** error syntax is requested from the system.

```
DECLARE null_not_allowed CONDITION FOR SQLSTATE '23000';
```

This error is called by the system when a null value is trying to be placed into a column (field) that can not accept null values; such as columns assigned as primary key fields. Another way to declare this condition is by using the MySQL error code equivalent, **1048**.

```
DECLARE null_not_allowed CONDITION FOR 1048;
```

The **condition_name** is optional but can be useful for identifying the condition context using verbiage rather than having to memorize error condition numbers or look them up to determine the error being monitored. In addition, the **condition_name** can be used in any subsequent **DECLARE HANDLER** syntax within the same compound statement.



DECLARE HANDLER

535

The **DECLARE ... HANDLER** statement specifies handlers that can handle, or deal with, one or more conditions. The condition, when called by the system, can be handled in one (1) of two (2) possible ways:

- **CONTINUE** – execution of the current stored routine will continue after the statement that caused the handler to be activated, following the **DECLARE CONTINUE HANDLER ...** syntax
- **EXIT** – terminates the current **BEGIN ... END** compound statement (block) where the handler was declared

The **DECLARE ... HANDLER** statement uses the following syntax:

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x = 1;
```

The **SQLSTATE '23000'** in the above statement is the **condition value** and **SET @x = 1** is the **handler statement** that is to be executed when the server identifies the error. There is additional **condition value** syntax that can be utilized with the **DECLARE HANDLER** statements:

- A **condition name** previously specified with a **DECLARE CONDITION** statement
- **SQLWARNING** is shorthand for all **SQLSTATE** codes that begin with 01
- **NOT FOUND** is shorthand for all **SQLSTATE** codes that begin with 02
- **SQLEXCEPTION** is shorthand for all **SQLSTATE** codes not caught by **SQLWARNING** or **NOT FOUND**

1) If a condition is to be ignored, declare a **CONTINUE** handler for it and associate it with an empty block. For example:

```
DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN END;
```

2) Errors that the stored routines come across that are not handled by handlers will escalate to the client. In that case, the execution of the routine is aborted and an error code will be displayed explaining the purpose of the termination.





InLine Lab 18-G

In this exercise you will force an error within a stored procedure to test the capabilities of the **DECLARE CONDITIONS/DECLARE HANDLERS** statements.

Step 1. Create a stored procedure that will be used to test for SQLSTATE '23000'

1. In the MySQL command client, using the **world** database, type the following statements to create a new table:

```
mysql> CREATE TABLE d_table (s1 int, primary key (s1));
```

This statement creates a table in the world database called **d_table**. The **d_table** contains one (1) column called **s1** which is defined as a primary key.

2. Create the following stored procedure called **dohandler**:

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE dohandler ()
-> BEGIN
-> DECLARE dup_keys CONDITION FOR SQLSTATE '23000';
-> DECLARE CONTINUE HANDLER FOR dup_keys SET @garbage = 1;
-> SET @x = 1;
-> INSERT INTO world.d_table VALUES (1);
-> SET @x = 2;
-> INSERT INTO world.d_table VALUES (1);
-> SET @x = 3;
-> END///
mysql> DELIMITER ;
```

This procedure, **dohandler**, uses both a **DECLARE ... CONDITION** and a **DECLARE ... HANDLER** to demonstrate how they work together.

In the first **INSERT INTO** statement, an initial value is entered in the table created in step 1-1. The second **INSERT INTO** will create an error which will initiate the **DECLARE CONTINUE HANDLER** which will set the **@garbage** variable and continue on with the procedure.

Step 2. Test the stored procedure just created

1. Type the following to execute the stored procedure just created in step 1:

```
mysql> CALL dohandler();
```

The **dohandler** procedure is run without any errors.

2. Check the value of the first user variable that was assigned in the execution of the **dohandler** stored procedure:

```
mysql> SELECT @x;
```



The value of the user defined variable, `@x`, that was set in the `dohandler` procedure is displayed.

Note: The value is 3 which identifies that the entire procedure completed. If the error was not handled, the procedure would have terminated prematurely and the value would have remained at 2.

3. Check the value of the second user variable that was assigned in the execution of the `dohandler` stored procedure:

```
mysql> SELECT @garbage;
```

The value of the user defined variable, `@garbage`, that was set in the `dohandler` procedure when the `DECLARE CONTINUE HANDLER` was called is displayed.

Note: If the `HANDLER` was not called, the value would be `NULL`.



18.12 Cursors

536

Cursors are a control structure within stored routines that are for the retrieval of records, one row at a time. The term “cursor” is short for CURrent Set Of Records. Due to the fact that cursors obtain one row at a time, most cursors are used in loops that fetch and process a row within each iteration of the loop. Cursors are currently asensitive, read-only, and non-scrolling. Asensitive means that the server may or may not make a copy of its result table.

Declaring cursors and handlers

537

Cursors must be declared before declaring **handlers**. Note: Variables along with conditions must be declared before declaring either cursors or handlers.

```
DECLARE cursor_name CURSOR FOR select_statement;
```

Multiple cursors may be declared in a routine, but each cursor in a given block must have a unique name. When a cursor is declared within a compound statement, the scope of the cursor is the compound statement itself.

There is a need to identify when the set of records being “retrieved” has reached it's end. This is accomplished through the use of a **HANDLER** called **SQLSTATE '02000'** which identifies when a set of records being retrieved has reached the end. The following is an example of this handlers usage:

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 'yes'
```

OPEN

This statement opens a previously declared cursor.

```
OPEN cursor_name;
```

538

FETCH

This statement obtains the next row (if a row exists) using the specified open cursor, and advances the cursor pointer. If there is no “next row”, it will result in an error.

```
FETCH cursor_name INTO var_name;
```

If the cursor produces multiple columns, the columns must be stored into separate variable names by creating the same number of variables and separating them by commas:

```
FETCH cursor_name INTO var_name1, var_name2, var_name3, ...
```



CLOSE

This statement closes a previously opened cursor. If not closed explicitly, a cursor is closed at the end of the compound statement in which it was declared.

```
CLOSE cursor_name;
```

539

Cursor Limitations

The following is a list of noteworthy limitations on Cursors in MySQL 5.0 (and later):

- Cursors are read-only; they cannot be used to update rows.
- **UPDATE WHERE CURRENT OF** and **DELETE WHERE CURRENT OF** are not implemented, because updatable cursors are not supported.
- Cursors are asensitive. Meaning that the server may or may not make a copy of its result table.
- Cursors are non-scrollable.
- Cursors are not named. The statement handler acts as the cursor ID.
- Only one cursor per prepared statement is allowed. If several cursors are required, several statements must be prepared.
- Cursors cannot be used for a statement that generates a result set if the statement is not supported in prepared mode. This includes statements such as **CHECK TABLES**, **HANDLER READ**, and **SHOW BINLOG EVENTS**.
- Cursors in MySQL work on a row base, you cannot skip multiple rows, you have to fetch row by row.





InLine Lab 18-H

In this exercise you will create a stored function that demonstrates the use of **cursors** and other stored routine features explained up to this point.

Step 1. Create a stored procedure that utilizes cursors

1. In the MySQL command client, using the **world** database, type the following statements to create a new stored procedure called **g_table**:

```
mysql> DELIMITER //
mysql> CREATE FUNCTION g_table (tbl_name CHAR(64))
-> RETURNS INT
-> BEGIN
->     DECLARE result BOOL DEFAULT FALSE;
->     DECLARE show_table CHAR(64);
->     DECLARE doneFlag BOOL DEFAULT FALSE;
->     DECLARE q1 CURSOR FOR
->         SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
->             WHERE TABLE_SCHEMA = DATABASE();
->     DECLARE CONTINUE HANDLER FOR NOT FOUND SET doneFlag = TRUE;
->     OPEN q1;
->     doloop: LOOP
->         FETCH q1 INTO show_table;
->         IF doneFlag THEN
->             LEAVE doloop;
->         END IF;
->         IF show_table = tbl_name THEN
->             SET result = TRUE, doneFlag = TRUE;
->         END IF;
->     END LOOP doloop;
->     CLOSE q1;
->     RETURN result;
-> END //
mysql> DELIMITER ;
```

This function, **g_table**, is responsible for determining if a table exists in the any of the databases on the server. The end user, or application that calls this function, will enter in the table to be checked. If the table exists in the current database, the function will return a one (1). If the table does not exist, the function will return a zero (0).

Note: In this function we are using a select statement in the cursor that pulls a list of tables using the **INFORMATION_SCHEMA** database. The cursor processed the **SELECT** statement one (1) record at a time and will set the **good_table** variable to one (1) if a match is found.



Step 2. Execute the cursors stored function

1. Execute the procedure called **g_table** with the following statement:

```
mysql> SELECT g_table ('howdy');
```

This select statement utilizes the **g_table** function created in step 1-1. In this case the table does not exist and the response is zero (**0**).

2. Execute the procedure called **g_table** again but this time with a valid table name:

```
mysql> SELECT g_table ('Country');
```

This select statement also utilizes the **g_table** function created in step 1. In this case; however, the table does exist and the response is one (**1**).





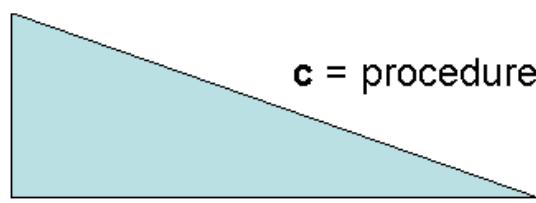
Further Practice

In this exercise, you will use the information covered in this section to implement **stored routines** using the **mysql** client, with the **world** database.

540

1. In the **world** database, create a function called **c_length** that returns the value of a right triangles' longest side (hypotenuse) based on input (from the end user) for the other two sides. Test the function with the values 3 and 4.

a=user # input



b=user # input

Hint: The square root function in MySQL is **SQRT**.

2. Review the details of the **c_length** procedure using the INFORMATION_SCHEMA database.
3. Create a function called **factorial** that calculates the factorial ($n!$) of a given value. Test the function with the values 3 and 10
4. Create a function called **country_name** that takes the three (3) character **country code** of the country to look up as input. The resulting output will be the **country name** based on the **country code**. Test the procedure with the values 'RUS' and 'USA'
5. Use **SHOW CREATE ...** to view the syntax used to create the **country_name** procedure.
6. Create a function **country_capital** that returns the name of the capital in that country, taking the country name as input. Test the function with 'Finland' and 'MySQL'
7. Create a procedure that deletes all information of a country in the world (from all three tables) database based on the country's name. The procedure should report how many rows were deleted from each table. Test it with 'Germany'
8. Create a procedure that inserts a new country into the country table. The values to be given are **Code**, **Name**, **Continent**, and **Population**. If the given Code already exists (duplicate key error) it should be handled with a handler. Test it with a country you make up, and with 'SWE', 'MySQL', 'Europe', 400.
9. Create a function that calculates the *Fibonacci* number of a given value (The Fibonacci numbers, can be described with the following recursive formula: $F(n) = F(n-1)+F(n-2)$, $F(0)=1$, $F(1)=1$). Test the function with 10 and 90.



541

18.13 Chapter Summary

This chapter introduced **Stored Routines** in MySQL. In this chapter, you learned to:

- Define a stored routine
- Differentiate between stored procedures and stored functions
- Create stored routines
- Execute stored routines
- Examine an existing stored routine
- Delete an existing stored routine
- Create stored routines with compound statements
- Assign variables in stored routines
- Create flow control statements
- Declare and use handlers
- Cursor usage and limitations



19 TRIGGERS

19.1 Learning Objectives

543

This chapter introduces **Triggers** in MySQL. In this chapter, you will learn to:

- Describe triggers
- Create new triggers
- Delete existing triggers



544

19.2 What are Triggers?

Database triggers are named database objects that are maintained within a database and are activated when data within a table is modified. Triggers bring a level of power and security to the data within the tables. For instance, using the **world** database, what would you do after changing the code of a country? The country code is stored in the **Country** table, as well as the **City** and **CountryLanguage** tables. Therefore, it would be best to change the code in all three tables simultaneously. A trigger can accomplish this task.

Trigger Features

Triggers give developers and administrators greater control over access to specific data, the ability to perform specific logging or auditing of the data itself. Triggers can be useful for:

- Examining data before it is inserted or updated, or verify deletes or updates
- Acting as a data filter by modifying data if it is out of range, before an insert or update
- Modifying how **INSERT**, **UPDATE**, and **DELETE** behave for a table
- Mimicking the behavior of foreign keys for storage engines that do not support foreign keys
- Logging functionality
- Automatic summary tables

545

19.2.1 Creating Triggers

To define a trigger for a table, use the **CREATE TRIGGER** statement, which has the following syntax:

```
CREATE TRIGGER trigger_name
  { BEFORE | AFTER }
  { INSERT | UPDATE | DELETE }
  ON table_name
  FOR EACH ROW
  triggered_statement
```

trigger_name is the name to give the trigger, and *table_name* is the table with which to associate the trigger. **BEFORE** or **AFTER** indicates whether the trigger activates before or after the triggering event, and **INSERT**, **UPDATE**, or **DELETE** indicates what that event is.

The following example creates a trigger named **DeletedCity** that activates on deletes from the **City** table (from the **world** database):

```
mysql> CREATE TRIGGER City_AD AFTER DELETE ON City
    -> FOR EACH ROW
    -> INSERT INTO DeletedCity (ID, Name) VALUES (OLD.ID, OLD.Name);
Query OK, 0 rows affected (#.# sec)
```



19.2.2 Triggers Events

546

Triggers are associated with individual tables. The method for activating triggers is called “events” and the following list describes those available:

- **BEFORE** – these types of events are based on activation times that take place before changes to the data in the table are written to the underlying database. These types of events can capture improper data entries and correct them or reject prior to storing them. There are three activation events associated with the **BEFORE** activation time:
 - **BEFORE INSERT** – this event is triggered prior to new data being added to the table.
 - **BEFORE UPDATE** – this event is triggered prior to existing data being updated (or overwritten) with new data.
 - **BEFORE DELETE** – this event is triggered prior to data being deleted from the table.
- **AFTER** – these types of events are based on activation times that take place after changes to the data in the table are written to the underlying database. These types of events can be used for logging or auditing the modification of data within the databases. There are three activation events associated with the **AFTER** activation time:
 - **AFTER INSERT** – this event is triggered after new data was added to the table.
 - **AFTER UPDATE** – this event is triggered after existing data has been updated (or overwritten) with new data.
 - **AFTER DELETE** – this event is triggered after data has been deleted from the tables.
- Foreign keys with cascading behavior currently do NOT activate triggers.

The procedure to create and use the above **City_AD** trigger is as follows:

547

1) First, check to see if a trigger of this type/name already exists. In this case, there is no trigger.

```
mysql> SHOW TRIGGERS\G
Empty set (#.## sec)
```

2) A table should be created to contain the data collected from the trigger.

```
mysql> CREATE TABLE DeletedCity
    -> (ID INT UNSIGNED, Name VARCHAR(50), DeleteDate TIMESTAMP);
Query OK, 0 rows affected (#.## sec)
```



3) Create the actual trigger, using the appropriate name according to the type of event. Here it is “**AD**” for **AFTER DELETE**. The **FOR EACH ROW** in the syntax means that execution occurs once “for each row deleted,” not “for each row currently in the table.”

```
mysql> CREATE TRIGGER City_AD AFTER DELETE ON City
-> FOR EACH ROW
-> INSERT INTO DeletedCity (ID, Name) VALUES (OLD.ID, OLD.Name);
Query OK, 0 rows affected (#.# sec)
```

4) Confirm that the **City_AD** trigger now exists.

```
mysql> SHOW TRIGGERS\G
***** 1. row *****
Trigger: City_AD
Event: DELETE
Table: city
Statement: INSERT INTO DeletedCity (ID, Name) VALUES (OLD.ID, OLD.Name)
Timing: AFTER
Created: NULL
sql_mode: STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
Definer: root@localhost
1 row in set (#.# sec)
```

5) Perform a query to confirm the existence of **City** data that fits the trigger criteria, before deleting. This example uses the city of **Dallas**.

```
mysql> SELECT * FROM City WHERE Name = 'Dallas';
+----+-----+-----+-----+
| ID | Name | CountryCode | District | Population |
+----+-----+-----+-----+
| 3800 | Dallas | USA | Texas | 1188580 |
+----+-----+-----+-----+
1 row in set (#.# sec)
```

6) Perform a **DELETE** that will create results to be placed in the **DeletedCity** trigger table.

```
mysql> DELETE FROM City WHERE Name = 'Dallas';
Query OK, 1 row affected (#.# sec)
```

7) Perform another query to confirm the *deletion* of the same **City** data. You see that the city of **Dallas** has been removed from the **City** table.

```
mysql> SELECT * FROM City WHERE Name = 'Dallas';
Empty set (#.# sec)
```



8) Perform a query on the **City_AD** trigger table. Notice that **Dallas** has been added to the table.

```
mysql> SELECT * FROM DeletedCity;
+----+----+-----+
| ID | Name | DeleteDate |
+----+----+-----+
| 3800 | Dallas | 2009-01-22 11:28:45 |
+----+----+-----+
1 row in set (#.## sec)
```

Other commands that have a direct affect on the data in the tables, like **REPLACE** and **LOAD DATA INFILE**, can also trigger the associated events. In addition, triggers can only have one event type.

Triggers can be used with *compound* statements as well. The **DELIMITER** statement is needed in the case of a compound statement, as with stored procedures.

548

19.2.3 Trigger Error Handling

Errors during trigger execution are handled as follows:

- If a **BEFORE** trigger event fails, the operation on the corresponding row is not performed.
- An **AFTER** trigger event is executed only if the **BEFORE** trigger event (if any) and the row operation both execute successfully.
- For transactional tables, failure of a trigger (and thus the whole statement) should cause rollback of all changes performed by the statement. For non-transactional tables, such rollback cannot be done, so although the statement fails, any changes performed prior to the point of the error remain in effect.

As of version 5.1, there is a **TRIGGER** privilege, so the user does not need **SUPER** privilege in order to create triggers or use the **DEFINER** clause. (*Prior to 5.1 SUPER privilege is required for these features.*)



19.3 Delete Triggers

549

```
DROP TRIGGER trigger_name;
```

This statement drops a trigger. However, when a table is deleted, all the triggers associated with that table are also deleted.

When using `DROP TRIGGER trigger_name`, the server will look for that trigger name in the current schema. If the `trigger_name` that is being deleted is in another schema, the issuer of the statement can include the schema name:

```
DROP TRIGGER schema_name.trigger_name;
```

Note: If you drop a table, the triggers are automatically dropped also. And there is no `IF EXISTS` option available with `DROP TRIGGER`.

As of version 5.1, there is a **TRIGGER** privilege, so the user does *not* need **SUPER** privilege in order to drop triggers. (*Prior to 5.1 SUPER privilege is required for this feature.*)

19.4 Restrictions on Triggers

550

Triggers bring a level of power and security down to the individual record sets of data within a table. However, there are *limitations* to triggers. The following are not allowed:

- SQL prepared statements (`PREPARE`, `EXECUTE`, `DEALLOCATE PREPARE`).
- Statements that do explicit or implicit `COMMIT` or `ROLLBACK`.
- Statements that return a result set. This includes `SELECT` statements that do *not* have an `INTO var_list` clause and `SHOW` statements. A trigger *can* process a result set either with `SELECT ... INTO var_list` or by using a cursor and `FETCH` statements.
- `FLUSH` statements.
- Recursive statements. That is, triggers cannot be used recursively.





InLine Lab 19-A

In this exercise you will create tables, **create triggers** on the tables created, and **drop the triggers**.

Step 1. Create tables to supporting testing the use of triggers

1. Using the **world** database, create the following table:

```
mysql> CREATE TABLE dotriggers (column1 INT);
```

The table **dotriggers** is created with one (1) column, **column1**, which is assigned as an integer.

2. Create the following table also:

```
mysql> CREATE TABLE audit_triggers  
-> (old_column INT, new_column INT, date_completed DATETIME);
```

The table **audit_triggers** is created with three (3) columns: **old_column**, which is an integer column; **new_column**, which is also an integer column; **date_completed**, which is a column that can hold the date and time.

Step 2. Create a trigger against the dotriggers table

1. Create the following trigger:

```
mysql> CREATE TRIGGER dotriggers_ai AFTER INSERT ON dotriggers  
-> FOR EACH ROW  
-> INSERT INTO audit_triggers (new_column, date_completed)  
-> VALUES (NEW.column1, NOW());
```

This trigger, **dotriggers_ai**, will insert records into the **audit_triggers** table after a record is inserted into the **dotriggers** table. Even though the **dotriggers** table only has one (1) column, the **dotriggers_ai** will insert two (2) columns of data into the **audit_triggers** table: **new_column** which will store the new entry added to the **dotriggers** table and **date_completed** which will store the date and time that the record was inserted.

Step 3. Insert records into the dotriggers table

1. Insert values into the **dotriggers** table using the following statement:

```
mysql> INSERT INTO dotriggers VALUES (1), (2), (3), (4), (5), (6);
```

This statement enters six (6) rows of records into the **dotriggers**.



2. Verify the records were added using the following statement:

```
mysql> SELECT * FROM dotriggers;
```

This statement displays the six (6) records that were added into the **dotriggers** table.

3. Verify the **dotriggers_ai** trigger performed the required actions by entering the following statement:

```
mysql> SELECT * FROM audit_triggers;
```

This statement displays the six (6) records that were added into the **audit_triggers** table based on the **dotriggers_ai** trigger. This display not only shows the value that was inserted but also the date and time when it was inserted.

Note: The value in the **old_column** is **NULL** because there was no old data being modified.

Step 4. Create a trigger against the dotriggers table that handles updates

1. Create the following trigger:

```
mysql> CREATE TRIGGER dotriggers_au AFTER UPDATE ON dotriggers
      -> FOR EACH ROW
      -> INSERT INTO audit_triggers (old_column, new_column, date_completed)
      -> VALUES (OLD.column1, NEW.column1, NOW());
```

This trigger, **dotriggers_au**, will insert records into the **audit_triggers** table after a record is updated in the **dotriggers** table. **dotriggers_au** will update three (3) columns in the **audit_triggers** table: **old_column** which will store the old value of the data being updated; **new_column** which will store the new entry added to the **dotriggers** table; **date_completed** which will store the date and time that the record was updated.

Step 5. Update records in the dotriggers table

1. Update values in the **dotriggers** table using the following statement:

```
mysql> UPDATE dotriggers SET column1=10 WHERE column1 < 3;
```

This statement will update all records in the **dotriggers** table with a **column1** value less than 3 to the value of 10.

2. Verify the records were updated using the following statement:

```
mysql> SELECT * FROM dotriggers;
```

This statement displays all the records of the **dotriggers** table including the first two (2) records which now have a value of 10.



3. Verify the **dotriggers_au** trigger performed the required actions by entering the following statement:

```
mysql> SELECT * FROM audit_triggers;
```

This statement displays all the records that are in the **audit_triggers**. Two (2) additional records have been added identifying the update that took place in the **dotriggers** table.

Step 6. Create a trigger against the dotriggers table that handles deletes

1. Create the following trigger:

```
mysql> CREATE TRIGGER dotriggers_ad AFTER DELETE ON dotriggers
    -> FOR EACH ROW
    -> INSERT INTO audit_triggers(old_column, date_completed)
    -> VALUES (OLD.column1, NOW());
```

This trigger, **dotriggers_ad**, will insert records into the **audit_triggers** table after a record is deleted in the **dotriggers** table. **dotriggers_ad** will update two (2) columns in the **audit_triggers** table: **old_column** which will store the old value of the data being deleted and **date_completed** which will store the date and time that the record was deleted.

Step 7. Delete records in the dotriggers table

1. Delete records in the **dotriggers** table using the following statement:

```
mysql> DELETE FROM dotriggers WHERE column1 > 2 AND column1 < 10;
```

This statement will delete all records from the **dotriggers** table that have a value greater than 2 and less than 10.

2. Verify the records were deleted using the following statement:

```
mysql> SELECT * FROM dotriggers;
```

This statement displays only two (2) records in the **dotriggers** table. The other records were deleted.

3. Verify the **dotriggers_ad** trigger performed the required actions by entering the following statement:

```
mysql> SELECT * FROM audit_triggers;
```

This statement displays all the records that are in the **audit_triggers**. Four (4) additional records have been added identifying the delete that took place in the **dotriggers** table.



Step 8. Drop the delete trigger associated with the dotriggers table

1. Drop the **dotriggers_ad** trigger by entering the following statement:

```
mysql> DROP TRIGGER dotriggers_ad;
```

The **dotriggers_ad** trigger is deleted from the **world** database. Any future deletes on the **dotriggers** table will not be recorded in the **audit_triggers** table.

Step 9. Delete records in the dotriggers table

1. Delete all the records in the **dotriggers** table using the following statement:

```
mysql> DELETE FROM dotriggers;
```

This statement will delete all records from the **dotriggers** table.

2. Verify the records were deleted using the following statement:

```
mysql> SELECT * FROM dotriggers;
```

This statement returns an empty set, all records were deleted.

3. Check to see if the records that were deleted were entered in the **audit_triggers** table by entering the following statement:

```
mysql> SELECT * FROM audit_triggers;
```

The records that were deleted in step 9-1 are not recorded due to the trigger for deletions being dropped in step 8-1.





Further Practice

In this exercise, you will use the information covered in this section to implement **triggers** using the **mysql** client, with the **world** database.

551

1. Using **SHOW** display all the tables in the **world** database.
2. Create a trigger on the **Country** table so that upon **DELETE** of a country the **City** and **CountryLanguage** tables get changed accordingly.
3. Review the number of records in both the **City** and **CountryLanguage** that are associated with 'Philippines' (PHL).
4. Delete the 'Philippines' from the **country** table.
5. Review again the number of records in both the **city** and **CountryLanguage** that are associated with 'Philippines' (PHL).
6. Create a trigger on the **country** table so that upon **UPDATE** of a country code the **city** and **CountryLanguage** tables get changed accordingly.
7. Review the number of records in both the **City** and **CountryLanguage** that are associated with 'New Zealand' (NZL).
8. Change the country code for 'New Zealand' (NZL) to **NEW**.
9. Review again the number of records in both the **City** and **CountryLanguage** that are associated with 'New Zealand' using the old **CountryCode** 'NZL'.
10. Review again the number of records in both the **City** and **CountryLanguage** that are associated with 'New Zealand' but this time use the new **CountryCode** 'NEW'.
11. Create triggers on the **City** table to make sure that the population is an even 1000 (before *insert* and before *update*).



19.5 Chapter Summary

This chapter introduced **Triggers** in **MySQL**. In this chapter, you learned to:

552

- Describe triggers
- Create new triggers
- Delete existing triggers





20 STORAGE ENGINES

20.1 Learning Objectives

554

This chapter introduces **Storage Engines** in MySQL. In this chapter, you will learn to:

- Describe the effect of storage engine assignment on MySQL performance
- List the most common storage engines available
- Differentiate between the features of each storage engine
- Set each individual storage engine type



555

20.2 SQL Parser and Storage Engine Tiers

A client retrieves data from tables or changes data in tables by sending requests to the server in the form of SQL statements. The server executes each statement using a two-tier processing model:

- The upper tier includes the SQL parser and optimizer.
- The lower tier comprises a set of storage engines.

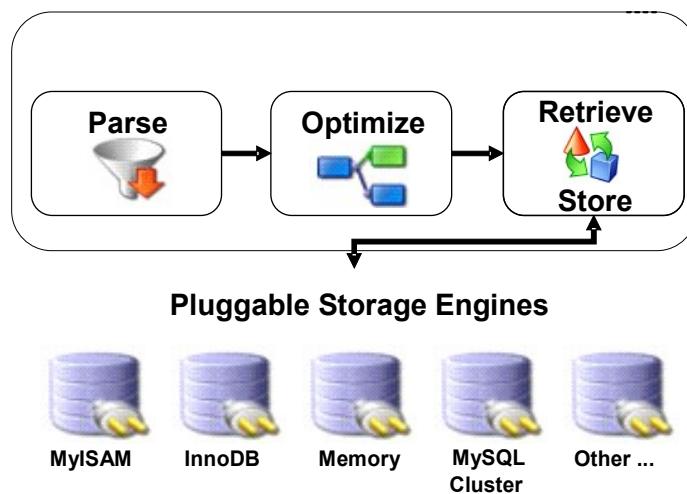
For the most part, the SQL tier is free of dependencies on which storage engine manages any given table. This means that clients normally need not be concerned about which engines are involved in processing SQL statements, and can access and manipulate tables using statements that are the same no matter which engine manages them. Exceptions to this engine-independence of SQL statements include the following:

- **CREATE TABLE** has an **ENGINE** option that enables you to specify which storage engine to use on a per-table basis. **ALTER TABLE** has an **ENGINE** option that enables you to convert a table to use a different storage engine.
- Some index types are available only for particular storage engines. For example, only the MyISAM engine supports full-text or spatial indexes.
- **COMMIT** and **ROLLBACK** have an effect only for tables managed by transactional storage engines such as InnoDB.

20.2.1 Storage Engine Breakdown

556

The following diagram represents a simplified view of the MySQL server and its interaction with the storage engines.



The following properties are storage engine *dependent*:

557

- **Storage Medium** – Each table uses its own method of storing the data it contains.
- **Transactional Capabilities** – Certain storage engines handle transactional processing which ensures that integrity of a database is maintained during the processing of multiple SQL statements.
- **Locking** – Each storage engines handles the processes of the synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution.
- **Backup and Recovery** – Based on the storage medium used, the backup of the table data and the recovery of that data can be distinct.
- **Optimization** – There are specific issues associated with each storage engine for the optimization of the storage of the data and retrieval of the data through the MySQL server.
- **Special Features** – There are a number of features that exist only in certain engine types to include full-text search, referential integrity and the ability to handle spatial data.

Most of the MySQL server operates in the same way no matter what storage engine is used: all the usual SQL commands are independent of the storage engine. Naturally, the optimizer may need to make different choices depending on the storage engine, but this is all handled through a standardized interface (API) which each storage engine supports.

20.3 Storage Engines and MySQL

558

When you create a table using MySQL, you can choose what storage engine to use. Typically, this choice is made according to which storage engine offers features that best fit the needs of your application. Each storage engine has a particular set of operational characteristics, such as in the way that they use locking to manage query contention, or in whether the tables that they provide are transactional or non-transactional. These engine properties have implications for query processing performance, concurrency, and deadlock prevention.

559

20.3.1 Available Storage Engines

MySQL provides and maintains several storage engines. It is also compatible with many third party storage engines. A MySQL storage engine is a low-level engine inside the database server that takes care of storing and retrieving data, and can be accessed through an internal MySQL API or, in some situations be accessed directly by an application. Note that one application can have more than one storage engine in use at any given time. Third party engines have different sources and features. The following are lists of the currently supported storage engines.

MySQL developed storage engines:

- MyISAM
- Falcon
- NDB/Cluster
- MEMORY
- ARCHIVE
- BLACKHOLE
- CSV



Third Party Storage Engines:

- InnoDB
- solidDB
- InfoBright- BrightHouse
- Nitro
- PBXT

560

20.3.2 The Most Common Storage Engines

MySQL works well with many storage engines offering a variety of choices according to specific needs. Of the many storage engines available for use with MySQL, the most commonly used are the following:

- MyISAM
- InnoDB
- MEMORY

Each of these storage engines is covered in detail in the following sections.

561

20.3.3 View Available Storage Engines

To see what storage engines are compiled into your server and whether they are available at runtime, use the **SHOW ENGINES** statement:

```
mysql> SHOW ENGINES\G
***** 1. row *****
Engine: MyISAM
Support: DEFAULT
Comment: Default engine as of MySQL 3.23 with great performance
...
***** 2. row *****
Engine: MEMORY
Support: YES
Comment: Hash based, stored in memory, useful for temporary tables
...
***** 3. row *****
Engine: InnoDB
Support: YES
Comment: Supports transactions, row-level locking, foreign keys
...
```

The value in the **Support** column is **YES** or **NO** to indicate that the engine is or is not available. **DISABLED** if the engine is present but turned off (to save resources), or **DEFAULT** for the storage engine that the server uses by default. The engine designated as **DEFAULT** should be considered available.



562

20.3.4 Setting the Storage Engine

To specify a storage engine explicitly in a `CREATE TABLE` statement, use an `ENGINE` option. The following statement creates `t` as an `InnoDB` table:

```
mysql> CREATE TABLE t (i INT) ENGINE = InnoDB;
```

If you create a table without using an `ENGINE` option to specify a storage engine explicitly, the MySQL server creates the table using the default engine, which is given by the value of the `storage_engine` system variable.

The storage engine of a table can be changed with `ALTER TABLE`:

```
mysql> ALTER TABLE t ENGINE = MEMORY;
```

563

20.3.5 Displaying Storage Engine Information

To determine which storage engine is used for a given table, you can use the `SHOW CREATE TABLE` or the `SHOW TABLE STATUS` statement:

```
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=MyISAM DEFAULT CHARSET=latin1
1 row in set (#.## sec)
```



564

```
mysql> SHOW TABLE STATUS LIKE 'CountryLanguage'\G
***** 1. row *****
      Name: CountryLanguage
      Engine: MyISAM
     Version: 10
   Row_format: Fixed
        Rows: 984
Avg_row_length: 39
  Data_length: 38376
Max_data_length: 167503724543
  Index_length: 22528
    Data_free: 0
Auto_increment: NULL
  Create_time: 2005-04-26 22:15:35
  Update_time: 2005-04-26 22:15:43
  Check_time: NULL
  Collation: latin1_swedish_ci
  Checksum: NULL
Create_options:
  Comment:
1 row in set (#.## sec)
```

565

You can also get information from the **INFORMATION_SCHEMA** database:

```
mysql> SELECT TABLE_NAME, ENGINE FROM INFORMATION_SCHEMA.TABLES
   -> WHERE TABLE_NAME = 'City' AND TABLE_SCHEMA = 'world'\G
***** 1. row *****
TABLE_NAME: city
  ENGINE: InnoDB
1 row in set (#.## sec)
```

Although you can choose which storage engine to use for a table, in most respects the way that you use the table after creating it is *engine independent*. Operations on tables of all types are performed using the SQL interface and MySQL manages engine-dependent details for you, at a lower level in its architecture. That is, your interface to tables is at the higher SQL tier, and table-management details are at the lower storage-engine tier. Nonetheless, there are times when knowing which storage engine manages a table can enable you to use the table more efficiently.

Before you can use a given storage engine, it must be compiled into the server and enabled. MySQL Server uses a modular architecture: Each storage engine is a software module that is compiled into the server. The use of this modular approach allows storage engines to be easily selected for inclusion in the server at configuration time.

As of MySQL version 5.1 , storage engines can be added at runtime, instead of compile-time.
--





InLine Lab 20-A

In this exercise you will use the various statements to display current storage engine information. This will require access to the mysql server and the **world** database.

Step 1. Use multiple means to acquire storage engine information from tables

1. Using the **world** database, execute the following statement to show the table creating statement for the **CountryLanguage** table and note the engine designation:

```
mysql> SHOW CREATE TABLE CountryLanguage\G
```

Shows the entire create statement for the **CountryLanguage** table, which includes the storage engine being used.

2. Execute the following statement to show the status for the **City** table and note the engine designation:

```
mysql> SHOW TABLE STATUS LIKE 'City'\G
```

Shows the list of attributes of the **City** table, including the storage engine;

```
***** 1. row *****
Name: city
Engine: InnoDB
...
```

3. Execute the following statement to acquire the storage engine setting from the **INFORMATION_SCHEMA** database for the **Country** table:

```
mysql> SELECT TABLE_NAME, ENGINE FROM INFORMATION_SCHEMA.TABLES
-> WHERE TABLE_NAME = 'Country' AND TABLE_SCHEMA = 'world'\G
```

Lists the table name and assigned storage engine;

```
***** 1. row *****
TABLE_NAME: country
ENGINE: MyISAM
1 row in set (#.## sec)
```

Step 2. Display a list of all storage engines installed

1. Execute the following statement to show the currently available storage engines:

```
mysql> SHOW ENGINES\G
```

Shows the list of engines and their support information.



566

20.4 The MyISAM Storage Engine

The **MyISAM storage engine** is the default engine in MySQL. MyISAM table management has the following characteristics (*and more*):



- Represents each table using three files:
 1. A *format* file--stores the definition of the table structure (`mytable.frm`)
 2. A *data* file--stores the contents of table rows (`mytable.MYD`)
 3. An *index* file that stores any indexes on the table (`mytable.MYI`)
- The most flexible **AUTO_INCREMENT** column handling of all the storage engines
- Can be converted into fast, compressed, read-only tables to save space
- Manages contention between queries for MyISAM table access using table-level locking
- Supports **FULLTEXT** searching and spatial data types
- Supports the **GIS** (Geographical Information System) geometric spatial extensions
- The table storage format is portable, so table files can be copied directly to another host and used by a server there
- Can improve performance by limiting table size to a certain number of rows
- When loading data into an empty table, you can disable updating of non-unique indexes and enable the indexes after loading
- Tables take up very little space

MyISAM tables use the Indexed Sequential Access Method for indexing, as did the older ISAM storage engine. MyISAM offers better performance and more features than ISAM, so MyISAM is preferred over ISAM, and ISAM is unavailable as of MySQL 5.0.

567

20.4.1 MyISAM Row Storage Formats

The MyISAM storage engine has the capability of storing rows in three formats: *fixed-row*, *dynamic-row*, and *compressed*. These formats have differing characteristics.

Fixed-row format:

- All rows have the same size.
- Rows are stored within the table at positions that are multiples of the row size, making them easy to look up.
- Fixed-size rows take more space (*unless the data is fixed length, then it actually takes up less space*).
- All columns must have fixed width data types.



Dynamic-row format:

- Rows take varying amounts of space.
- Rows cannot be looked up as efficiently.
- Dynamic-rows tables usually take less space because rows are not padded to a fixed size.
- Fragmentation can occur more easily than for fixed-row tables.
- Repair/corruption problems can occur

Compressed format:

- Tables are packed to save space, using the **myisampack** utility.
- Storage is optimized for quick retrieval.
- Tables are read-only.
- Reversible operation.

568

20.4.2 Compressing MyISAM Tables

MyISAM tables are fixed or dynamic by default according to your table designs. However, a table does not get compressed by default. You must deliberately compress a table using the **myisampack** utility.

Although compressed tables can be a good thing, they only make sense for some applications. You have to keep in mind that compressed tables are read-only. If you need to alter, update, or insert data in a table, you need to decompress the entire table, make your changes, and then recompress the table.

myisampack includes a mixture of true compressions (Huffman coding) and a set of optimizations aimed at shrinking columns, such as converting data types to smaller data types and converting columns to enums. Because each record is compressed separately, there is only a small cost when decompressing. This may even help on slow devices due to the reduction in data that needs to be read from disk.

It is also necessary to use **myisamchk** afterward to update the indexes. The following example demonstrates how to perform this procedure, using the tables in the `world` database. **Note:** A table must not be in use by other programs (including the server) while you compress or uncompress it. The easiest thing to do is to stop the server while using **myisampack** or **myisamchk**.

1. Back up the tables, just in case:

```
shell> mysqldump world > world.sql
```

2. Stop the server so that it will not use the tables while you are packing them.



3. Change location into the database directory where the **world** tables are stored, and then use **myisampack** to compress them:

```
shell> myisampack Country City CountryLanguage
Compressing Country.MYD: (239 records)
- Calculating statistics
- Compressing file
72.95%
Compressing City.MYD: (4079 records)
- Calculating statistics
- Compressing file
70.94%
Compressing CountryLanguage.MYD: (984 records)
- Calculating statistics
- Compressing file
71.42%
Remember to run myisamchk -rq on compressed tables
```

myisampack also understands index filenames as arguments:

```
shell> myisampack *.MYI
```

Using index filenames does not affect the way **myisampack** works. It simply gives you an easier way to name a group of tables, because you can use filename patterns.

4. After compressing a table, you should run **myisamchk** to rebuild the indexes (as the final line of **myisampack** output indicates). Like **myisampack**, **myisamchk** understands index filename arguments for naming tables, so you can rebuild the indexes as follows:

```
shell> myisamchk -rq *.MYI
```

The equivalent long-option command is:

```
shell> myisamchk --recover --quick *.MYI
```

5. Restart the server.

569

20.4.3 MyISAM Locking

MyISAM locking occurs at the table level. This is not as desirable as page or row locking for concurrency in a mixed read/write environment. However, deadlock can occur with any transaction.

When processing queries on MyISAM tables, the server manages contention for the tables by simultaneous clients by implicitly acquiring any locks it needs. You can also lock tables explicitly with the **LOCK TABLES** and **UNLOCK TABLES** statements.



MyISAM tables support concurrent inserts. If a MyISAM table has no holes in the middle (resulting from deleted or updated records), inserts always take place at the end of the table and can be performed while other clients are reading the table. Concurrent inserts can take place even for a table that has been read-locked explicitly if the locking client acquired a **READ LOCAL** lock rather than a regular **READ** lock.

If a table *does* have holes, concurrent inserts will not be performed by default. However, you can remove the holes by using **OPTIMIZE TABLE** to defragment the table. (Note that a record deleted from the end of the table does *not* create a hole and does *not* prevent concurrent inserts.) Or you can set the **concurrent_inserts** system variable to 2 (same as **--concurrent-inserts** `mysqld` option), then concurrent inserts are enforced, with all new rows being appended to the end of the table, even if there are deleted rows.

For applications that use MyISAM tables, you can change the priority of statements that retrieve or modify data. This can be useful in situations where the normal scheduling priorities do not reflect the application's requirements.

By default, the server schedules queries for execution as follows:

- Write requests (such as **UPDATE** and **DELETE** statements) take priority over read requests (such as **SELECT** statements).
- The server performs the write queries in the order that it receives them.

However, if a table is being read from when a write request arrives, the write request cannot be processed until all current readers have finished. Any read requests that arrive after the write request must wait until the write request finishes, even if they arrive before the current readers finish. That is, a new read request by default does not jump ahead of a pending write request.

570

When working with MyISAM tables, certain scheduling modifiers are available to change the priority of requests:

- The **LOW_PRIORITY** modifier may be applied to statements that update tables (**INSERT**, **DELETE**, **REPLACE**, or **UPDATE**). A low-priority write request waits not only until all current readers have finished, but for any pending read requests that arrive while the write request itself is waiting. That is, it waits until there are no pending read requests at all. It is therefore possible for a low-priority write request never to be performed, if read requests keep arriving while the write request is waiting.
- **HIGH_PRIORITY** may be used with a **SELECT** statement to move it ahead of updates and ahead of other **SELECT** statements that do not use the **HIGH_PRIORITY** modifier.
- **DELAYED** may be used with **INSERT** (and **REPLACE**). The server buffers the rows in memory and inserts them when the table is not being used. Delayed inserts increase efficiency because they're done in batches rather than individually. While inserting the rows, the server checks periodically to see whether other requests to use the table have arrived. If so, the server suspends insertion of delayed rows until the table becomes free again. Using **DELAYED** allows the client to proceed immediately after issuing the **INSERT** statement rather than waiting until it completes.

Consider an application consisting of a logging process that uses **INSERT** statements to record information in a log table, and a summary process that periodically issues **SELECT** queries to generate reports from the log table. Normally, the server will give table updates priority over retrievals, so at times of heavy logging activity, report generation might be delayed. If the application places high importance on having the summary process execute as quickly as possible, it can use scheduling modifiers to alter the usual query priorities. Two approaches are possible:



- To elevate the priority of the summary queries, use **SELECT HIGH_PRIORITY** rather than **SELECT** with no modifier. This will move the **SELECT** ahead of pending **INSERT** statements that have not yet begun to execute.
- To reduce the priority of record logging statements, use **INSERT** with either the **LOW_PRIORITY** or **DELAYED** modifier.

If you use **DELAYED**, keep the following points in mind:

- Delayed rows tend to be held for a longer time on a very busy server than on a lightly loaded one.
- If a crash occurs while the server is buffering delayed rows in memory, those rows are lost.

The implication is that **DELAYED** is more suitable for applications where loss of a few rows is not a problem, rather than applications for which each row is critical. For example, **DELAYED** can be appropriate for an application that logs activity for informational purposes only and for which it is not important if a small number of rows is lost.





InLine Lab 20-B

In this exercise you will change the current setting of a table storage engine to the **MyISAM engine**. This will require access to the mysql server and the **world** database.

Step 1. View the current storage engine

1. Using the **world** database, show the create table statement for the **City** table using the following statement and note the engine designation:

```
mysql> SHOW CREATE TABLE City;
```

Shows the entire create statement for the **City** table, which includes the storage engine;

```
...  
) ENGINE=InnoDB ...
```

Step 2. Alter the storage engine being used

1. Change the current setting of the **City** table storage engine to be MyISAM using the following statement:

```
mysql> ALTER TABLE City ENGINE=MyISAM;
```

Returns an OK, showing that all rows of the table were affected.

2. Confirm the change to the **City** table engine designation using the following statement:

```
mysql> SHOW CREATE TABLE City;
```

Shows the entire create statement for the **City** table, which includes the storage engine;

```
...  
) ENGINE=MyISAM ...
```



571

20.5 The InnoDB Storage Engine

The **InnoDB** storage engine table management has the following characteristics (*and more*):

- Each **InnoDB** table is represented on disk by an **.frm** format file in the database directory, as well as data and index storage in the **InnoDB tablespace**:
 - The tablespace is a set of files (1 or more) that InnoDB uses to store data and indexes
 - By default, it uses a single tablespace that is shared by all tables
 - Table sizes can exceed the maximum file size allowed by the filesystem
 - Can configure **InnoDB** to create each table with its own tablespace
- Supports transactions, with **COMMIT** and **ROLLBACK**
- Provides full **ACID** compliance
- Provides auto-recovery after a crash of the MySQL server
- Row level locking with MVCC (Multi-Versioning Concurrency Control) and non-locking reads
- Supports foreign keys and referential integrity, including cascaded deletes and updates
- Supports consistent and online logical backup



572

20.5.1 The InnoDB Tablespace and Logs

InnoDB operates using two primary disk-based resources: a tablespace for storing table contents, and a set of log files for recording transaction activity.

Each InnoDB table has a format (**.frm**) file in the database directory of the database to which the table belongs. This is the same as tables managed by any other MySQL storage engine, such as MyISAM. However, InnoDB manages table contents (data rows and indexes) on disk differently than does the MyISAM engine. By default, InnoDB uses a shared “tablespace,” which is one or more files that form a single logical storage area. All InnoDB tables are stored together within the tablespace. There are no table-specific data files or index files for InnoDB the way there are for MyISAM tables. The tablespace also contains a rollback segment. As transactions modify rows, undo log information is stored in the rollback segment. This information is used to roll back failed transactions.

Although InnoDB treats the shared tablespace as a single logical storage area, it can consist of one file or multiple files. Each file can be a regular file or a raw partition. The final file in the shared tablespace can be configured to be auto-extending, in which case InnoDB expands it automatically if the tablespace fills up. Because the shared tablespace is used for InnoDB tables in all databases (and thus is not database specific), tablespace files are stored by default in the server's data directory, not within a particular database directory.

If you do not want to use the shared tablespace for storing table contents, you can start the server with the **--innodb-file-per-table** option. In this case, for each new table that InnoDB creates, it sets up an **.ibd** file in the database directory to accompany the table's **.frm** file. The **.ibd** file acts as the table's own tablespace file and InnoDB stores table contents in it. (The shared tablespace still is needed because it contains the InnoDB data dictionary and the rollback segment.)

Use of the **--innodb-file-per-table** option does not affect accessibility of any InnoDB tables that may already have been created in the shared tablespace. Those tables remain accessible.



573

In addition to its tablespace files, the InnoDB storage engine manages a set of InnoDB-specific log files that contain information about ongoing transactions. As a client performs a transaction, the changes that it makes are held in the InnoDB log. The more recent log contents are cached in memory. Normally, the cached log information is written and flushed to log files on disk at transaction commit time, though that may also occur earlier.

If a crash occurs while the tables are being modified, the log files are used for auto-recovery: When the MySQL server restarts, it reapplys the changes recorded in the logs, to ensure that the tables reflect all committed transactions.

574

20.5.2 InnoDB and ACID Compliance

InnoDB satisfies the conditions for ACID compliance, assuming that its log flushing behavior is set appropriately. InnoDB can be configured for log flushing that provides ACID compliance, or for flushing that gains some in performance at the risk of losing the last few transactions if a crash occurs. By default, InnoDB log flushing is set for ACID compliance.

20.5.3 InnoDB Locking

InnoDB has the following general locking properties:

- InnoDB does not need to set locks to achieve consistent reads because it uses multi-versioning to make them unnecessary: Transactions that modify rows see their own versions of those rows, and the undo logs allow other transactions to see the original rows. Locking reads may be performed by adding locking modifiers to `SELECT` statements.
- When locks are necessary, InnoDB uses row-level locking. In conjunction with multi-versioning, this results in good query concurrency because a given table can be read and modified by different clients at the same time. Row-level concurrency properties are as follows:
 - Different clients can read the same rows simultaneously.
 - Different clients can modify different rows simultaneously.
 - Different clients cannot modify the same row at the same time. If one transaction modifies a row, other transactions cannot modify the same row until the first transaction completes. Other transactions cannot read the modified row, either, unless they are using the `READ UNCOMMITTED` isolation level. That is, they will see the original unmodified row.
- During the course of a transaction, InnoDB may acquire row locks as it discovers them to be necessary. However, it never escalates a lock (for example, by converting it to a page lock or table lock). This keeps lock contention to a minimum and improves concurrency. Although it does use table-level locking for some operations.
- Deadlock is possible because InnoDB does not acquire locks during a transaction until they are needed. The deadlocked transactions eventually begin to time out and InnoDB rolls them back as they do.
- Deadlock detection occurs immediately.



InnoDB supports two locking modifiers that may be added to the end of **SELECT** statements. They acquire shared or exclusive locks and convert non-locking reads into locking reads:

- With **LOCK IN SHARE MODE**, InnoDB locks each selected row with a shared lock. It is a shared lock, this means that no other transactions can take exclusive locks but other transactions can also use shared locks. As normal reads do not lock anything they aren't affected by the locks.
- If the **SELECT** will select rows that have been modified in an uncommitted transaction, **LOCK IN SHARE MODE** will cause the **SELECT** to lock until that transaction commits.
- With **FOR UPDATE**, InnoDB locks each selected row with an exclusive lock, preventing others from acquiring any lock on the rows, but allows reading the rows.

In the **REPEATABLE READ** isolation level, you can add **LOCK IN SHARE MODE** to **SELECT** operations to force other transactions to wait for your transaction if they want to modify the selected rows. This is similar to operating at the **SERIALIZABLE** isolation level, for which InnoDB implicitly adds **LOCK IN SHARE MODE** to **SELECT** statements that have no explicit locking modifier.

575

20.6 The MEMORY Storage Engine

The **MEMORY** storage engine uses tables that are stored in memory and that have fixed-length rows. **MEMORY** storage engine tables are temporary. The **MEMORY** storage engine table management has the following characteristics:

- Each table is represented on disk by an **.frm** format file in the database directory. Table data and indexes are stored in memory
- In-memory storage results in very fast performance
- Contents do not survive a restart of the server (structure survives, but table contains zero rows)
- Limited by **max_heap_table_size** so they do not get too large
- MySQL manages query contention using table-level locking. Deadlock cannot occur.
- Cannot contain **TEXT** or **BLOB** columns
- Can use different character sets for different columns

The **MEMORY** stored engine formerly was called the **HEAP** engine. You might still see **HEAP** in older SQL code, and MySQL Server still recognizes **HEAP** for backward compatibility.

20.6.1 MEMORY Indexing Options

The **MEMORY** storage engine supports two indexing algorithms, **HASH** and **BTREE**:

- MEMORY** tables use hash indexes by default. They use hash indexes by default, which makes them very fast, and very useful for creating temporary tables. However, when the server shuts down, all rows stored in **MEMORY** tables are lost. However, hash indexes are usable only for comparisons that use the **=** or **<= >** operator.
- The **BTREE** index algorithm is preferable if the indexed column will be used with comparison operators other than **=** or **<= >**. For example, **BTREE** can be used for range searches such as **id < 100** or **id BETWEEN 200 AND 300**.



576

20.6.2 Storage Engine Summary

	MyISAM	MEMORY	InnoDB
Usage	Fastest for read heavy apps	In-Memory storage	Fully ACID compliant transactions
Locking	Large-grain table locks, no non-locking reads	Large grain table locks	Multi-versioning, Row-level locking
Durability	Table recovery	No disk I/O or persistence	Durability recovery
Supports Transactions	NO	NO	YES

577

20.7 Other Storage Engines

Some storage engines are always available, such as MyISAM, InnoDB, and MEMORY. Other engines are optional. Support for optional engines typically can be selected when MySQL is configured and built. Compiled-in optional engines also typically can be enabled or disabled with a server startup option. There are about a dozen storage engines currently supported by MySQL, including the following;

- **Falcon** – designed to work within high-traffic transactional applications for systems that are able to support larger memory architectures and multi-threaded or multi-core CPU environments
- **NDB** – MySQL's cluster storage engine
- **EXAMPLE** – used only for development and testing
- **ARCHIVE** – archival storage for large number of records that will never be altered
- **CSV** – stores data in comma-separated values format, as plain text
- **BLACKHOLE** -- data stored in a table disappears because the engine discards it

To reduce memory use, do not configure unneeded storage engines into the server. You can start the mysql client with the `--skip-engine` option. For example, `--skip-InnoDB` disables the **InnoDB** engine.





Quiz

In this exercise you will answer questions pertaining to **Storage Engines**.

1. If you create a table without using an _____ option to specify a storage engine explicitly, the MySQL server creates the table using the default server?

2. List the three most commonly used storage engines.

3. How can the MyISAM storage engine be disabled?

- a. Only when compiling MySQL from source, by using the --without-myisam option.
- b. By starting the server with the --skip-myisam option.
- c. By issuing the statement SET GLOBAL have_myisam = 0. You need the SUPER privilege to do this.
- d. The MyISAM storage engine cannot be disabled.

4. With table-level locking, what's the default behavior of the server?

- a. Read requests take priority over write requests.
- b. Write requests take priority over read requests.

5. Which statements are true for InnoDB tables?

- a. The tablespace consist of one or more files that are either regular files or raw partitions.
- b. If InnoDB is configured to use one tablespace per table, the number of tablespace files is equal to the number of InnoDB tables.

6. What kind of locking methods does MyISAM use?

- a. Table-level locking.
- b. Page-level locking.
- c. Row-level locking with lock escalation.
- d. Row-level locking without lock escalation.

7. In InnoDB, what happens if a deadlock occurs?



8. How can you set the transaction level to `SERIALIZE` at server startup?

9. The `SELECT ... FOR UPDATE` SQL statement makes sense for a transactional storage engine like InnoDB, but not for a non-transactional storage engine like MyISAM. True or False ?
10. According to the following session listing, it appears that the `SELECT` query took longer than two minutes to execute. What is the most probable reason for that?

```
mysql> INSERT DELAYED INTO City (ID, name) VALUES (20000, 'Delayne');  
Query OK, 1 row affected (#.## sec)  
  
mysql> SELECT ID, Name FROM City WHERE ID = 20000;  
+----+-----+  
| ID | Name |  
+----+-----+  
| 20000 | Delayne |  
+----+-----+  
1 row in set (2 min 5.61 sec)
```





Further Practice

In this exercise, you will use the information covered in this chapter to set and use various **storage engines** using the **world** database.

578

1. Create a MEMORY table containing the Code, Name and Continent columns of all North American countries from Country table in the world database.
2. Create another MEMORY table containing the same columns of all South American countries.
3. Verify that your tables are created with the correct engines.
4. Check your data directory to see what files the tables contain.
5. Change the storage engine of your new tables to MyISAM and look at the data directory again.



579

20.8 Chapter Summary

This chapter introduced **Storage Engines** in MySQL. In this chapter, you learned to:

- Describe the effect of storage engine assignment on MySQL performance
- List the most common storage engines available
- Differentiate between the features of each storage engine
- Set each individual storage engine type



21 OPTIMIZATION

21.1 Learning Objectives

581

This chapter introduces **Optimization** in MySQL. In this chapter, you will learn to:

- Describe the strategies available for optimizing queries
- Use the **EXPLAIN** statement to predict query performance
- Choose the most optimal storage engine for your query types
- Use Indexes for optimization



21.2 Overview of Optimization Principles

582

Why be concerned about optimization? The most obvious reason is to reduce query execution time. Another is that optimizing your queries helps everybody who uses the server, not just you. When the server runs more smoothly and processes more queries with less work, it performs better as a whole:

- A query that takes less time to run doesn't hold locks as long, so other clients that are trying to update a table do not have to wait as long. This reduces the chance of a query backlog building up.
- A query might be slow due to lack of proper indexing. If MySQL cannot find a suitable index to use, it must scan a table in its entirety. For a large table, that involves a lot of processing and disk activity. This extra overhead affects not only your own query, but it takes machine resources that could be devoted to processing other queries. Adding effective indexes allows MySQL to read only the relevant parts of the table, which is quicker and less disk intensive.

There are several optimization strategies that you can take advantage of to make your queries run faster:

- The primary optimization technique for reducing lookup times is to use indexing properly. This is true for retrievals (**SELECT** statements), and indexing also reduces row lookup time for **UPDATE** and **DELETE** statements as well. You should know general principles for creating useful indexes and for avoiding unnecessary ones.
- The way a query is written might prevent indexes from being used even if they are available. Rewriting the query often will allow the optimizer to use an index and process a query faster.
- The **EXPLAIN** statement provides information about how the MySQL optimizer processes queries. This is of value when you're trying to determine how to make a query run better (for example, if you suspect indexes are not being used as you think they should be).
- In some cases, query processing for a task can be improved by using a different approach to the problem. This includes techniques such as generating summary tables rather than selecting from the raw data repeatedly.
- Queries run more efficiently when you choose a storage engine with properties that best match application requirements.



21.3 Using Indexes for Optimization

583

Tables in MySQL can grow very large, but as a table gets bigger, retrievals from it become slower. To keep your queries performing well, it is essential to index your tables. When you create a table, consider whether it should have indexes, because they have important benefits:

- *Indexes contain sorted values.* This allows MySQL to find rows containing particular values faster. The effect can be particularly dramatic for joins, which have the potential to require many combinations of rows to be examined.
- *Indexes result in less disk I/O.* The server can use an index to go directly to the relevant table records, which reduces the number of records it needs to read. Furthermore, if a query displays information only from indexed columns, MySQL might be able to process it by reading only the indexes and without accessing data rows at all.
- *Indexes enforce uniqueness constraints* to ensure that duplicate values do not occur and that each row in a table can be distinguished from every other row. The terms *index* and *constraint* are used interchangeably from an SQL standpoint.

The downside of indexing is that it takes additional space, which can hurt performance of some data manipulation actions.

584

21.3.1 Types of Indexes

MySQL supports four general types of indexes:

- A **primary key** is an index for which each index value differs from every other and uniquely identifies a single row in the table. A primary key cannot contain **NULL** values.
- A **foreign key** is an index that references the primary key of another table.
- A **unique index** is similar to a primary key, except that it can be allowed to contain **NULL** values. Each non-**NULL** value uniquely identifies a single row in the table.
- A **non-unique** index is an index in which any key value may occur multiple times.

There are also more specialized types of indexes:

- A **FULLTEXT** index is specially designed for text searching.
- A **SPATIAL** index applies only to columns that have spatial data types.
- Not available in all storage engines.



585

Example:

```
mysql> CREATE TABLE `CountryLanguage` (
    -> `CountryCode` char(3) NOT NULL default '',
    -> `Language` char(30) NOT NULL default '',
    -> `IsOfficial` enum('T','F') NOT NULL default 'F',
    -> `Percentage` float(4,1) NOT NULL default '0.0',
    -> PRIMARY KEY (`CountryCode`, `Language`)
    -> ) ENGINE=MyISAM COMMENT='List Languages Spoken'
```

Storage engine-specific indexing:

586

21.3.2 Creating Indexes

You can create indexes at the same time that you create a table by including index definitions in the **CREATE TABLE** statement along with the column definitions. It is also possible to add indexes to an existing table with **ALTER TABLE** or **CREATE INDEX**.

Defining indexes at table creation time

To define indexes for a table at the time you create it, include the index definitions in the **CREATE TABLE** statement along with the column definitions. An index definition consists of the appropriate index-type keyword or keywords, followed by a list in parentheses that names the column or columns to be indexed. Suppose that the definition of a table **HeadOfState** without any indexes looks like this:

```
mysql> CREATE TABLE HeadOfState
    -> (ID INT NOT NULL,
    -> LastName CHAR(30) NOT NULL,
    -> FirstName CHAR(30) NOT NULL,
    -> CountryCode CHAR(3) NOT NULL,
    -> Inauguration DATE NOT NULL
    -> );
```

587

To create the table with the same columns but with a non-unique index on the date-valued column **Inauguration**, include an **INDEX** clause in the **CREATE TABLE** statement as follows:

```
mysql> CREATE TABLE HeadOfState
    -> (ID INT NOT NULL,
    -> LastName CHAR(30) NOT NULL,
    -> FirstName CHAR(30) NOT NULL,
    -> CountryCode CHAR(3) NOT NULL,
    -> Inauguration DATE NOT NULL
    -> INDEX (Inauguration)
    -> );
```

The keyword **KEY** may be used instead of **INDEX**.



To include multiple columns in an index (that is, to create a composite index), list all the column names within the parentheses, separated by commas. For example, a composite index that includes both the `LastName` and `FirstName` columns can be defined as follows:

```
mysql> CREATE TABLE HeadOfState
-> (ID INT NOT NULL,
-> LastName CHAR(30) NOT NULL,
-> FirstName CHAR(30) NOT NULL,
-> CountryCode CHAR(3) NOT NULL,
-> Inauguration DATE NOT NULL
-> INDEX (LastName, FirstName)
-> );
```

588

Composite indexes can be created for any type of index. The preceding indexing examples each include just one index in the table definition, but a table can have multiple indexes. The following table definition includes two indexes:

```
mysql> CREATE TABLE HeadOfState
-> (ID INT NOT NULL,
-> LastName CHAR(30) NOT NULL,
-> FirstName CHAR(30) NOT NULL,
-> CountryCode CHAR(3) NOT NULL,
-> Inauguration DATE NOT NULL
-> INDEX (LastName, FirstName)
-> INDEX (Inauguration)
-> );
```

To create a unique-valued index, use the `UNIQUE` keyword instead of `INDEX`. For example, if you want to prevent duplicate values in the `ID` column, create a `UNIQUE` index for it like this:

```
mysql> CREATE TABLE HeadOfState
-> (ID INT NOT NULL,
-> LastName CHAR(30) NOT NULL,
-> FirstName CHAR(30) NOT NULL,
-> CountryCode CHAR(3) NOT NULL,
-> Inauguration DATE NOT NULL
-> UNIQUE INDEX (ID)
-> );
```

There's one exception to the uniqueness of values in a `UNIQUE` index: If a column in the index may contain `NULL` values, multiple `NULL` values are allowed. This differs from the behavior for non-`NULL` values.



589

A **PRIMARY KEY** is similar to a **UNIQUE** index. The differences between the two are as follows:

- A **PRIMARY KEY** cannot contain **NULL** values; a **UNIQUE** index can. If a unique-valued index must be allowed to contain **NULL** values, you must use a **UNIQUE** index, not a **PRIMARY KEY**.
- Each table may have only one index defined as a **PRIMARY KEY**. (The internal name for a **PRIMARY KEY** is always **PRIMARY**, and there can be only one index with a given name.) It's possible to have multiple **UNIQUE** indexes for a table.

It follows from the preceding description that a **PRIMARY KEY** is a type of unique-valued index, but a **UNIQUE** index isn't necessarily a primary key unless it disallows **NULL** values. If it does, a **UNIQUE** index that cannot contain **NULL** is functionally equivalent to a **PRIMARY KEY**.

To index a column as a **PRIMARY KEY**, use the keywords **PRIMARY KEY** rather than **UNIQUE** and declare the column **NOT NULL** to make sure that it cannot contain **NULL** values.

590

Naming Indexes

For all index types other than **PRIMARY KEY**, you can name an index by including the name just before the column list. For example, the following definition uses names of **NameIndex** and **IDIndex** for the two indexes in the table:

```
mysql> CREATE TABLE HeadOfState
    -> (ID INT NOT NULL,
    -> LastName CHAR(30) NOT NULL,
    -> FirstName CHAR(30) NOT NULL,
    -> CountryCode CHAR(3) NOT NULL,
    -> Inauguration DATE NOT NULL
    -> INDEX NameIndex (LastName, FirstName)
    -> UNIQUE INDEX IDIndex (ID)
    -> );
```

- If you do not provide a name for an index, MySQL assigns a name for you based on the name of the first column in the index.
- For a **PRIMARY KEY**, you provide no name because the name is always **PRIMARY**.





InLine Lab 21-A

In this exercise you will use **CREATE TABLE** statement to create a table with indexes. This will require a MySQL command line client and access to the mysql server.

Step 1. Create a table with indexes

1. Change the database to the **test** database:

```
mysql> USE test;
```

Returns the message Database changed confirming the use of the **test** database.

2. Create a new table called **Foo2** with two columns; **col1** – data type **INT** (width of 6), **col2** – data type **VARCHAR** (width of 50) and index both the above columns:

```
mysql> CREATE TABLE Foo2
    -> (col1 int(6),
    ->     col2 varchar(50),
    ->     INDEX (col1, col2)
    -> );
```

Returns OK.

Step 2. Review the table that was created

1. Show the create table statement for the Foo2 table:

```
mysql> SHOW CREATE TABLE Foo2\G
```

Confirm the proper column attributes and indexes (shown as keys);

```
***** 1. row *****
Table: Foo2
Create Table: CREATE TABLE `foo2` (
  `col1` int(6) NOT NULL DEFAULT '0',
  `col2` varchar(50) DEFAULT NULL,
  KEY `col1` (`col1`,`col2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```



591

21.3.3 Adding Indexes to Existing Tables

To add an index to a table, you can use **ALTER TABLE** or **CREATE INDEX**. Of these statements, **ALTER TABLE** is the most flexible, as will become clear in the following discussion.

To add an index to a table with **ALTER TABLE**, use **ADD** followed by the appropriate index-type keywords and a parenthesized list naming the columns to be indexed. For example, assume that the **HeadOfState** table used earlier in this chapter is defined without indexes as follows:

```
mysql> CREATE TABLE HeadOfState(
    ->   ID INT NOT NULL,
    ->   LastName CHAR(30) NOT NULL,
    ->   FirstName CHAR(30) NOT NULL,
    ->   CountryCode CHAR(3) NOT NULL,
    ->   Inauguration DATE NOT NULL
    -> );
```

Using **ALTER TABLE** to Add Indexes

To create a **PRIMARY KEY** on the **ID** column and a composite index on the **LastName** and **FirstName** columns, you could issue these statements:

```
mysql> ALTER TABLE HeadOfState ADD PRIMARY KEY (ID);
mysql> ALTER TABLE HeadOfState ADD INDEX (LastName,FirstName);
```

However, MySQL allows multiple actions to be performed with a single **ALTER TABLE** statement. One common use for multiple actions is to add several indexes to a table at the same time, which is more efficient than adding each one separately. Thus, the preceding two **ALTER TABLE** statements can be combined as follows:

```
mysql> ALTER TABLE HeadOfState ADD PRIMARY KEY (ID),
    ->   ADD INDEX (LastName,FirstName);
```



592

Using CREATE INDEX to Add Indexes

The syntax for **CREATE INDEX** is as follows, where the statements shown create a single-column **UNIQUE** index and a multiple-column non-unique index, respectively:

```
mysql> CREATE UNIQUE INDEX IDIndex ON HeadOfState (ID);
mysql> CREATE INDEX NameIndex ON HeadOfState (LastName, FirstName);
```

Note that with **CREATE INDEX**, it's necessary to provide a name for the index. With **ALTER TABLE**, MySQL creates an index name automatically if you do not provide one.

Unlike **ALTER TABLE**, the **CREATE INDEX** statement can create only a single index per statement. In addition, only **ALTER TABLE** supports the use of **PRIMARY KEY**. For these reasons, **ALTER TABLE** is more flexible.

Building multiple indexes with **ALTER TABLE** requires a single pass through the table data, thus is faster than multiple **CREATE INDEX** statements.

Using Index Prefixes

For **CHAR**, **VARCHAR**, **BINARY**, and **VARBINARY** columns, indexes can be created that use only the leading part of column values by specifying an index prefix length. **BLOB** and **TEXT** columns also can be indexed, but a prefix length *must* be given. Prefix lengths are given in characters for non-binary string types and in bytes for binary string types. That is, index entries consist of the first *length* characters of each column value for **CHAR**, **VARCHAR** and **TEXT** columns, and the first *length* bytes of each column value for **BINARY**, **VARBINARY**, and **BLOB** columns. Index prefixes can be up to 1000 bytes long (767 bytes for **InnoDB** tables).

This example creates an index using the first 10 characters of the **name** column:

```
mysql> CREATE INDEX part_of_name ON customer (name(10));
```

If the names in the column normally differ in the first 10 characters, this index should not be much slower than an index created from the entire **name** column. Also, using partial columns for indexes can make the index file much smaller, which could save a lot of disk space and might also speed up **INSERT** operations.





InLine Lab 21-B

In this exercise you will use **ALTER TABLE** statement to alter the indexes in a pre-existing table. This will require a MySQL command line client and access to the mysql server.

Step 1. Show the table indexes for the Foo2 table

1. Change the database to the **test** database:

```
mysql> USE test;
```

Returns the message Database changed confirming the use of the **test** database.

2. Show the create table statement for the **Foo2** table using the follow statement:

```
mysql> SHOW CREATE TABLE Foo2\G
```

Note the column attributes and indexes (shown as keys).

Step 2. Add an additional index to the Foo2 table

1. Add a primary key to the **Foo2** table as the **col1** entity:

```
mysql> ALTER TABLE Foo2 ADD PRIMARY KEY (Col1);
```

Returns OK.

2. Show the create table statement for the **Foo2** table using the follow statement:

```
mysql> SHOW CREATE TABLE Foo2\G
```

Note that the primary key has been added;

```
***** 1. row *****
Table: Foo2
Create Table: CREATE TABLE `foo2` (
  `col1` int(6) NOT NULL DEFAULT '0',
  `col2` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`col1`),
  KEY `col1` (`col1`,`col2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```



593

21.3.4 Dropping Indexes

To drop an index from a table, use **ALTER TABLE** or **DROP INDEX**.

With **ALTER TABLE**, use a **DROP** clause and name the index to be dropped. Dropping a **PRIMARY KEY** is easy:

```
mysql> ALTER TABLE HeadOfState DROP PRIMARY KEY;
```

To drop another kind of index, you must specify its name. If you do not know the name, you can use **SHOW CREATE TABLE** to see the table's structure, including any index definitions, as shown here:

```
mysql> SHOW CREATE TABLE HeadOfState\G
***** 1. row *****
      Table: HeadOfState
Create Table: CREATE TABLE `HeadOfState` (
  `ID` int(11) NOT NULL default '0',
  `LastName` char(30) NOT NULL default '',
  `FirstName` char(30) NOT NULL default '',
  `CountryCode` char(3) NOT NULL default '',
  `Inauguration` date NOT NULL default '0000-00-00',
  KEY `NameIndex` (`LastName`, `FirstName`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

The **KEY** clause of the output shows that the index name is **NameIndex**, so you can drop the index using the following statement:

```
mysql> ALTER TABLE HeadOfState DROP INDEX NameIndex;
```

After you've dropped an index, you can add it back using the **ADD INDEX** clause:

```
mysql> ALTER TABLE HeadOfState ADD INDEX NameIndex (LastName, FirstName);
```

Dropping an index differs from dropping a database or a table, which cannot be undone except by recourse to backups. The distinction is that when you drop a database or a table, you're removing data. When you drop an index, you aren't removing table data, you're removing only a structure that's derived from the data. The act of removing an index is a reversible operation as long as the columns from which the index was constructed have not been removed. However, for a large table, dropping and recreating an index may be a time-consuming operation.



To drop an index with **DROP INDEX**, indicate the index name and table name:

```
mysql> DROP INDEX NameIndex ON t;
```

To drop a primary key with **DROP INDEX**, refer to the index name (PRIMARY), but use a quoted identifier because this name is a reserved word:

```
mysql> DROP INDEX `PRIMARY` ON t;
```

Unlike **ALTER TABLE**, the **DROP INDEX** statement can drop only on a single index per statement.





InLine Lab 21-C

In this exercise you will use **DROP INDEX** statement to remove an index in a pre-existing table. This will require a MySQL command line client and access to the mysql server.

Step 1. Show the table indexes for the Foo2 table

1. Change the database to the **test** database:

```
mysql> USE test;
```

Returns the message Database changed confirming the use of the **test** database.

2. Show the create table statement for the **Foo2** table using the follow statement:

```
mysql> SHOW CREATE TABLE Foo2\G
```

Note the column attributes and indexes (shown as keys).

Step 2. Remove an index from the Foo2 table

1. Remove the primary key from the **Foo2** table:

```
mysql> DROP INDEX `PRIMARY` ON Foo2;
```

Returns OK.

2. Show the create table statement for the **Foo2** table using the follow statement:

```
mysql> SHOW CREATE TABLE Foo2\G
```

Note that the primary key has been removed.



594

21.3.5 Principles for Index Creation

An index helps MySQL perform retrievals more quickly than if no index is used, but indexes can be used with varying degrees of success. Keep the following index-related considerations in mind when designing tables:

- Declare an indexed column **NOT NULL** if possible. Although **NULL** values can be indexed, **NULL** is a special value that requires additional decisions by the server when performing comparisons on key values. An index without **NULL** can be processed more simply and thus faster.
- Avoid over-indexing; do not index a column just because you can. If you never refer to a column in comparisons (such as in **WHERE**, **ORDER BY**, or **GROUP BY** clauses), there's no need to index it.
- Another reason to avoid unnecessary indexing is that every index you create slows down table updates. If you insert a row, an entry must be added to each of the table's indexes. Indexes help when looking up values for **UPDATE** or **DELETE** statements, but any change to indexed columns requires the appropriate indexes to be updated as well.
- One strategy the MySQL optimizer uses is that if it estimates that an index will return a large percentage of the records in the table, it will be just as fast to scan the table as to incur the overhead required to process the index. As a consequence, an index on a column that has very few distinct values is *unlikely* to do much good. Suppose that a column is declared as **ENUM('Y', 'N')** and the values are roughly evenly distributed such that a search for either value returns about half of the records. In this case, an index on the column is unlikely to result in faster queries.
- Choose unique and non-unique indexes appropriately. The choice might be influenced by the data type of a column. If the column is declared as an **ENUM**, the number of distinct column values that can be stored in it is fixed. This number is equal to the number of enumeration elements, plus one for the '' (empty string) element that is used when you attempt to store an illegal value.
- Index a column prefix rather than the entire column. MySQL caches prefix index information in memory whenever possible to avoid reading it from disk repeatedly. Shortening the length of key values can improve performance by reducing the amount of disk I/O needed to read the index and by increasing the number of key values that fit into the key cache. This applies primarily to longer **CHAR** and **VARCHAR** data types.
- Avoid creating multiple indexes that overlap (have the same initial columns). This is wasteful because MySQL can use a multiple-column index even when a query uses just the initial columns for lookups.
- The index creation process itself can be optimized if you are creating more than one index for a given table. **ALTER TABLE** can add several indexes in the same statement, which is faster than processing each one separately. **CREATE INDEX** allows only one index to be added or dropped at a time.
- Know if hash or tree indexes are better. Hash indexes are faster, but offer fewer features.

For indexed MyISAM or InnoDB tables, keeping the internal index statistics up-to-date helps the query optimizer process queries more efficiently. You can update the statistics with the **ANALYZE TABLE** statement.



595

21.3.6 Indexing Column Prefixes

Short index values can be processed more quickly than long ones. Therefore, when you index a column, ask whether it's sufficient to index partial column values rather than complete values. This technique of indexing a column prefix can be applied to string data types.

Suppose that you're considering creating a table using this definition:

```
mysql> CREATE TABLE t  
-> (name CHAR(255),  
-> INDEX (name)  
-> );
```

If you index all 255 characters of the values in the `name` column, index processing will be relatively slow:

- It's necessary to read more information from disk.
- Longer values take longer to compare.
- The prefix index cache is not as effective because fewer key values fit into it at a time.

It's often possible to overcome these problems by indexing only a prefix of the column values. For example, if you expect column values to be distinct most of the time in the first 15 characters, index only that many characters of each value, not all 255 characters.

To specify a prefix length for a column, follow the column name in the index definition by a number in parentheses. The following table definition is the same as the previous one, except that key values in the index use only the first 15 characters of the column values:

```
mysql> CREATE TABLE t  
-> (name CHAR(255),  
-> INDEX (name(15))  
-> );
```

596

Indexing a column prefix can speed up query processing, but works best when the prefix values tend to have about the same amount of uniqueness as the original values. Don't use such a short prefix that you produce a very high frequency of duplicate values in the index. It might require some testing to find the optimal balance between long index values that provide good uniqueness versus shorter values that compare more quickly but have more duplicates. To determine the number of records in the table, the number of distinct values in the column and the number of duplicates, use this query:

```
mysql> SELECT  
-> COUNT(*) AS 'Total Rows',  
-> COUNT(DISTINCT name) AS 'Distinct Values',  
-> COUNT(*) - COUNT(DISTINCT name) AS 'Duplicate Values'  
-> FROM t;
```



The query gives you an estimate of the amount of uniqueness in the `name` values. Then run a similar query on the prefix values:

```
mysql> SELECT
->   COUNT(DISTINCT LEFT(name,n)) AS 'Distinct Prefix Values',
->   COUNT(*) - COUNT(DISTINCT LEFT(name,n))
->   AS 'Duplicate Prefix Values'
->   FROM t;
```

That tells you how the uniqueness characteristics change when you use an `n`-character prefix of the `name` values. Run the query with different values of `n` to determine an acceptable prefix length.

597

21.3.7 Leftmost Index Prefixes

In a table that has a composite (multiple-column) index MySQL can use leftmost index prefixes of that index. A leftmost prefix of a composite index consists of one or more of the initial columns of the index. MySQL's capability to use leftmost index prefixes enables you to avoid creating unnecessary (redundant) indexes.

598

The `CountryLanguage` table in the `world` database provides an example of how a leftmost prefix applies. The table has a two-part primary key on the `CountryCode` and `Language` columns:

```
mysql> SHOW INDEX FROM CountryLanguage\G
***** 1. row *****
      Table: CountryLanguage
Non_unique: 0
      Key_name: PRIMARY
Seq_in_index: 1
Column_name: CountryCode
      Collation: A
Cardinality: NULL
...
      Index_type: BTREE
      Comment:
***** 2. row *****
      Table: CountryLanguage
Non_unique: 0
      Key_name: PRIMARY
Seq_in_index: 2
Column_name: Language
      Collation: A
Cardinality: 984
...
      Index_type: BTREE
      Comment:
```



The index on the **CountryCode** and **Language** columns allows records to be looked up quickly based on a given country name and language. However, MySQL also can use the index when given only a country code. Suppose that you want to determine which languages are spoken in France:

```
mysql> SELECT * FROM CountryLanguage WHERE CountryCode = 'FRA';
```

MySQL can see that **CountryCode** is a leftmost prefix of the primary key and use it as though it were a separate index. This means there's no need to define a second index on the **CountryCode** column alone.

On the other hand, if you want to perform indexed searches using just the **Language** column of the **CountryLanguage** table, you do need to create a separate index because **Language** is not a leftmost prefix of the existing index.

Note that a leftmost prefix of an index and an index on a column prefix are two different things. A leftmost prefix of an index consists of leading columns in a multiple-column index. An index on a column prefix indexes the leading characters of values in the column.

599

21.3.8 FULLTEXT Indexes

MySQL has support for full-text indexing and searching:

- A full-text index in MySQL is an index of type **FULLTEXT**.
- Full-text indexes can be used only with MyISAM tables, and can be created only for **CHAR**, **VARCHAR**, or **TEXT** columns.
- A **FULLTEXT** index definition can be given in the **CREATE TABLE** statement when a table is created, or added later using **ALTER TABLE** or **CREATE INDEX**.
- For large datasets, it is much faster to load your data into a table that has no **FULLTEXT** index and then create the index after that, than to load data into a table that has an existing **FULLTEXT** index.

Full-text searching is performed using **MATCH()** . . . **AGAINST** syntax.

```
MATCH (col1,col2,...) AGAINST (expr [search_modifier])
```

Search_modifier:

```
{
  IN BOOLEAN MODE
  | IN NATURAL LANGUAGE MODE
  | IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION
  | WITH QUERY EXPANSION
}
```

MATCH() takes a comma-separated list that names the columns to be searched. **AGAINST** takes a string to search for, and an optional modifier that indicates what type of search to perform. The search string must be a *literal string*, not a variable or a column name.



600

There are three types of full-text searches:

- A **boolean** search interprets the search string using the rules of a special query language. The string contains the words to search for. It can also contain operators that specify requirements such that a word must be present or absent in matching rows, or that it should be weighted higher or lower than usual. Common words such as “some” or “then” are stopwords and do not match if present in the search string. The **IN BOOLEAN MODE** modifier specifies a boolean search.
- A **natural language** search interprets the search string as a phrase in natural human language (a phrase in free text). There are no special operators. The stopword list applies. In addition, words that are present in more than 50% of the rows are considered common and do not match. Full-text searches are natural language searches if the **IN NATURAL LANGUAGE MODE** modifier is given or if no modifier is given.
- A **query expansion** search is a modification of a natural language search. The search string is used to perform a natural language search. Then words from the most relevant rows returned by the search are added to the search string and the search is done again. The query returns the rows from the second search. The **IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION** or **WITH QUERY EXPANSION** modifier specifies a query expansion search.

For example, given a table called **Books** that contains a collection of book titles and corresponding authors, you can do specific text searches on the titles.

id author	title
1 Dr. Seuss	Green Eggs and Ham
2 Maurice Sendak	Where The Wild Things Are
3 C.S. Lewis	The Lion, the Witch, and the Wardrobe
4 Antoine de Saint-Exupery	The Little Prince
5 Lewis Carroll	Alice in Wonderland
6 J.K. Rowling	Harry Potter and the Half-Blood Prince
7 Kathleen Olmstead	Anne of Green Gables

The create statement for this table shows the FULLTEXT index that was defined:

```
mysql> SHOW CREATE TABLE books\G
***** 1. row *****
      Table: books
Create Table: CREATE TABLE `books` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `author` varchar(64) DEFAULT NULL,
  `title` varchar(128) DEFAULT NULL,
  PRIMARY KEY (`id`),
  FULLTEXT KEY `title` (`title`)
) ENGINE=MyISAM AUTO_INCREMENT=8 DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```



601

The following statement performs a search for all titles including the word ‘**prince**’:

```
mysql> SELECT title FROM books WHERE MATCH(title) AGAINST('prince');
+-----+
| title           |
+-----+
| The Little Prince          |
| Harry Potter and the Half-Blood Prince |
+-----+
```

The following statement performs a search for all titles including both the words ‘**green**’ and ‘**Anne**’:

```
mysql> SELECT title, author FROM books WHERE MATCH(title)
      -> AGAINST('green +Anne' IN BOOLEAN MODE);
+-----+-----+
| title       | author      |
+-----+-----+
| Anne of Green Gables | Kathleen Olmstead |
+-----+-----+
```

602

21.4 Using EXPLAIN to Analyze Queries

When a **SELECT** query does not run as quickly as you think it should, use the **EXPLAIN** statement to ask the MySQL server for information about how the query optimizer processes the query. This information is useful in several ways:

- **EXPLAIN** can provide information that points out the need to add an index.
- If a table already has indexes, you can use **EXPLAIN** to find out whether the optimizer is using them.
- If indexes exist but aren’t being used, you can try writing a query different ways. **EXPLAIN** can tell you whether the rewrites are better for helping the server use the available indexes.

When using **EXPLAIN** to analyze a query, it’s helpful to have a good understanding of the tables involved. If you need to determine a table’s structure, remember that you can use **DESCRIBE** to obtain information about a table’s columns, and **SHOW INDEX** for information about its indexes.

EXPLAIN works with **SELECT** queries, but can be used in an indirect way for **UPDATE** and **DELETE** statements as well: Write a **SELECT** statement that has the same **WHERE** clause as the **UPDATE** or **DELETE** and use **EXPLAIN** to analyze the **SELECT**.



603

21.4.1 How EXPLAIN Works

To use **EXPLAIN**, write your **SELECT** query as you normally would, but place the keyword **EXPLAIN** in front of it.

In the preceding section, an example is shown in which it is stated that of the following two queries, the second can be executed more efficiently because the optimizer can use an index on the column **d**:

```
mysql> SELECT * FROM t WHERE YEAR(d) >= 1994;
mysql> SELECT * FROM t WHERE d >= '1994-01-01';
```

To verify whether MySQL actually will use an index to process the second query, use the **EXPLAIN** statement to get information from the optimizer about the execution plans it would use. For the two date-selection queries just shown, you might find that **EXPLAIN** tells you something like this:

```
mysql> EXPLAIN SELECT * FROM t WHERE YEAR(d) >= 1994\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t
        type: index
possible_keys: NULL
          key: d
     key_len: 4
       ref: NULL
      rows: 36997
    Extra: Using where; Using index

mysql> EXPLAIN SELECT * FROM t WHERE d >= '1994-01-01'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t
        type: range
possible_keys: d
          key: d
     key_len: 4
       ref: NULL
      rows: 2784
    Extra: Using where; Using index
```

604

These results indicate that the second query is indeed better from the optimizer's point of view. MySQL can perform a range scan using the index for the column **d**, drastically reducing the number of rows that need to be examined. (The **rows** value drops from 36,997 to 2,784)



605

21.4.2 EXPLAIN Output Columns

The `EXPLAIN` statement produces one row of output for each table named in each `SELECT` of the analyzed statement. (A statement can have more than one `SELECT` if it uses subqueries or `UNION`.) To use `EXPLAIN` productively, it's important to know the meaning of the columns in each row of output:

- `id` indicates which `SELECT` in the analyzed statement that the `EXPLAIN` output row refers to.
- `select_type` categorizes the `SELECT` referred to by the output row. This column can have any of the values shown in the following table. The word `DEPENDENT` indicates that a subquery is correlated with the outer query.

<code>select_type</code> Value	Meaning
SIMPLE	Simple <code>SELECT</code> statement (no subqueries or unions)
PRIMARY	The outer <code>SELECT</code>
UNION	Second or later <code>SELECT</code> in a union
DEPENDENT UNION	Second or later <code>SELECT</code> in a union that is dependent on the outer query
UNION RESULT	Result of a union
SUBQUERY	First <code>SELECT</code> in a subquery
DEPENDENT SUBQUERY	First <code>SELECT</code> in a subquery that is dependent on the outer query
DERIVED	Subquery in the <code>FROM</code> clause

- `table` is the name of the table to which the information in the row applies. The order of the tables indicates the order in which MySQL will read the tables to process the query. This is not necessarily the order in which you name them in the `FROM` clause, because the optimizer attempts to determine which order will result in the most efficient processing. The example in the preceding section showed this: The table order displayed by successive `EXPLAIN` statements changed as indexes were added.

- `type` indicates the join type. The value is a measure of how efficiently MySQL can scan the table. The possible `type` values are described later in this section.

- `possible_keys` indicates which of the table's indexes MySQL considers to be candidates for identifying rows that satisfy the query. This value can be a list of one or more index names, or `NULL` if there are no candidates. The word `PRIMARY` indicates that MySQL considers the table's primary key to be a candidate.

- `key` indicates the optimizer's decision about which of the candidate indexes listed in `possible_keys` will yield most efficient query execution. If the `key` value is `NULL`, it means no index was chosen. This might happen either because there were no candidates or because the optimizer believes it will be just as fast to scan the table rows as to use any of the possible indexes. A table scan might be chosen over an index scan if the table is small, or because the index would yield too high a percentage of the rows in the table to be of much use.

606



- **key_len** indicates how many bytes of index rows are used. From this value, you can derive how many columns from the index are used. For example, if you have an index consisting of three `INT` columns, each index row contains three 4-byte values. If `key_len` is 12, you know that the optimizer uses all three columns of the index when processing the query. If `key_len` is 4 or 8, it uses only the first one or two columns (that is, it uses a leftmost prefix of the index).
- If you've indexed partial values of string columns, take that into account when assessing the `key_len` value. Suppose that you have a composite index on two `CHAR(8)` columns that indexes only the first 4 bytes of each column. In this case, a `key_len` value of 8 means that both columns of the index would be used, not just the first column.
- **ref** indicates which indexed column or columns are used to choose rows from the table. `const` means key values in the index are compared to a constant expression, such as in `Code='FRA'`. `NULL` indicates that neither a constant nor another column is being used, indicating selection by an expression or range of values. It might also indicate that the column does not contain the value specified by the constant expression. If neither `NULL` nor `const` is displayed, a `table_name.column_name` combination will be shown, indicating that the optimizer is looking at `column_name` in the rows returned from `table_name` to identify rows for the current table.
- **rows** is the optimizer's estimate of how many rows from the table it will need to examine. The value is an approximation because, in general, MySQL cannot know the exact number of rows without actually executing the query. For a multiple-table query, the product of the `rows` values is an estimate of the total number of row combinations that need to be read. This product gives you a rough measure of query performance. The smaller the value, the better.
- **Extra** provides other information about the join. The possible values are described later in this section.

EXPLAIN for Joins:

607

EXPLAIN is especially important for join analysis because joins have such enormous potential to increase the amount of processing the server must do. If you select from a table with a thousand rows, the server might need to scan all one thousand rows in the worst case. But if you perform a join between two tables with a thousand rows each, the server might need to examine every possible combination of rows, which is one million combinations. That is a much worse, worst case. **EXPLAIN** can help you reduce the work the server must do to process such a query, so it's well worth using.

The value in the `type` column of **EXPLAIN** output indicates the join type, but joins may be performed with varying degrees of efficiency. The `type` value provides a measure of this efficiency by indicating the basis on which rows are selected from each table. The following list shows the possible values, from the best type to the worst:

- **system**
The table has exactly one row.



- **const**

The table has exactly one matching row. This `type` value is similar to `system`, except that the table may have other, non-matching rows. The `EXPLAIN` output from the query with `WHERE Code='FRA'` is an example of this:

```
mysql> EXPLAIN SELECT * FROM Country WHERE Code = 'FRA'\G
***** 1. row ****
      id: 1
  select_type: SIMPLE
        table: Country
        type: const
possible_keys: PRIMARY
         key: PRIMARY
    key_len: 3
       ref: const
      rows: 1
     Extra:
```

The query has a `type` value of `const` because only one row out of all its rows need be read. If the table contained *only* the row for France, there would be no non-matching rows and the `type` value would be `system` rather than `const`.

For both `system` and `const`, because only one row matches, any columns needed from it can be read once and treated as constants while processing the rest of the query.

- **eq_ref**

Exactly one row is read from the table for each combination of rows from the tables listed earlier by `EXPLAIN`. This is common for joins where MySQL can use a primary key to identify table rows.

- **ref**

Several rows may be read from the table for each combination of rows from the tables listed earlier by `EXPLAIN`. This is similar to `eq_ref`, but can occur when a non-unique index is used to identify table rows or when only a leftmost prefix of an index is used. For example, the `CountryLanguage` table has a primary key on the `CountryCode` and `Language` columns. If you search using only a `CountryCode` value, MySQL can use that column as a leftmost prefix, but there might be several rows for a country if multiple languages are spoken there.

608

- **ref_or_null**

Similar to `ref`, but MySQL also looks for rows that contain `NULL`.

- **index_merge**

MySQL uses an index merge algorithm.

- **unique_subquery**

Similar to `ref`, but used for `IN` subqueries that select from the primary key column of a single table.

- **index_subquery**

Similar to `unique_subquery`, but used for `IN` subqueries that select from an indexed column of a single table.



- **range**

The index is used to select rows that fall within a given range of index values. This is common for inequality comparisons such as `id < 10`.

- **index**

MySQL performs a full scan, but it scans the index rather than the data rows. An index scan is preferable: The index is sorted and index rows usually are shorter than data rows, so index rows can be read in order and more of them can be read at a time.

- **ALL**

A full table scan of all data rows. Typically, this indicates that no optimizations are done and represents the worst case. It is particularly unfortunate when tables listed later in `EXPLAIN` output have a join type of `ALL` because that indicates a table scan for every combination of rows selected from the tables processed earlier in the join.

609

EXPLAIN for Table Processing:

The `Extra` column of `EXPLAIN` output provides additional information about how a table is processed. Some values indicate that the query is efficient:

- **Using index**

MySQL can optimize the query by reading values from the index without having to read the corresponding data rows. This optimization is possible when the query selects only columns that are in the index.

- **Where used**

MySQL uses a `WHERE` clause to identify rows that satisfy the query. Without a `WHERE` clause, you get all rows from the table.

- **Distinct**

MySQL reads a single row from the table for each combination of rows from the tables listed earlier in the `EXPLAIN` output.

- **Not exists**

MySQL can perform a `LEFT JOIN` “missing rows” optimization that quickly eliminates rows from consideration.

610

By contrast, some `Extra` values indicate that the query is not efficient:

- **Using filesort**

Rows that satisfy the query must be sorted, which adds an extra processing step.

- **Using temporary**

A temporary table must be created to process the query.

- **Range checked for each record**

MySQL cannot determine in advance which index from the table to use. For each combination of rows selected from previous tables, it checks the indexes in the table to see which one will be best. This is not great, but it's better than using no index at all.

Using filesort and **Using temporary** generally are the two indicators of extremely poor performance.



To use **EXPLAIN** for query analysis, examine its output for clues to ways the query might be improved. Modify the query, and then run **EXPLAIN** again to see how its output changes. Changes might involve rewriting the query or modifying the structure of your tables.

611

21.5 Query Rewriting Techniques

The way you write a query often affects how well indexes are used. Use the following principles to make your queries more efficient:

- Do not refer to an indexed column within an expression that must be evaluated for every row in the table. Doing so prevents use of the index. Instead, isolate the column onto one side of a comparison when possible. Suppose that a table **t** contains a **DATE** column **d** that is indexed. One way to select rows containing date values from the year 1994 and up is as follows:

```
mysql> SELECT * FROM t WHERE YEAR(d) >= 1994;
```

In this case, the value of **YEAR(d)** must be evaluated for every row in the table, so the index cannot be used. Instead, write the query like this:

```
mysql> SELECT * FROM t WHERE d >= '1994-01-01';
```

In the rewritten expression, the indexed column stands by itself on one side of the comparison and MySQL can apply the index to optimize the query.

- Indexes are particularly beneficial for joins that compare columns from two tables. Consider the following join:

```
mysql> SELECT * FROM Country JOIN CountryLanguage
      ->   ON Country.Code = CountryLanguage.CountryCode;
```

If neither the **Code** or **CountryCode** column is indexed, every pair of column values must be compared to find those pairs that are equal. For example, for each **Code** value from the **CountryLanguage** table, MySQL would have to compare it with every **CountryCode** value from the **CountryCode** table. If instead **CountryCode** is indexed, then for each **Code** value that MySQL retrieves, it can use the index on **CountryCode** to quickly look up the rows with matching values. (In practice, you'd normally index both of the joined columns when you use inner joins because the optimizer might process the tables in either order.)

- When comparing an indexed column to a value, use a value that has the same data type as the column. For example, you can look for rows containing a numeric **id** value of **18** with either of the following **WHERE** clauses:

```
WHERE id = 18
WHERE id = '18'
```



612

MySQL will produce the same result either way, even though the value is specified as a number in one case and as a string in the other case. However, for the string value, MySQL must perform a string-to-number conversion, which might cause an index on the `id` column not to be used.

- In certain cases, MySQL can use an index for pattern-matching operations performed with the `LIKE` operator. This is true if the pattern begins with a literal prefix value rather than with a wildcard character. An index on a `name` column can be used for a pattern match like this:

```
WHERE name LIKE 'de%'
```

That's because the pattern match is logically equivalent to a search for a range of values:

```
WHERE name >= 'de' AND name < 'df'
```

On the other hand, the following pattern makes `LIKE` more difficult for the optimizer:

```
WHERE name LIKE '%de%'
```

When a pattern starts with a wildcard character as just shown, MySQL cannot use indexes associated with that column. (That is, even if an index *is* used, the entire index must be scanned.)

- For queries written like this:

```
WHERE LENGTH(column)=5
```

Consider adding triggers that maintain an additional column for the length of a column, and rewrite the query as follows:

```
WHERE column_length=5
```

613

21.6 Optimizing Queries by Limiting Output

Some optimizations can be done independently of whether indexes are used. A simple but effective technique is to reduce the amount of output a query produces.

One way to eliminate unnecessary output is by using a `LIMIT` clause in a `SELECT` query. If you do not need the entire result set, specify how many rows the server should return by including `LIMIT` in your query. This helps in two ways:

- Less information need be returned over the network to the client.
- In many cases, `LIMIT` allows the server to terminate query processing earlier than it would otherwise. Some row-sorting techniques have the property that the first `n` rows can be known to be in the final order even before the sort has been done completely. This means that when `LIMIT n` is combined with `ORDER BY`, the server might be able to determine the first `n` rows and then terminate the sort operation early.



Don't use **LIMIT** as a way to pull out just a few rows from a gigantic result set. For example, if a table has millions of rows, the following statement does not become efficient simply because it uses **LIMIT**:

```
mysql> SELECT * FROM t LIMIT 10;
```

Instead, try to use a **WHERE** clause that restricts the result so that the server doesn't retrieve as many rows in the first place.

Another way to reduce query output is to limit it "horizontally." Select only the columns you need, rather than using **SELECT *** to retrieve all columns. Suppose that you want information about countries having names that begin with '**M**'. The following query produces that information, but it also produces every other column as well:

```
mysql> SELECT * FROM Country WHERE Name LIKE 'M%';
```

If all you really want to know is the country names, do not write the query like that. Most of the information retrieved will be irrelevant to what you want to know, resulting in unnecessary server effort and network traffic. Instead, select specifically just the **Name** column:

```
mysql> SELECT Name FROM Country WHERE Name LIKE 'M%';
```

The second query is faster because MySQL has to return less information when you select just one column rather than all of them.

614

21.7 Using Summary Tables

Suppose that you run an analysis consisting of a set of retrievals that each perform a complex **SELECT** of a set of records (perhaps using an expensive join), and that differ only in the way they summarize the records. That's inefficient because it unnecessarily does the work of selecting the records repeatedly. A better technique is to select the records once, and then use them to generate the summaries. In such a situation, consider the following strategy:

1. Select the set of to-be-summarized records into a temporary table. In MySQL, you can do this easily with a **CREATE TABLE ... SELECT** statement. If the summary table is needed only for the duration of a single client connection, you can use **CREATE TEMPORARY TABLE ... SELECT** and the table will be dropped automatically when you disconnect.
2. Create any appropriate indexes on the temporary table.
3. Select the summaries using the temporary table.

The technique of using a summary table has several benefits:

- Calculating the summary information a single time reduces the overall computational burden by eliminating most of the repetition involved in performing the initial record selection.

If the original table is a type that is subject to table-level locking, such as a **MyISAM** table, using a summary table leaves the original table available more of the time for updates by other clients by reducing the amount of time that the table remains locked. (**Note:** It is possible to use triggers to update summary tables, although it might not always be worth the trouble.)



- If the summary table is small enough that it is reasonable to hold in memory, you can increase performance even more by making it a **MEMORY** table. Queries on the table will be especially fast because they require no disk I/O. When the **MEMORY** table no longer is needed, drop it to free the memory allocated for it.

The following example creates a summary table containing the average GNP value of countries in each continent. Then it compares the summary information to individual countries to find those countries with a GNP much less than the average and much more than the average.

615

First, create the summary table:

```
mysql> CREATE TABLE ContinentGNP
->   SELECT Continent, AVG(GNP) AS AvgGNP
->   FROM Country GROUP BY Continent;

mysql> SELECT * FROM ContinentGNP;
+-----+-----+
| Continent | AvgGNP |
+-----+-----+
| Asia      | 150105.725490 |
| Europe    | 206497.065217 |
| North America | 261854.789189 |
| Africa    | 10006.465517 |
| Oceania   | 14991.953571 |
| Antarctica | 0.000000 |
| South America | 107991.000000 |
+-----+-----+
```

616

Next, compare the summary table to the original table to find countries that have a GNP less than 1% of the continental average:

```
mysql> SELECT Country.Continent, Country.Name, Country.GNP AS CountryGNP,
->   ContinentGNP.AvgGNP AS ContinentAvgGNP FROM Country, ContinentGNP
->   WHERE Country.Continent = ContinentGNP.Continent
->   AND Country.GNP < ContinentGNP.AvgGNP * .01
->   ORDER BY Country.Continent, Country.Name;
+-----+-----+-----+-----+
| Continent | Name       | CountryGNP | ContinentAvgGNP |
+-----+-----+-----+-----+
| Asia      | Bhutan     | 372.00    | 150105.725490 |
| Asia      | East Timor | 0.00      | 150105.725490 |
| Asia      | Laos        | 1292.00   | 150105.725490 |
...
...
```

617

Disadvantages of Summary Tables:

- Up to date only as long as the original values remain unchanged.
- Storing data twice; original table and newly created summary table.





InLine Lab 21-D

In this exercise you will use various **optimization** techniques. This will require access to the mysql server and the **world** database.

Step 1. Display how MySQL will execute a statement

1. Obtain optimization information about the **City** table for the district of ‘California’ by executing the following statement:

```
mysql> EXPLAIN SELECT * FROM City WHERE District='California' \G
```

Lists the optimization for the specified table condition;

```
***** 1. row *****
    id: 1
  select_type: SIMPLE
        table: city
        type: ALL
possible_keys: NULL
         key: NULL
    key_len: NULL
        ref: NULL
       rows: 4321
     Extra: Using where
```

Step 2. Add index and re-display how MySQL will execute a statement

1. Add an index for the column called ‘**District**’ to the **City** table:

```
mysql> ALTER TABLE City ADD INDEX (District);
```

Returns OK.



2. Confirm the index change made above by obtaining optimization information about the **City** table for the district of ‘California’ by executing the following statement:

```
mysql> EXPLAIN SELECT * FROM City WHERE District='California'\G
```

Lists the optimization for the specified table condition, with the change of index information

```
***** 1. row *****
    id: 1
  select_type: SIMPLE
        table: city
        type: ref
possible_keys: District
            key: District
       key_len: 20
        ref: const
       rows: 68
     Extra: Using where
```

Step 3. Review other index information for the City table

1. Obtain optimization information about the **City** table, for the **ID** of ‘3803’ by executing the following statement:

```
mysql> EXPLAIN SELECT * FROM city WHERE id=3803\G
```

Lists the optimization for the specified table condition;

```
***** 1. row *****
    id: 1
  select_type: SIMPLE
        table: city
        type: const
possible_keys: PRIMARY
            key: PRIMARY
       key_len: 4
        ref: const
       rows: 1
     Extra:
```



Step 4. Create and use a summary table

1. Create a summary table called ‘ContinentGNP’ by entering the following statement:

```
mysql> CREATE TABLE ContinentGNP
->   SELECT Continent, AVG(GNP) AS AvgGNP FROM Country
->   GROUP BY Continent;
```

The table is created.

2. Use the summary table to find countries that have a GNP more than 10 times the continental average:

```
mysql> SELECT Country.Continent, Country.Name, Country.GNP AS CountryGNP,
->   ContinentGNP.AvgGNP AS ContinentAvgGNP FROM Country, ContinentGNP
->   WHERE Country.Continent = ContinentGNP.Continent
->   AND Country.GNP > ContinentGNP.AvgGNP * 10
->   ORDER BY Country.Continent, Country.Name;
```

Lists the 5 continents and the GNP information specified in the summary compare query;

Continent	Name	CountryGNP	ContinentAvgGNP
Asia	Japan	3787042.00	150105.725490
Europe	Germany	2133367.00	205536.911111
North America	United States	8510700.00	261854.789189
Africa	South Africa	116729.00	9548.263158
Oceania	Australia	351182.00	14991.953571

5 rows in set (#.# sec)



618

21.8 Optimizing Updates

The optimizations discussed so far have been shown for **SELECT** statements, but optimization techniques can be used for statements that update tables, too:

- For a **DELETE** or **UPDATE** statement that uses a **WHERE** clause, try to write it in a way that allows an index to be used for determining which rows to delete or update. The techniques for this that were discussed earlier for **SELECT** statements apply to **DELETE** and **UPDATE** as well.
- EXPLAIN** is used with **SELECT** queries, but you might also find it helpful for analyzing **UPDATE** and **DELETE** statements. Write a **SELECT** that has the same **WHERE** clause as the **UPDATE** or **DELETE** and analyze that.
- Use multiple-row **INSERT** statements instead of multiple single-row **INSERT** statements. For example, instead of using three single-row statements like this:

```
mysql> INSERT INTO t (id, name) VALUES(1, 'Bea');
mysql> INSERT INTO t (id, name) VALUES(2, 'Belle');
mysql> INSERT INTO t (id, name) VALUES(3, 'Bernice');
```

You could use a single multiple-row statement that does the same thing:

```
mysql> INSERT INTO t (id, name) VALUES (1,'Bea'),(2,'Belle'),(3,'Bernice');
```

The multiple-row statement is shorter, which is less information to send to the server. More important, it allows the server to perform all the updates at once and flush the index a single time, rather than flushing it after each of the individual inserts. This optimization can be used with any storage engine.

If you're using an InnoDB table, you can get better performance even for single-row statements by grouping them within a transaction rather than by executing them with autocommit mode enabled:

619

```
mysql> START TRANSACTION;
mysql> INSERT INTO t (id, name) VALUES(1, 'Bea');
mysql> INSERT INTO t (id, name) VALUES(2, 'Belle');
mysql> INSERT INTO t (id, name) VALUES(3, 'Bernice');
mysql> COMMIT;
```

Using a transaction allows InnoDB to flush all the changes at commit time. In autocommit mode, InnoDB flushes the changes for each **INSERT** individually.

- For any storage engine, **LOAD DATA INFILE** is even faster than multiple-row **INSERT** statements.
- You can disable index updating when loading data into an empty **MyISAM** table to speed up the operation. **LOAD DATA INFILE** does this automatically for non-unique indexes if the table is empty; it disables index updating before loading and enables it again after loading.
- To replace existing rows, use **REPLACE** rather than **DELETE** plus **INSERT**.



620

21.9 Choosing Appropriate Storage Engines

When creating a table, ask yourself what types of queries you'll use it for. Then choose a storage engine that uses a locking level appropriate for the anticipated query mix. MyISAM table-level locking works best for a query mix that is heavily skewed toward retrievals and includes few updates. Use InnoDB if you must process a query mix containing many updates. InnoDB's use of row-level locking and multi-versioning provides good concurrency for a mix of retrievals and updates. One query can update rows while other queries read or update different rows of the table.

If you're using **MyISAM** tables, choose their structure to reflect whether you consider efficiency of processing speed or disk usage to be more important. Different **MyISAM** storage formats have different performance characteristics. This influences whether you choose fixed-length or variable-length columns to store string data:

- Use fixed-length columns (**CHAR**, **BINARY**) for best speed. Fixed-length columns allow MySQL to create the table with fixed-length rows. The advantage is that fixed-length rows all are stored in the table at positions that are a multiple of the row length and can be looked up very quickly. The disadvantage is that fixed-length values are always the same length even for values that do not use the full width of the column, so the column takes more storage space.
- Use variable-length columns (**VARCHAR**, **VARBINARY**, **TEXT**, **BLOB**) for best use of disk space. For example, values in a **VARCHAR** column take only as much space as necessary to store each value and on average use less storage than a **CHAR** column. The disadvantage is that variable-length columns result in variable-length rows. These are not stored at fixed positions within the table, so they cannot be retrieved as quickly as fixed-length rows. In addition, the contents of a variable-length row might not even be stored all in one place, another source of processing overhead.

Another option with MyISAM tables is to use compressed read-only tables.

621

For InnoDB tables, it is also true that **CHAR** columns take more space on average than **VARCHAR**. But there is no retrieval speed advantage for InnoDB as there is with MyISAM, because the InnoDB engine implements storage for both **CHAR** and **VARCHAR** in a similar way. In fact, retrieval of **CHAR** values might be slower because on average they require more information to be read from disk.

Use the **MEMORY** storage engine to store data that you either do not need persistently or that you can regenerate from disk based tables. Session variables are an example.





Quiz

In this exercise you will answer questions pertaining to **Optimization**.

1. Under what circumstances can adding indexes to a table make table operations slower?
2. Indexing improves performance of **SELECT** queries only; it deteriorates performance of queries that change data. Is this true?
3. “For my purposes, it doesn't matter whether a query returns a result within one second or one minute, so I do not care about indexing.” Do you object to anything about that statement?
4. What are the main reasons **not** to index table columns?
5. Here's an excerpt of the **Country** table definition:

```
mysql> DESCRIBE Country\G
...
***** 3. row *****
Field: Continent
Type: enum('Asia','Europe','North America','Africa','Oceania',
'Antarctica','South America')
Null: NO
Key:
Default: Asia
Extra:
***** 4. row *****
...
...
```

If the **Continent** column had a **PRIMARY** key on it, how many rows could it have? Would the situation be different if it had a **UNIQUE** key on it?

6. What do you have to consider when making index values as short as possible?
7. What's the statement for creating a five character-long index on the **Name** column of the **City** table?
8. What are the main reasons why the use of the **LIMIT** clause can help improve the performance of queries?
9. What could you do to speed up data insertion operations?



622

21.10 Chapter Summary

This chapter introduced **Optimization** in MySQL. In this chapter, you learned to:

- Describe the strategies available for optimizing queries
- Use the **EXPLAIN** statement to predict query performance
- Choose the most optimal storage engine for your query types
- Use Indexes for optimization



22 CONCLUSION

22.1 Learning Objectives

This chapter concludes the course and allows the instructor to answer any remaining questions you may have at this time. It also contain additional information regarding MySQL training and certification. You should now be able to:

624

- Describe specific contents of the MySQL training and certification web pages
- Complete a course evaluation, to aid in continuing improvements to MySQL courses
- Use the various contact information for additional training information and support
- Find additional training information



Instructor note:

Prior to delivering the conclusion chapter, please do a quick review of the course contents/objectives. There is no repeat of the course objectives in the guide, nor in the slides. You may use the last “**Course Contents**” slide to aid in a quick review, or bring back up the objectives slide from the beginning of the course (if time permits).

625

22.2 Training and Certification Website

View the course catalog, schedule, certification program information, venue information (and more) on the Training and Certification website: <http://www.mysql.com/training/>

The screenshot shows the MySQL AB :: MySQL Training and Certification - Mozilla Firefox window. The page features a header with the MySQL logo, a search bar, and language selection buttons. A navigation menu at the top includes Home, Products, Services, Partners & Solutions, Community, Customers, Why MySQL?, News & Events, About, and How to Buy. On the left, a sidebar titled 'Training & Certification' lists options like Certification, Curriculum Paths, Catalog, Schedule, Delivery Options, Savings and Promotions, Training Bundles, Training Partners, Testimonials, FAQ, Consulting, and Support. Below this is a 'Training Bundles' section with a list of benefits. A central banner for 'MySQL Training and Certification' features a woman working at a desk with a laptop and coffee cup, with the text 'Comprehensive Training to Help You Succeed'. It also mentions MySQL offers a comprehensive set of training courses for building world-class database solutions. A 'Featured Classes' section highlights 'MySQL Cluster for High Availability' (Jan 14: Sydney, Jan 23: San Francisco), 'MySQL Boot Camp' (Nov 26: Washington DC, Jan 21: Boston), and 'MySQL for Beginners' (Nov 27: Toronto, Jan 07: Columbus, OH). Two promotional boxes for 'MySQL Boot Camp' and 'E-Learning: MySQL Partitioning' are shown, each with a 'Register Now' button. At the bottom, there's a section for 'MySQL Virtual Courses' with a note about no travel required, a 'Leave a message' form, and a note that the user is currently offline.



On the site, you will find links to:

- **Certification** – Description of current certification program. Links to exam registration and more information.
- **Curriculum Paths** – The categories and order in which training courses should be taken.
- **Catalog** – Comprehensive course catalog. Links to more information on courses.
- **Schedule** – Schedule of courses, listed by title. Links to course registration and more information.
- **Delivery Options** – The available options for training delivery.
- **Savings and Promotions** – Information about training bundles and credits.
- **Training Bundles** – Available bundles and pricing.
- **Training Partners** – List and links of MySQL training partners.
- **Testimonials** – Training endorsements.
- **FAQ** – Frequently asked questions and links to more other training information.

Additional training information, including Articles, White Papers and Web Seminars, can be found on the community **Developer Zone** web page: <http://dev.mysql.com/>

**Instructor note:**

Just a reminder of where they can go for more info. Go through the brief descriptions of each menu link. You may want to ask students if there is anything in particular that they want to know how to find on the training/certification/main web site.

626

22.3 Course Evaluation

We need your evaluation!

Please take the time to complete a course evaluation to let us know about your experience. We are continuously making improvements to our courses and our overall training program. The feedback of our students is invaluable in assisting us to make the right choices.

Please follow the instructions given by your instructor regarding completion of the evaluation form.

Thank you in advance for taking the time to give us your opinions!

627

22.4 Thank you!

We appreciate your attendance at this course. Congratulations on your completion. We hope that your training needs regarding this course topic have been fully met. Do not hesitate to contact us, if you have any training questions:

Website: <http://www.mysql.com/training/>

Email: training@mysql.com

Phone: USA Toll Free: 1-866-697-7522

Worldwide: 1-208-514-4780

Instructor note:

Instructor Notes: Using the URL on the slide, have students fill out an online evaluation. The URL is on the slide: <http://www.mysql.com/training/evaluation.php>.

This link will direct the attendees to an attendee profile page (hosted by Rainmaker Systems). The attendees will be prompted to enter their first name, last name and email address to access their profile details and the course survey (evaluation). Once the attendee has successfully reached his/her profile page, they will see a link under course name for Evaluation; select the link and the attendee will be directed to a separate page where he/she complete the survey (submit the survey by selecting finish).



The only part that is a bit difficult to manage is that the **First name, last name and email must be identical to the information used to register for the course.**

**** Make sure to tell students that evaluations will assist in improving future revisions of this (and other) courseware. ****

628

22.5 Q&A Session

If you have more questions after the class, get answers from the MySQL Reference Manual **FAQ** page:
<http://dev.mysql.com/doc/refman/5.1/en/faqs.html>.

And if you would like to work on the exercises after completing this course, you may download the **world** database (which was used throughout the course) from the **Documentation** page of our website: <http://dev.mysql.com/doc> .

You may record any questions and answers that are shared at this point in the class, in the space provided below:

