MySQL for Developers

Electronic Presentation

SQL-4501 Release 2.2

D61830GC10
Edition 1.0

Time for Individual Presentations!

- What is your background?
  - Your name and your organization
  - How does your organization use MySQL?
- Why are you here?
  - What do you expect from the course?
  - What do you want to learn?

- Have experience using MySQL?
- Have experience from other SQL dialects?
- Have installed MySQL yourself?
- Use MySQL under Linux? Windows? Solaris? MacOSX? Other?
- Use MySQL 3.21? 3.22? 3.23? 4.0? 4.1? 5.0? 5.1?
- Use MySQL with PHP? Perl? Java? C? ODBC? Others?
- Are you MySQL Certified? Which certifications?

Practical Checklist

- Participant list -- email addresses -- checkup email
- Breaks -- timing, duration
- Where are the toilets/restrooms?
- Lunch arrangements
- Time allotted for exercises -- Solutions
- When does the day end? Can you stay longer?
- When do the days begin? Can you come earlier?
- Evaluation forms

Slide Format (Topic)

- Presentation sub-topics
  - More sub-topic information

Topics and Sub-Topics, presented in detail by instructor, Following Student Guide content

Course Name and Version

```
mysql -h training.mysql.com -u wld world
```

Code Examples

Current Chapter Title

Current *<chapter.section>* Numbers and Title

Course Objectives (1/2)

- Describe the MySQL client/server architecture
- Invoke MySQL client programs
- Decribe 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

Course Objectives (2/2)

- 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

Course Content

DEVELOPER I

1. INTRODUCTION
2. MySQL CLIENT/SERVER CONCEPTS
3. MySQL CLIENTS
4. QUERYING FOR TABLE DATA
5. HANDLING ERRORS AND WARNINGS
6. DATA TYPES
7. SQL EXPRESSIONS
8. OBTAINING METADATA
9. DATABASES
10. TABLES
11. MANIPULATING TABLE DATA
12. TRANSACTIONS

DEVELOPER II

13. JOINS
14. SUBQUERIES
15. VIEWS
16. PREPARED STATEMENTS
17. EXPORTING AND IMPORTING DATA
18. STORED ROUTINES
19. TRIGGERS
20. STORAGE ENGINES
21. OPTIMIZATION
22. CONCLUSION

Learning Objectives

- 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

MySQL Overview

- Relational Database Management System program suite
- World's most popular open source database
  - Fastest growing with over 70,000 downloads per day
- Originally developed by MySQL AB (Sweden)

MySQL is installed on every continent in the world

Yes, even Antarctica!

Sun Acquisition of MySQL

- Acquired by Sun Microsystems in 2008
  - Giving MySQL the considerable resources of a major mainstream company
- Sun and MySQL together will provide powerful, global product and services
  - Enterprise class support 24x7x365
  - More resources to draw from
  - More platform choices now available
- Both companies are strong open source supporters

Sun/MySQL Mission:

Make superior database software available and affordable to all!

You Are in Good Company!

Web / Web 2.0

OEM / ISV's

On Demand, SaaS, Hosting

Telecommunications

Enterprise 2.0

Open-source is powering the World!

MySQL Database Products

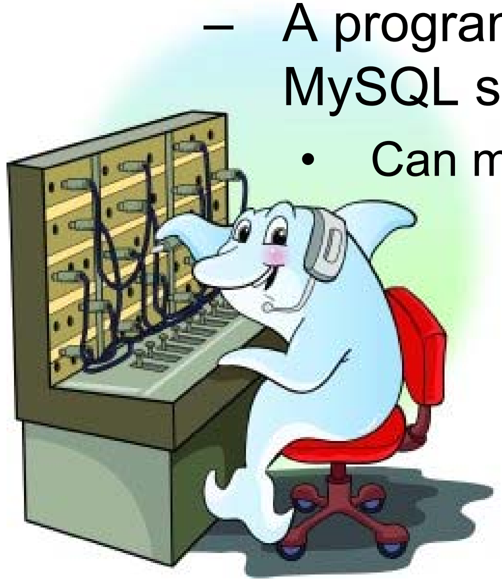
- Enterprise server
  - Enterprise-grade database
- Community server
  - Database server for open source developers
- Embedded database
  - Database server for OEMs/ISVs to bundle cost-effectively
- Cluster
  - Fault tolerant database clustering architecture

MySQL GUI Tools

- Graphical user interfaces to your MySQL database
- MySQL Migration Toolkit
  - Migration GUI Wizard
- MySQL Administrator
  - Administration console
- MySQL Query Browser
  - Create databases, execute and optimize SQL queries

Other MySQL Tools

- MySQL Workbench
  - Visual database design tool
    - Used to efficiently design, manage and document databases
- MySQL Proxy
  - A program that communicates between a client and a MySQL server
    - Can monitor, analyze or transform communication



MySQL Drivers

- MySQL C API
  - Uses native client library (`libmysql`) which can be wrapped by other languages
- MySQL Connector/ODBC
  - ODBC driver for Windows and Unix-like systems (uses `libmysql`)
- MySQL Connector/J
  - JDBC 4.0 driver (for Java 1.4 and higher)
- MySQL Connector/Net
  - Fully managed ADO.NET provider (for .NET Framework 1.1 and higher)
- MySQL Connector/PHP
  - `mysql` and `mysqli` extensions using `libmysql`
  - `mysqlnd` - the PHP native driver

Solutions for Embedding MySQL

- MySQL also provides libraries to embed a database
- libmysqld
  - Embedded edition of the **`mysqld`** server program wrapped in a shared library
  - Allows the MySQL to be embedded in C programs
- MySQL MXJ
  - A JAR wrapper around **`mysqld`** binaries
  - Allows java programs and J2EE environments to instantiate (and install) a MySQL server

MySQL Services

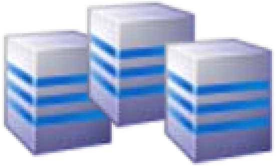
- MySQL Training
  - Comprehensive set of MySQL training courses
- MySQL Certification
  - High quality certification for MySQL Developers and Database Administrators
- MySQL Consulting
  - Full range of consulting services from start-up to optimization
- MySQL Support
  - Community
  - Enterprise (and other levels of purchased support)

MySQL Enterprise Subscription

- Annual offering
- Access to Enterprise Server and other premium products
- Enterprise-grade software, support and services
- Highest level of reliability, security and availability
- Pro-active services help eliminate problems *before* they occur

MySQL Enterprise Server

- Most reliable, secure, updated version of MySQL
- Updates and service packs
- Emergency hot fix builds
- Drivers
- MySQL Workbench
- Many different supported platforms



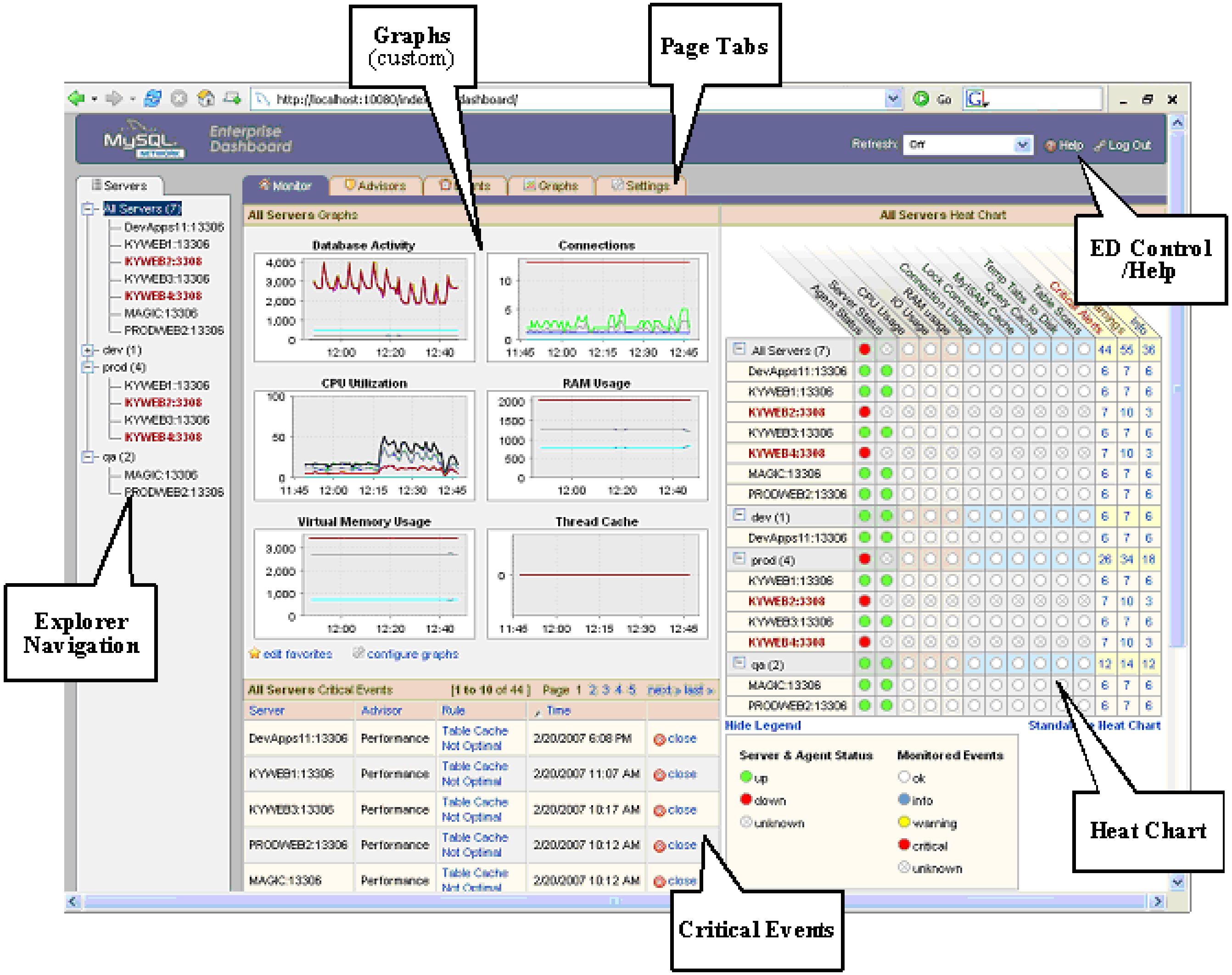
MySQL Enterprise Support

- 24x7 production-level support
- Flexible service levels available
- High priority problem resolution
- Consultative support
- Technical Account Manager (TAM)
- Online Knowledge Base

MySQL Enterprise Monitor

- Web-based monitoring and advising system
- Virtual DBA assistant
- Runs completely within corporate firewall
- Principle features:
  - Enterprise Dashboard
  - Server/group management
  - "At-a-glance" monitoring
  - MySQL and custom advisors and rules
  - Advisor rule scheduler
  - Customizable thresholds and alerts
  - Events and Alert history
  - Specialized scale-out assistance

Enterprise Dashboard



MySQL Enterprise Advisors

Administration
- Monitors and Advises on Optimal Configuration and Variable Settings
- Ensures Recoverability

Security
- Monitors and Advises on Unplanned Security Changes
- Uncovers Security loopholes

Upgrade
- Monitors and Advises Bugs that affect current installation
- Provides update path to correcting MRU/QSP

Custom
- Built by DBA to Enforce Organization specific best practices.
- Create New or Tailor MySQL Advisors to fit needs.

Replication
- Monitors and Advises on Master/Slave Latency Issues
- Makes suggestions for improving replication design

Memory Usage
- Monitors for optimal use of memory/cache
- Advises on memory tuning for better performance

Schema
- Monitors and Advises on Unplanned Schema Change
- Uncovers Security loopholes

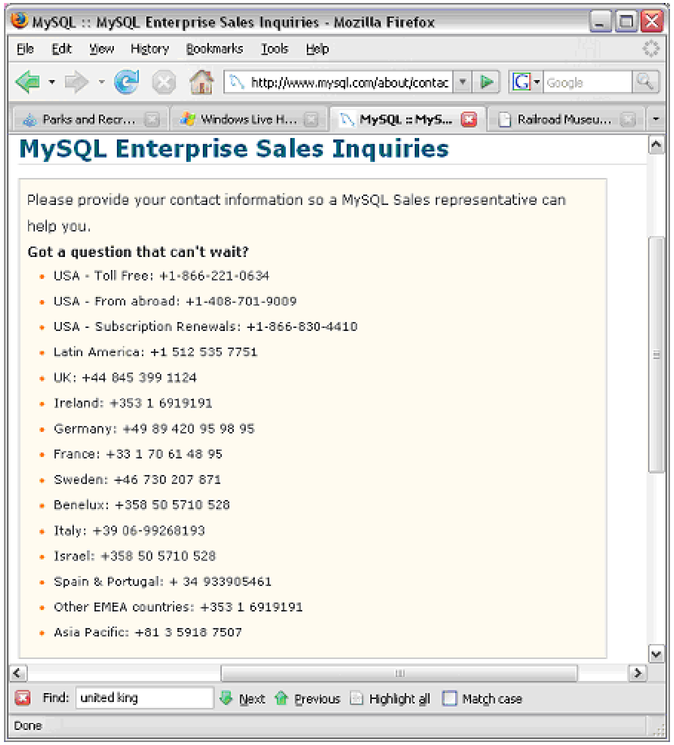
Performance
- Monitors and Advises on Optimal Performance Variable Settings
- Identifies performance bottlenecks

Enterprise Monitor Subscription Levels

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

Obtaining a MySQL Enterprise Subscription

- Contact sales personnel
- Purchase from our website
  - https://shop.mysql.com/enterprise/
- 30 day trial
  - Limited features
  - http://www.mysql.com/trials/



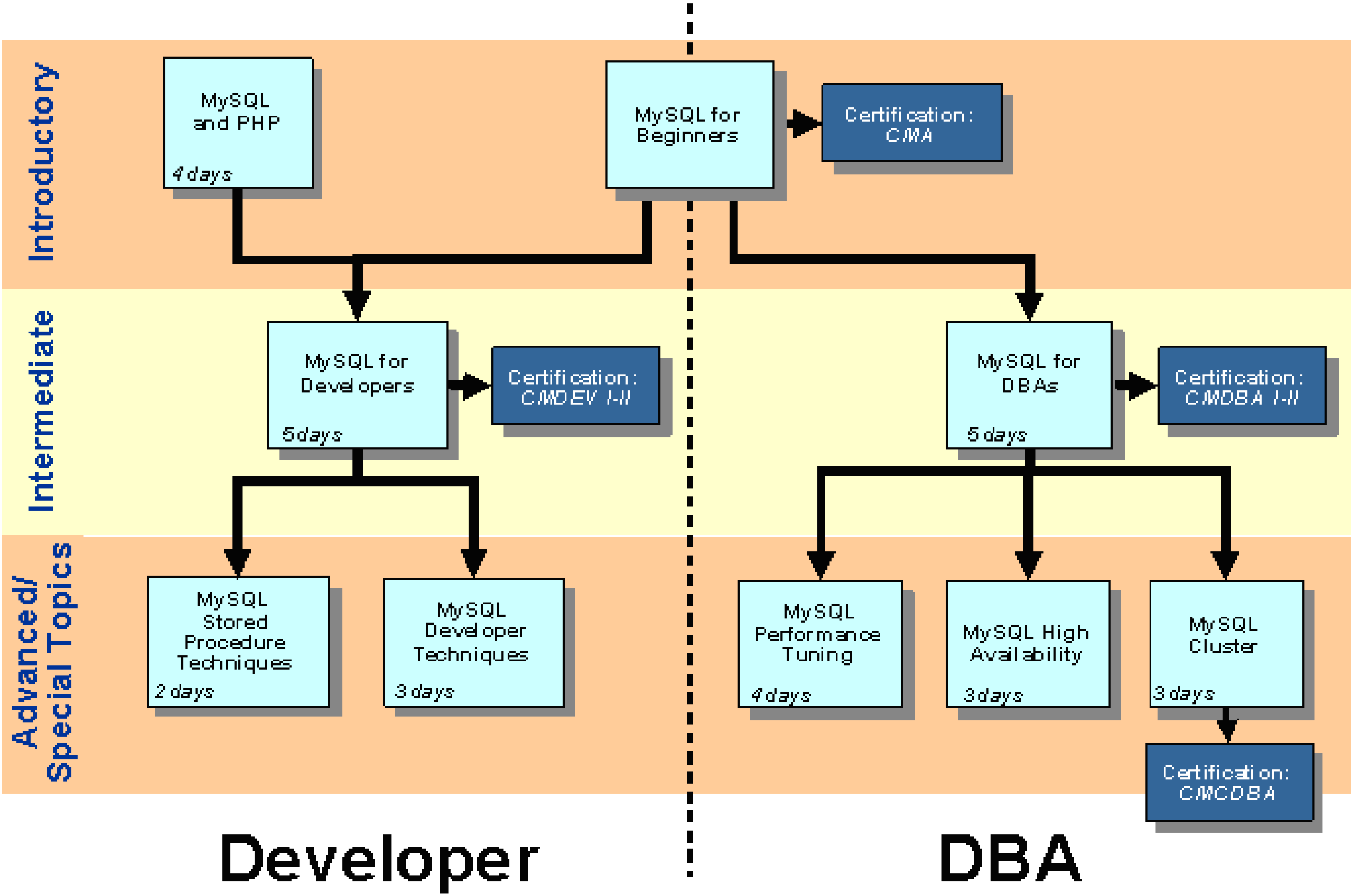
MySQL Supported Operating Systems

- More than 20 platforms
- Control and flexibility for users
- Currently available for MySQL download:
  - Windows (multiple)
  - Linux (multiple)
  - Solaris
  - FreeBSD
  - Mac OS X
  - HP-UX
  - IBM AIX and i5
  - QNX
  - Open BSD
  - SGI Irix
  - Novell NetWare
  - Source Code
  - Special Builds

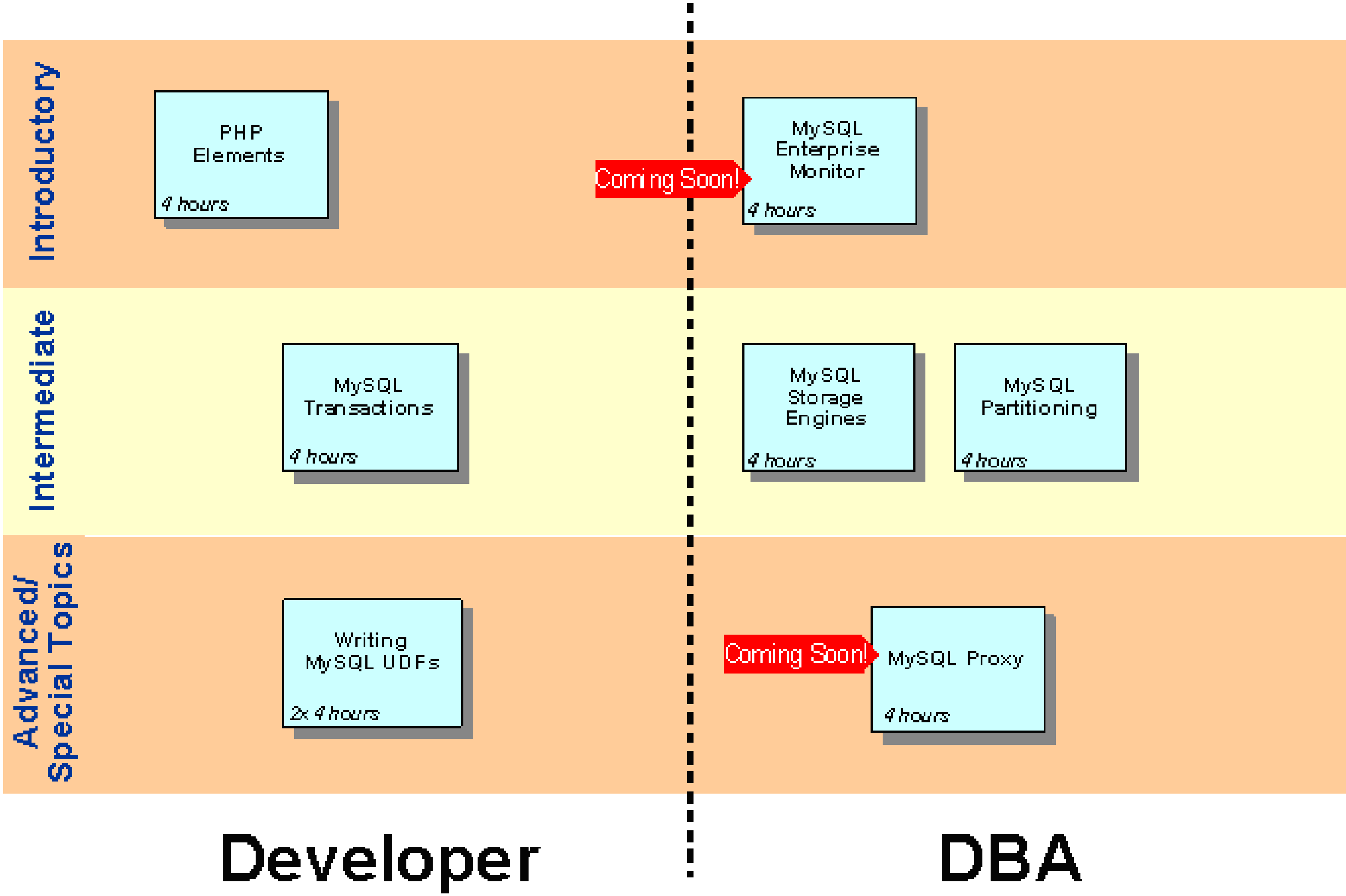
Certification Program Overview

- MySQL certifications available
  - Certified MySQL Associate (CMA)
  - Certified MySQL 5.0 Developer (CMDEV)
  - Certified MySQL 5.0 Database Administrator (CMDBA)
  - Certified MySQL Cluster DBA (CMCDBA)
- Certification web page
  - http://www.mysql.com/certification/
- Exam administration
  - Pearson VUE

Curriculum Paths

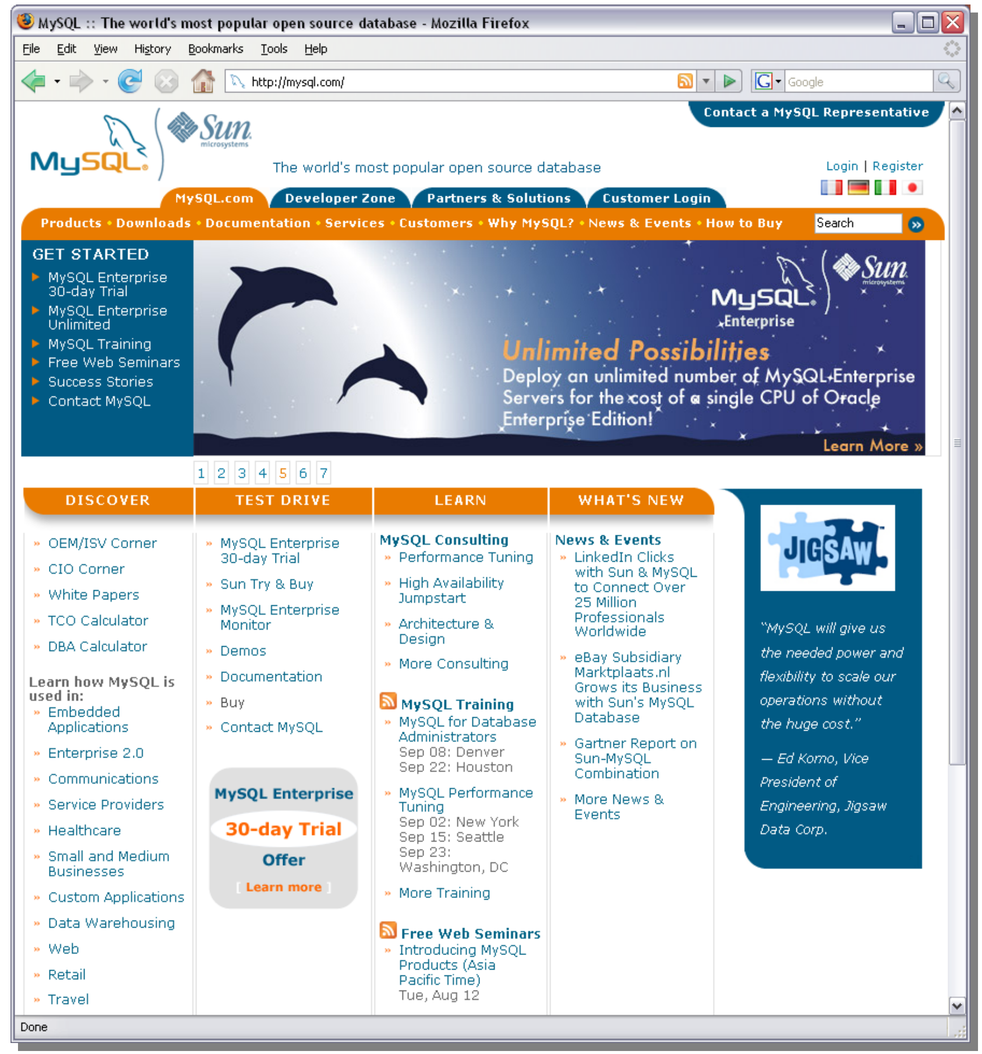


Virtual Classes



MySQL Website

- http://www.mysql.com/



MySQL Community Web Page (1/2)

- **Developer Zone**
- http://dev.mysql.com/

MySQL Community Web Page (2/2)

- Current product/service promotions
- Get Started with MySQL
- Developing with...
- Quality Contribution Program
- MySQL Server Community Edition
- New Releases
- Software Previews
- What's New
- MySQL Training
- MySQL Quickpoll
- Stay Connected
- Resources

MySQL Online Documentation

- 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

Installing MySQL

- Use the MySQL website to download
  - http://dev.mysql.com/downloads
- Several different platforms are supported
- “Windows” used for this course

Installing 'world' Database

- MySQL provides three example databases
- Can be downloaded from our website
- **'world'** database will be used for demonstration and exercises in this course

Chapter Summary

- 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

Course Content

Learning Objectives

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

MySQL General Architecture

- Networked environment using client/server
- Components of MySQL installation
  - MySQL server
  - Client programs
  - Non-client programs

MySQL Server

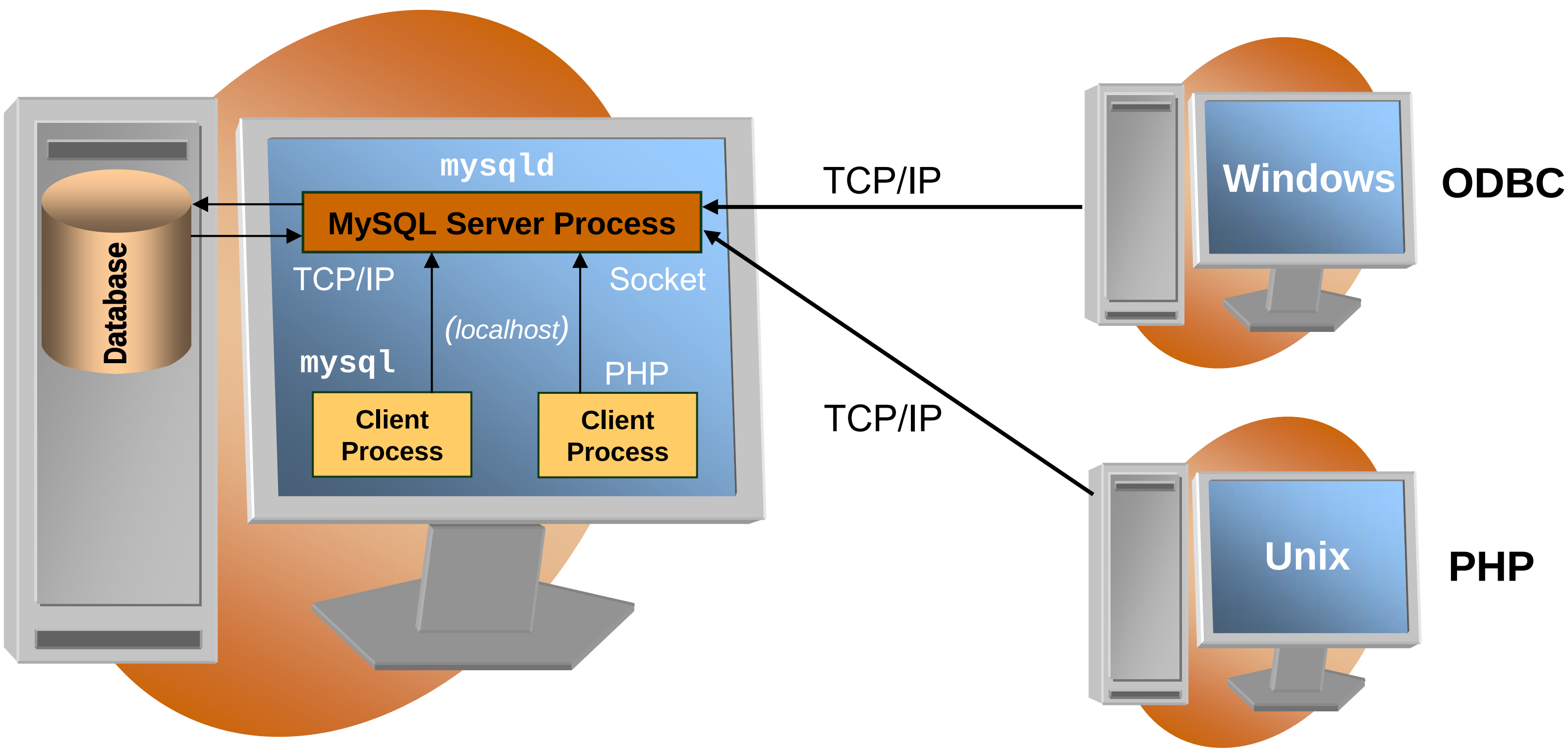
- `mysqld`
- Single process architecture
- Manages access to databases
- Multi-threaded connections
- Supports multiple storage engines
- Server vs. Host
  - *Server* is software
  - *Host* is physical machine which runs software

Client Programs

- Communicate with server to manipulate databases
- Common client programs
  - `mysql`
  - `mysqlimport`
  - `mysqldump`
  - `mysqladmin`
  - `mysqlcheck`

MySQL Client/Server Model

- Server -- the central database management program
- Client -- program(s) connect to the server to retrieve or modify data



Communication Protocols

- TCP/IP
  - Transmission Control Protocol
  - Internet Protocol
- Unix only
  - Unix socket
- Windows only
  - Shared Memory
  - Named Pipes

MySQL Non-Client Utilities

- Programs independent of the server
  - `myisamchk`
  - `myisampack`
- Access MyISAM tables directly

How MySQL Uses Disk Space

- Primarily for directories and files
- Server uses data directory
  - Database directory
  - Table format files (.frm)
  - Data and index files
  - InnoDB has its own tablespace and log files
  - Server log files and status files
  - Other database objects (i.e., triggers, views)
  - Authentication information
  - `mysqld`

How MySQL Uses Memory

- Primarily for data structures
  - Server sets up to manage clients and databases
- Server allocates memory for information
  - Thread handlers
  - MEMORY creates tables held in memory
  - Temporary tables
  - Buffers for each client connection
  - Global buffers and caches

Chapter Summary

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

Course Content

Learning Objectives

- Invoke client programs within the MySQL Client/Server architecture
- Use many of the features of the mysql client
- Describe the client interfaces provided by MySQL
- 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

Invoking Client Programs

- Can be invoked from command line
  - Windows console prompt
  - UNIX shell prompt
- Determine supported options

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

- Determine version

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

General Command Option Syntax

- Two general forms
  - Long options
  - Short options
- Examples:

```
shell> mysql --version

shell> mysql -V

shell> mysql --host=myhost.example.com

shell> mysql -h myhost.example.com

shell> mysql -hmyhost.example.com
```

Connection Parameter Options (1/3)

- Connect with host indicated
  - Locally to server running on same host
  - Remotely to server running on different host
- Client-specific options
- Common client options

`--protocol=`*`protocol_name`*

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

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

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

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

`--compress` or **`-C`**

Connection Parameter Options (2/3)

- User identification

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

  `--password=`*`pass_value`* or **`-p`*`pass_value`***

- Password options
  - Short form uses no space after option
  - Can omit password value to get prompt

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

Connection Parameter Options (3/3)

```
shell> mysql
shell> mysql --protocol=memory
shell> mysql --host=localhost --password --user=myname
shell> mysql -h localhost -p -u myname
shell> mysql --host=192.168.1.33 --user=myname
        --password=mypass
shell> mysql --host=localhost --compress
```

LAB

Using Option Files (1/3)

- Client programs look for files at startup
- Saves time and effort
- Options files organized into groups
  - Group-name line
  - Groups of groups (like client)
- Example file groups
  - [mysql]
  - [mysqldump]
  - [client]

SAMPLE FILE

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

[mysql]
safe-updates
```

Using Option Files (2/3)

- Option file location
  - Depends on Operating System
- Windows
  - `my.ini` and `my.cnf`
  - Windows install wizard places in `C:\Program Files\MySQL\<version_name>`
- Unix/Linux
  - `my.cnf` global option file
  - Global option file location `/etc/my.cnf`
- Multiple option files
- Need write permission to edit option files

Using Option Files (3/3)

- Specify single option file
  - Must be first option on command line

```
--defaults-file=file_name
--defaults-extra-file=file_name
--no-defaults
```

- Example

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

- Reference other option files

```
!include file_name
!include dir_name
```

- Last value takes precedence
- Command line options override config file

Using `mysql` Interactively

- **`mysql`** enables server queries
  - Interactive
  - Batch Mode
- Execute within the **`mysql`** client

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

- Execute from the command line

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

Statement Terminators (1/3)

- Several terminators can be used within **mysql**
  - Semi-colon (**;**)
  - Ego (**\g**)
  - Go (**\G**)
- Examples

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

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

Statement Terminators (2/3)

- Multi-line statements
  - Terminator required at end
  - Prompt changes from to **->**

```
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 |
+--------------------+------------+
```

Statement Terminators (3/3)

- Abort a statement
  - Use the **`\c`** terminator
- Exit a **`mysql`** server session
  - Use the **`\q`** terminator, or **`QUIT`**, or **`EXIT`**
- Examples

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

The MySQL Prompts (1/2)

- Several prompts

Prompt	Meaning of prompt
`mysql>`	Ready for new statement
`->`	Waiting for next line of statement (i.e., `';'` or `\G`)
`'>`	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 Prompts (2/2)

- Redefine the prompt

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

- Adding information within prompt

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

- Return to original prompt

```
prompt
mysql>
```

Add a trailing space at the end of the prompt by adding a `\_`:

```
prompt (\u@\h) [\d]\>\_
```

Using Editing Keys in MySQL

- Supports input-line editing
- Supports Unix/Linux tab completion
- Keyboard editing
  - Arrow keys
  - Full readline capabilities (Unix/Linux)
  - Command history (Unix/Linux)

Using Script Files with MySQL

- Input file containing SQL statements
  - “Script File” or “Batch File”
  - Plain text
  - Each statement must have a terminator
- Source file
  - No quotes required around filename or semi-colon at end
  - Example

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

MySQL Output Formats

- Two formats
  - Invoked Interactively
  - Invoked in Batch Mode
- Override the default

  `--batch` or **`-B`**

  `--table` or **`–t`**
- Suppress conversion

  `--raw` or **`-r`**
- Select output format

  `--html` or **`–H`**

  `--xml` or **`-X`**

Client Commands and SQL Statement

- Program send statement to MySQL server
  - **SELECT**
  - **INSERT**
  - **UPDATE**
  - **DELETE**
- Display information about the server connection

```
STATUS
--------------
mysql  Ver 14.14 Distrib 5.1.30, for Win32 (ia32)
Connection id:          2
Current database:       world
Current user:           root@localhost
...
```

- TEE file

```
tee tee_file.txt
```

```
shell> mysql -u user_name -ppassword --tee tee_file.txt
```

Using Server-Side Help (1/2)

- Reference manual lookups in `mysql`
- Specific topics
- Examples

```
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
   ...
```

Using Server-Side Help (2/2)

- Examples *(continued)*

```
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
   ...

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']
...
```

Using the Safe Updates Option

- Option:

  `--safe-updates`

- Useful for MySQL beginners
- Protects users from issuing potentially dangerous statements
  - `UPDATE` and `DELETE` only allowed with `WHERE` or `LIMIT`
  - `SELECT` output restricted
- Synonymous option:

  `--i-am-a-dummy`

MySQL Connectors

- Application Programming Interfaces (API's)
- Drivers
- Connectors available for Windows and Unix
- Officially supported connectors
  - Native “C”
  - MySQL Connector/ODBC
  - MySQL Connector/J (Java)
  - MySQL Connector/NET (.Net)
  - MySQL Connector/MXJ

- Available for download from MySQL website

Third-Party API's

- Several available
- API must be formatted for specific server version
- Based on “C” client library
- **`mysql`** for API's
  - PHP (**`mysqli`**) -- Ruby
  - Perl -- Tcl
  - ODBC -- Eiffel
  - C/C++ --Pascal
  - Python

Chapter Summary

- Invoke client programs within the MySQL Client/Server architecture
- Use many of the features of the mysql client
- Describe the client interfaces provided by MySQL
- 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

Course Content

DEVELOPER I

1. INTRODUCTION
2. MySQL CLIENT/SERVER CONCEPTS
3. MySQL CLIENTS
4. QUERYING FOR TABLE DATA
5. HANDLING ERRORS AND WARNINGS
6. DATA TYPES
7. SQL EXPRESSIONS
8. OBTAINING METADATA
9. DATABASES
10. TABLES
11. MANIPULATING TABLE DATA
12. TRANSACTIONS

DEVELOPER II

13. JOINS
14. SUBQUERIES
15. VIEWS
16. PREPARED STATEMENTS
17. EXPORTING AND IMPORTING DATA
18. STORED ROUTINES
19. TRIGGERS
20. STORAGE ENGINES
21. OPTIMIZATION
22. CONCLUSION

Learning Objectives

- Execute table data queries using the SELECT statement
- Aggregating queries for table data
- Use the UNION keyword to concatenate results from multiple SELECT statements

The SELECT Statement (1/2)

- Most commonly used command for queries
- Retrieves rows from tables in a database
- General syntax

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

The SELECT Statement (2/2)

- Examples

```
SELECT Name FROM Country;
+-------------------------------------+
| Name                                |
+-------------------------------------+
| Afghanistan                         |
| Netherlands                         |
:                                     :
| French Southern Territories         |
| Unites States Minor Outlying Islands |
+-------------------------------------+
239 rows in set (#.## sec)

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

Basic Uses of SELECT

- Clauses used to yield specific results
  - DISTINCT
  - FROM
  - WHERE
  - ORDER BY
  - LIMIT
- Syntax example:

```
SELECT DISTINCT values_to_display
FROM table_name
WHERE expression
ORDER BY how_to_sort
LIMIT row_count;
```

SELECT Tips

- **Commands (and clauses) are not case-sensitive (unless host is set as such)**
- **Use \c to abort a command**
- **Use \G in place of the ;) to return results by the row**
- **Use of * (all row data) can give random results and waste resources**
- **Keep clauses in proper order of precedence**

FROM Clause

- Optional element of the **`SELECT`** statement
  - May appear immediately after the expression list that appears after the **`SELECT`** keyword
  - Specifies a table which is processed by the statement
  - Denotes a table from which data can be retrieved
- Other uses of **`FROM`**
  - Other types of SQL statements for the addition, removal or modification of data process the `FROM` clause in their respective way

Using the FROM Clause

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

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

- Qualified table name
  - Prepending the name of database wherein the table resides is referred to as qualifying the table name
  - Accomplished by separating the database name and the table name with a dot (period):

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

- Table alias
  - Within a SQL statement, a table reference in the `FROM` clause can be given a temporary name

SELECT/DISTINCT (1/2)

- Removes duplicate rows
- Example

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

--->

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

SELECT/DISTINCT (2/2)

- Treats NULL values as the same
- Example

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

--->

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

LAB

SELECT/WHERE (1/4)

- Filters out unwanted rows
- Specify column values
- Example

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

SELECT/WHERE (2/4)

- Operators used with WHERE
  - Arithmetic
  - Comparison
  - Logical
- Arithmetic
  - +, -, *, /, DIV, %
- Comparison
  - <, <=, =, <=>, <> *or* !=, >=, >
- Logical
  - AND, OR, XOR, NOT
- Additional Options
  - IN, BETWEEN, *etc.*

SELECT/WHERE (3/4)

- Example

```
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 (0.31 sec)
```

SELECT/WHERE (4/4)

- Example

```
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)
```

A Word About NULL's (1/2)

- Most operators evaluate to NULL if one of the operands evaluates to NULL
  - NULL is a value-expression for which it is stipulated that it is unknown to which other value it is equal to
  - Evaluation of an operator that has to operate on such a value leads to another unknown value
- Logical and relational operators
  - Evaluates to a boolean value; TRUE or FALSE
  - The 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

A Word About NULL's (2/2)

- **WHERE** clause
  - Discards those rows for which the condition does not hold TRUE
  - Condition may not of necessarily been FALSE for the rows that are discarded
    - The case that the condition evaluated could of returned a NULL

SELECT/ORDER BY (1/3)

- Will return output rows in a specific order
- Example

```
SELECT Name FROM Country ORDER BY Name;
+----------------------+
| Name                 |
+----------------------+
| Afghanistan          |
| Albania              |
| Algeria              |
| American Samoa       |
| Andorra              |
| Angola               |
| Anguilla             |
| Antarctica           |
| Antigua and Barbuda  |
...
```

SELECT/ORDER BY (2/3)

- Ascending order is default
- Specify order with ASC and DESC
- Example

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

SELECT/ORDER BY (3/3)

- Sort multiple columns simultaneously
- Example

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

SELECT/LIMIT (1/3)

- Specify number of rows output
- Example

```
SELECT Name FROM Country LIMIT 8;
+----------------------+
| name                 |
+----------------------+
| Afghanistan          |
| Netherlands          |
| Netherlands Antilles |
| Albania              |
| Algeria              |
| American Samoa       |
| Andorra              |
| Angola               |
+----------------------+
8 rows in set (#.## sec)
```

MySQL specific keyword.

SELECT/LIMIT (2/3)

- Specify skip rows
- Example

```
SELECT name, population FROM country LIMIT 20,8;
+------------------------+------------+
| name                   | population |
+------------------------+------------+
| Belgium                |   10239000 |
| Belize                 |     241000 |
| Benin                  |    6097000 |
| Bermuda                |      65000 |
| Bhutan                 |    2124000 |
| Bolivia                |    8329000 |
| Bosnia and Herzegovina |    3972000 |
| Botswana               |    1622000 |
+------------------------+------------+
8 rows in set (#.## sec)
```

SELECT/LIMIT (3/3)

- Use with ORDER BY for ordered output
- Examples

```
SELECT * FROM t ORDER BY id LIMIT 1;

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)
```

LAB

Why Use Aggregate Functions? (1/2)

- Summary functions
  - Perform summary operations on a set of values
- Returns single value based on group of values
  - Turn many rows into one value
- Only NON NULL

Aggregate Functions:	Definition:
`MIN()`	Find the smallest value
`MAX()`	Find the largest value
`SUM()`	Summarize numeric value totals
`AVG()`	Summarize numeric value averages
`STD()`	Returns the population 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

Why Use Aggregate Functions? (2/2)

- Examples

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

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

Grouping with SELECT/GROUP BY

- Use GROUP BY for sub-group
- Based on values on one + columns of rows
- Example

```
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)
```

GROUP BY with GROUP_CONCAT()

- Concatenated result from each group
- Example

```
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)
```

GROUP BY with WITH ROLLUP (1/2)

- Multiple levels of summary values
- Example

```
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 |
|               | 6078749450 |
+---------------+------------+
8 rows in set (0.53 sec)
```

GROUP BY with WITH ROLLUP (2/2)

- Super aggregate operation
- Example

```
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)
```

GROUP BY with HAVING

- Eliminates rows based on aggregate values
- Example

```
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)
```

Using UNION (1/3)

- Concatenates results from multiple SELECTs
- Syntax

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

- When to use UNION
  - Retrieve rows from multiple tables with similar data
  - Retrieve several sets of rows from same table

Using UNION (2/3)

- Example of combining three tables

Original tables…

```
CREATE TABLE list1
(subscriber  CHAR(60),
 email       CHAR(60));
```

```
CREATE TABLE list2
(name      CHAR(96),
 address   CHAR(128));
```

```
CREATE TABLE list3
(email       CHAR(50),
 real_name   CHAR(30));
```

then UNION…

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

Using UNION (3/3)

- ORDER BY and LIMIT used to sort UNION

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

- Can also be applied to individual SELECTs

```
(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);
```

Further Practice: Chapter 4

- Comprehensive exercises

Chapter Summary

- Execute table data queries using the SELECT statement
- Aggregating queries for table data
- Use the UNION keyword to concatenate results from multiple SELECT statements

Course Content

Learning Objectives

- 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

SQL Modes for Syntax Checking

- View current SQL mode value

```
SELECT @@global.sql_mode;
SELECT @@session.sql_mode;
SELECT @@sql_mode;
```

SESSION is default operating mode.

- Check SQL mode setting

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

Setting the SQL Mode (1/2)

- Start server with SQL mode option

```
--sql-mode="mode_value"
```

- Use SET

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

- Clear SQL mode

```
SET sql_mode=''
```

- Single SQL mode

```
SET sql_mode = ANSI_QUOTES;
```

- Multiple SQL modes

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

Setting the SQL Mode (2/2)

- Composite SQL modes

```
SET sql_mode='TRADITIONAL';
Query OK, 0 rows affected (#.## sec)

SELECT @@sql_mode\G
********************** 1. row **********************
@@sql_mode:
   STRICT_TRANS_TABLES,STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZER
   O_DATE,ERROR_FOR_DIVISION_BY_ZERO,TRADITIONAL,
   NO_AUTO_CREATE_USER
1 row in set (#.## sec)
```

SQL Mode Values

- Commonly used modes
  - ANSI
  - ONLY_FULL_GROUP_BY
  - ERROR_FOR_DIVISION_BY_ZERO
  - STRICT_TRANS_TABLES, STRICT_ALL_TABLES
  - NO_ZERO_DATE, NO_ZERO_IN_DATE
  - TRADITIONAL

Handling Missing or Invalid Data

- Validation of values and report errors
- MySQL more "forgiving" data handling
  - Negative number in an UNSIGNED column converts to zero (0)

Handling Missing Values (1/2)

- INSERT may not specify a value for every column
- Example

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

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

Handling Missing Values (2/2)

- When column contains DEFAULT
  - MySQL inserts default specified
  - If no value, **DEFAULT NULL** added
- Example

```
   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
```

- Handling varies for columns with no DEFAULT

Invalid Values in Non-Strict Mode (1/2)

- Performs type conversion based on column constraints
  - When inserting or updating column values
  - When specifying a default
- Adjusts invalid values to legal values
  - Generates warning messages
- Conversions MySQL performs
  - Conversion of out-of-range
  - String Truncation
  - Enumeration and SET Value
  - Conversion to data type default
  - Assignment of NULL to NOT NULL

Invalid Values in Non-Strict Mode (2/2)

- Using ALTER TABLE
- Converting to DATE or INT

String Value	Converted to DATE	Converted to INT
'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

Invalid Values in Strict Mode

- Input values invalid for many reasons
- Strict mode rejects out-of-range values
- STRICT_TRANS_TABLES
- STRICT_ALL_TABLES

Additional Input Data Restrictions

- Strict mode turns on input value restrictions
  - Division by zero
  - Zero dates
  - Examples

```
SET sql_mode = 'STRICT_ALL_TABLES,ERROR_FOR_DIVISION_BY_ZERO';
SET sql_mode = 'STRICT_ALL_TABLES,NO_ZERO_DATE,NO_ZERO_IN_DATE';
```

- TRADTIONAL mode

```
SET sql_mode = 'TRADITIONAL';
```

Interpreting Error Messages (1/2)

- MySQL produces diagnostic messages
  - Error or warning messages

- Errors for failed SQL statements

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

- Information string for multi-row statements

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

Interpreting Error Messages (2/2)

- Operating system-level error

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

- Error code number
  - MySQL Reference Manual (Error Codes and Messages)

The SHOW WARNINGS Statement (1/3)

- MySQL generates warnings for non-compliance
- Display warning description
- Example

```
CREATE TABLE integers (i INT UNSIGNED NOT NULL);
Query OK, 0 rows affected (#.## sec)

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

…then execute **SHOW WARNINGS** *to display warnings…*

The SHOW WARNINGS Statement (2/3)

- Example *(continued)*

```
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)
```

The SHOW WARNINGS Statement (3/3)

- With LIMIT clause

```
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)
```

- With COUNT function

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

Warning Levels (1/2)

- "Warning" levels of severity
  - Error
  - Warning
  - Note
- Example of error

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

SHOW WARNINGS;
+-------+------+------------------------------------+
| Level | Code | Message                            |
+-------+------+------------------------------------+
| Error | 1146 | Table 'test.no_such_table' doesn't exist |
+-------+------+------------------------------------+
1 row in set (#.## sec)
```

Warning Levels (2/2)

- Example of Note

```
DROP TABLE IF EXISTS no_such_table;
Query OK, 0 rows affected, 1 warning (#.## sec)

SHOW WARNINGS;
+-------+------+-------------------------------+
| Level | Code | Message                       |
+-------+------+-------------------------------+
| Note  | 1051 | Unknown table 'no_such_table' |
+-------+------+-------------------------------+
1 row in set (#.## sec)
```

- Suppress Note warnings

```
SET sql_notes = 0;
```

The SHOW ERRORS Statement

- Similar to SHOW WARNINGS
  - Displays only errors
- Higher severity
- Also supports LIMIT and COUNT

The perror Utility

- Command line utility
  - Included with MySQL distributions
- Operating system level errors
  - Information on error codes
- Example

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

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

Further Practice: Chapter 5

- Comprehensive exercises

Chapter Summary

- 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

Course Content

DEVELOPER I

1. INTRODUCTION
2. MySQL CLIENT/SERVER CONCEPTS
3. MySQL CLIENTS
4. QUERYING FOR TABLE DATA
5. HANDLING ERRORS AND WARNINGS
6. DATA TYPES
7. SQL EXPRESSIONS
8. OBTAINING METADATA
9. DATABASES
10. TABLES
11. MANIPULATING TABLE DATA
12. TRANSACTIONS

DEVELOPER II

13. JOINS
14. SUBQUERIES
15. VIEWS
16. PREPARED STATEMENTS
17. EXPORTING AND IMPORTING DATA
18. STORED ROUTINES
19. TRIGGERS
20. STORAGE ENGINES
21. OPTIMIZATION
22. CONCLUSION

Learning Objectives

- 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/NOT NULL

Data Type Overview

- Four major categories
  - Numeric
  - Character
  - Binary
  - Temporal
- ABC's of data types
  - **A**pt
  - **B**rief
  - **C**omplete

Creating Tables with Data Types

- Column declarations
- Example

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

Numeric Data Types

- Store numeric data
- Types
  - Integer
  - Floating-Point
  - Fixed-Point
  - BIT
- Precision and scale

Integer Types

- Whole numbers
- Types
  - TINYINT
  - SMALLINT
  - MEDIUMINT
  - INT
  - BIGINT
- Example
  - **World** database, **City** table, **Population** column

    ```
    Population INT(11)
    ```
  - Largest value output (uses 8, 11 allowed)

    ```
    10500000
    ```

Integer Syntax:

```
INT (M)
```

Integer Type Comparison

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

Floating-Point Types

- Used for approximate-value numbers
  - Integer, Fractional or both
- Types
  - FLOAT
  - DOUBLE
- May declare with precision and scale
- Example
  - **World** database, **Country** table, **GNP** entity

    ```
    GNP FLOAT(10,2)
    ```
  - Largest value output (uses 7, 10 allowed; 2 to right of decimal)

    ```
    8510700.00
    ```

Syntax:

```
FLOAT (M,D)
```

Float Type Comparison

Column Type	Storage	Range
`FLOAT`	4 bytes	-3.402823466E+38 to -1.175494351E-38, 0 and 1.175494351E-38 to 3.402823466E+38

`DOUBLE`
`REAL`
`DOUBLE PRECISION`
	8 bytes	-1.7976931348623157E+308 to -2.2250738585072014E-308,
		0 and
		2.2250738585072014E-308 to 1.7976931348623157E+308

Fixed-Point Types

- Exact-value numbers
  - Integer, Fractional or both
- Types
  - DECIMAL
  - NUMERIC
- Example
  - To represent currency values such as dollars and cents

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

  - Example value output

```
650.88
```

Syntax:

```
DECIMAL (M,D)
```

BIT Types

- Bit-field values
  - Column Width (M) is number of bits per value
  - 1 to 64 bits
- Example
  - Storing 4 and 20 bits

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

- Can assign values using numeric expressions

Syntax:

```
BIT(M)
```

Rule of thumb:
n=81 Byte

Character String Data Types

- Sequence of alphanumeric characters
- Used to store text or integer data
- Factors to consider when choosing type

Comparison Values	Type	Description
Text	**CHAR**	Fixed-length character string
	VARCHAR	Variable-length character string
	TEXT	Variable-length character string
Integer	**ENUM**	Enumeration consisting of a fixed set of legal values
	SET	Set consisting of a fixed set of legal values

Text Types (1/2)

- CHAR/VARCHAR
  - CHAR
  - VARCHAR

- Example
  - **World** database, **CountryLanguage** table, **Language** entity

    ```
    Language CHAR(30)
    ```
  - Largest value output (uses 25, 30 allowed)

    ```
    Southern Slavic Languages
    ```

Text Types (2/2)

- TEXT
  - TINYTEXT
  - TEXT
  - MEDIUMTEXT
  - LONGTEXT

Text Type Summary

Type	Storage Required	Maximum Length
CHAR(*M*)	*M* characters	255 characters
VARCHAR(*M*)	#characters plus 1 or 2 bytes	65,535 bytes (subject to limitations)
TINYTEXT	#characters + 1 byte	255 bytes
TEXT	#characters + 2 bytes	65,535 bytes
MEDIUMTEXT	#characters + 3 bytes	16,777,215 bytes
LONGTEXT	#characters + 4 bytes	4,294,967,295 bytes

Structured Character String Types

- ENUM
  - Enumeration
- Example

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

- SET
  - List of string values
- Example

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

Character Set and Collation Support (1/3)

- Character set is a named encoded character Repertoire
  - Governed by Rules of Collation
- Collation is a names collating sequence
  - Defines character sort order
- String characteristics

Character Set and Collation Support (2/3)

- MySQL offers several character sets
  - Proper choice can make a big performance impact
  - Use **SHOW CHARACTER SET** to view list

```
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 |
...
```

Character Set and Collation Support (3/3)

- A character set may have several collations
  - Use **SHOW COLLATION** to view available collations

```
SHOW COLLATION LIKE 'latin1%';
```

Collation	Charset	Id	Default	Compiled	Sortlen
latin1_german1_ci	latin1	5			0
latin1_swedish_ci	latin1	8	Yes	Yes	1
latin1_danish_ci	latin1	15			0
latin1_german2_ci	latin1	31		Yes	2
latin1_bin	latin1	47		Yes	1
latin1_general_ci	latin1	48			[illegible]
latin1_general_cs	latin1	49			[illegible]
latin1_spanish_ci	latin1	94			[illegible]

LAB

Binary String Data Types

- Sequence of bytes
  - Binary digits (Bits) grouped in eights (octets)
- Example binary uses
  - Compiled computer programs/applications
  - Image and sound files
- Binary types
  - BINARY
  - VARBINARY
  - TINYBLOB
  - BLOB
  - MEDIUMBLOB
  - LONGBLOB

Binary String Type Comparison

Type	Storage Required	Maximum Length
BINARY(*M*)	*M* bytes	255 bytes
VARBINARY(*M*)	M bytes plus 1 or 2	65,533 bytes (subject to limitations)
TINYBLOB	#bytes + 1	255 bytes
BLOB	#bytes + 2	65,535 bytes
MEDIUMBLOB	#bytes + 3	16,777,215 bytes
LONGBLOB	#bytes + 4	4,294,967,295 bytes

Temporal Data Types (1/2)

- TIME
  - HH:MM:SS > **`12:59:02`**
- YEAR
  - Two or Four digit > **`2006`**
- DATE
  - YYYY-MM-DD > **`2006-08-04`**
- DATETIME
  - YYYY-MM-DD HH:MM:SS > **`2006-08-04 12:59:02`**
- TIMESTAMP > **`2006-08-04 12:59:02`**

Temporal Data Types (2/2)

Type	Storage Required	Range
DATE	3 bytes	'1000-01-01' to '9999-12-31'
TIME	3 bytes	'-838:59:59' to '838:59:59'
DATETIME	8 bytes	'1000-01-01 00:00:00' to '9999-12-31 23:59:59'
TIMESTAMP	4 bytes	'1970-01-01 00:00:00' to mid-year 2037
YEAR	1 byte	1901 to 2155 (for YEAR(4)), 1970 to 2069 (for YEAR(2))

The Meaning of NULL

- NULL can set data types to allow missing values
- NULL can be an empty query result
- Conceptually has several meanings
  - "no value"
  - "unknown value"
  - "missing value"
  - "out of range"
  - "not applicable"
- Two categories
  - Unknown
  - Not Applicable

When to Use NULL

- During database design
- Cases where column information is not available
  - Determine whether NULL values can be allowed
- Can also be changed in an existing table
- Example from *world*
  - **Country** table contains countries with no value for this column

```
`LifeExpectancy` float(3,1) DEFAULT NULL
```

When *NOT* to Use NULL

- Primary keys
- Column that must have a value
- Example from *world*
  - The City table primary key

```
`ID` int(11) NOT NULL AUTO_INCREMENT
```

  - CountryLanguage table contains a column to indicate official language, with a default which forces a value

```
`IsOfficial` enum('T','F') NOT NULL DEFAULT 'F'
```

Chapter Summary

- 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/NOT NULL

Course Content

DEVELOPER I	DEVELOPER II
1. INTRODUCTION	13. JOINS
2. MySQL CLIENT/SERVER CONCEPTS	14. SUBQUERIES
3. MySQL CLIENTS	15. VIEWS
4. QUERYING FOR TABLE DATA	16. PREPARED STATEMENTS
5. HANDLING ERRORS AND WARNINGS	17. EXPORTING AND IMPORTING DATA
6. DATA TYPES	18. STORED ROUTINES
7. SQL EXPRESSIONS	19. TRIGGERS
8. OBTAINING METADATA	20. STORAGE ENGINES
9. DATABASES	21. OPTIMIZATION
10. TABLES	22. CONCLUSION
11. MANIPULATING TABLE DATA	
12. TRANSACTIONS	

Learning Objectives

- 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

Components of SQL Expressions (1/3)

- Expressions identify which rows to effect
- Terms of expressions
  - Numbers
  - Strings
  - Dates/Times
  - NULL values
  - Column references
  - Function calls

Components of SQL Expressions (2/3)

- Examples of expressions

```
SELECT Name, Population FROM Country;

SELECT 14, -312.82, 4.32E-03, 'I am a string';

SELECT CURDATE(), VERSION();
```

Components of SQL Expressions (3/3)

- Examples of expressions *(continued)*

```
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 |
...
```

Numeric Expressions (1/2)

- Literal values
  - Exact-value
  - Approximate-value
  - Numerical expressions with NULL usually return a null value
- Expressions with NULL will return NULL
- Results depend on literal values

```
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 |
+-----------------+-----------------------+
```

Numeric Expressions (2/2)

- Mixing numbers with strings

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

String Expressions (1/3)

- Literal strings are quoted
  - Single or double quotes
  - ANSI_QUOTES sql mode special
- Data types
- Comparison operations

Operator:	Definition:
`<`	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 ***<x AND Y>***	Indicate a range of numerical values

String Expressions (2/3)

- Function examples

```
SELECT CONCAT('abc','def',REPEAT('X',3));
+------------------------------------+
| CONCAT('abc','def',REPEAT('X',3)) |
+------------------------------------+
| abcdefXXX                          |
+------------------------------------+

SELECT 'abc' || 'def';
+-----------------+
| 'abc' || 'def' |
+-----------------+
|               0 |
+-----------------+
1 row in set, 2 warnings (#.## sec)
```

String Expressions (3/3)

- Function examples *(continued)*

```
SET sql_mode = 'PIPES_AS_CONCAT';
Query OK, 0 rows affected (#.## sec)

SELECT 'abc' || 'def';
+----------------+
| 'abc' || 'def' |
+----------------+
| abcdef         |
+----------------+
1 row in set (#.## sec)
```

Case Sensitivity in String Comparisons (1/2)

- More complex than numeric
- Associated with character Set and Collation
- Rules for string comparison

```
SELECT 'Hello' = 'hello';
+-------------------+
| 'Hello' = 'hello' |
+-------------------+
|                 1 |
+-------------------+

SELECT 'Müller' = 'Mueller';
+----------------------+
| 'Müller' = 'Mueller' |
+----------------------+
|                    0 |
+----------------------+
```

Case Sensitivity in String Comparisons (2/2)

- Changing collation

```
SET collation_connection = latin1_german2_ci;

SELECT 'Müller' = 'Mueller';
+----------------------+
| 'Müller' = 'Mueller' |
+----------------------+
|                    1 |
+----------------------+
```

Using LIKE for Pattern Matching (1/2)

- Comparisons based on similarity
- Use LIKE pattern-matching operator
  - Percent character '%'
  - Underscore character '_'
- NOT LIKE opposite comparison

Using LIKE for Pattern Matching (2/2)

- Examples (LIKE vs. NOT LIKE)

```
SELECT Name FROM Country
 WHERE Name LIKE 'United%';
+-------------------------------------+
| Name                                |
+-------------------------------------+
| United Arab Emirates                |
| United Kingdom                      |
| United States Minor Outlying Isl.   |
| United States                       |
+-------------------------------------+
4 rows in set (#.## sec)
```

```
SELECT Name FROM Country
 WHERE Name NOT LIKE 'United%';
+-----------------------------------+
| Name                              |
+-----------------------------------+
| Aruba                             |
...
| Zambia                            |
| Zimbabwe                          |
+-----------------------------------+
235 rows in set (#.## sec)
```

Regular expressions (1/7)

- Keywords: REGEXP or RLIKE

  `string-expr RLIKE regexp-string`

- Complex patterns that go beyond catch-all wildcard characters such as % and _ in LIKE patterns
- Can be used to test
  - URLs
  - IP addresses
  - E-mail addresses
  - Postal codes
  - Etc.

RLIKE pattern syntax (2/7)

- Literal text (matches as is)
- Period (`.` equivalent to underscore in `LIKE` patterns)
- Occurrence indicators (`*`, `?`, `+`, `{`*`m`*`,`*`n`*`}`, denote the number of occurrences)
- Choice (`|` match either one of two alternatives)
- Escaped special characters using the backslash (\)
- Anchors match special positions (^ and $ match start and end of input, `[[:>:]]` and `[[:<:]]` match word boundaries)
- Character classes (`[`*`spec`*`]` match against a collection)

RLIKE character class syntax (3/7)

- Literal text,
  - `'[abc]'` matches either `'a'` or `'b'` or `'c'`
- Negation
  - `'[^abc]'` matches everything but `'a'`, `'b'` or `'c'`
- Ranges:
  - `'[a-z]'` matches `'a'`, `'b'`, `'c'`,...`'z'`
- Named range:
  - `'[[:space:]]'` matches all whitespace characters

Regular Expression Examples (4/7)

- Simple (any match will do):

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

- Using anchors (only complete match):

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

Regular Expression Examples (5/7)

- Alternation (choice):

```
SELECT Name FROM City
 WHERE Name RLIKE ' Los | Las ';
+-------------------+
| Name              |
+-------------------+
| Las Heras         |
.....................
| East Los Angelos |
| Las Heras         |
+-------------------+
```

- Equivalent character class notation:

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

Regular Expression Examples (6/7)

- Argentinian postal codes
  - Letter (A..H; J..N; P..Z) for the province
    - `[A-HJ-NP-Z]`
  - Four digits for the municipality
    - `[0-9]{4}`
  - Optionally, 3 Letters for the side of the street block
    - `([A-Z]{3})?`

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

Regular Expression Examples (7/7)

- Requires 'double' escaping

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

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

- No escaping possible inside character class

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

Temporal Expressions (1/2)

- Date and time values
- Used primarily in comparison operations
  - Add or subtract Interval
- Functions
- Date components

Type:	Default Format:
`DATE`	YYYY-MM-DD
`TIME`	HH:MM:SS
`DATETIME`	YYYY-MM-DD HH:MI:SS
`TIMESTAMP`	YYYY-MM-DD HH:MI:SS
`YEAR`	YYYY

Temporal Expressions (2/2)

- Comparison operators
  - =, <>, <, BETWEEN, etc
- Interval arithmetic
  - INTERVAL keyword

```
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                     |
+--------------------------------+-------------------------
SELECT '2010-01-01' - INTERVAL 10 DAY;
+--------------------------------+
| '2010-01-01' - INTERVAL 10 DAY |
+--------------------------------+
| 2009-12-22                     |
+--------------------------------+
```

LAB

Functions in SQL Expressions

- Functions can be invoked within expressions
- Can return a value used in place of function call
- No space after parenthesis
  - Unless IGNORE_SPACE is set

```
SELECT PI ();
ERROR 1305 (42000): FUNCTION world.PI does not exist

SET sql_mode = 'IGNORE_SPACE';
Query OK, 0 rows affected (#.## sec)

SELECT PI ();
+----------+
| PI ()    |
+----------+
| 3.141593 |
+----------+
1 row in set (#.## sec)
```

Comparison Functions (1/2)

- Test relative values or membership value
- Functions
  - LEAST( ) returns the smallest value from a set
  - GREATEST( ) returns the largest value from a set
- Examples

```
SELECT LEAST(4,3,8,-1,5), LEAST('cdef','ab','ghi');
+-------------------+--------------------------+
| LEAST(4,3,8,-1,5) | LEAST('cdef','ab','ghi') |
+-------------------+--------------------------+
|                -1 | ab                       |
+-------------------+--------------------------+

SELECT GREATEST(4,3,8,-1,5),
 GREATEST('cdef','ab','ghi');
+----------------------+-----------------------------+
| GREATEST(4,3,8,-1,5) | GREATEST('cdef','ab','ghi') |
+----------------------+-----------------------------+
|                    8 | ghi                         |
+----------------------+-----------------------------+
```

Comparison Functions (2/2)

- INTERVAL()
  - This function takes only integer expressions as arguments
  - The value of the first argument is compared to the value of the subsequent arguments
    - The function will then return the index of the last argument that has a value that is equal to or less than the first argument:

```
SELECT INTERVAL(10, 1, 2, 4, 8, 16);
+-------------------------------+
| INTERVAL(10, 1, 2, 4, 8, 16) |
+-------------------------------+
|                             4 |
+-------------------------------+
1 row in set (#.## sec)
```

Flow Control Functions (1/7)

- Choose between different values based on the result of an expression
- IF() tests the expression
  - Examples

True

```
SELECT IF(1 > 0, 'YES', 'NO');
```

False

```
+------------------------+
| IF(1 > 0, 'YES', 'NO') |
+------------------------+
| YES                    |
+------------------------+
1 row in set (#.## sec)
```

Flow Control Functions (2/7)

- Examples *(continued)*

```
SELECT name FROM country
 ORDER BY IF(code='USA',1,2), name
 LIMIT 10;
+----------------------+
| name                 |
+----------------------+
| United States        |
| Afghanistan          |
| Albania              |
| Algeria              |
| American Samoa       |
| Andorra              |
| Angola               |
| Anguilla             |
| Antarctica           |
| Antigua and Barbuda  |
...
| Zimbabwe             |
+----------------------+
239 rows in set (#.## sec)
```

Flow Control Functions (3/7)

- Examples *(continued)*

```
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 |
+----------+--------+--------+--------+--------+--------+--------+
```

Flow Control Functions (4/7)

- Dealing with NULL's in the expression

```
SELECT IF(1 > NULL, 'yes','no');
+---------------------------+
| IF(1 > NULL, 'yes','no') |
+---------------------------+
| no                        |
+---------------------------+

SELECT IF(NULL = NULL, 'yes','no');
+------------------------------+
| IF(NULL = NULL, 'yes','no') |
+------------------------------+
| no                           |
+------------------------------+

SELECT IF(NULL <> NULL, 'yes','no');
+-------------------------------+
| IF(NULL <> NULL, 'yes','no') |
+-------------------------------+
| no                            |
+-------------------------------+
```

Flow Control Functions (5/7)

- CASE/WHEN provides branching flow control
- General syntax

```
CASE case_expr
  WHEN when_expr THEN result
  [WHEN when_expr THEN result] ...
  [ELSE result]
END
```

Flow Control Functions (6/7)

- Example

```
SELECT name FROM country
 ORDER BY
 CASE code
   WHEN 'USA' THEN 1
   WHEN 'CAN' THEN 2
   WHEN 'MEX' THEN 3
 ELSE 4 END, name;
+-----------------+
| name            |
+-----------------+
| United States   |
| Canada          |
| Mexico          |
| Afghanistan     |
| Albania         |
| Algeria         |
| American Samoa  |
...
| Zimbabwe        |
+-----------------+
239 rows in set (#.## sec)
```

Flow Control Functions (7/7)

- Second general syntax

```
CASE
  WHEN when_expr THEN result
  [WHEN when_expr THEN result] ...
  [ELSE result]
END
```

- Example

```
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)
```

Numeric Functions (1/5)

- Mathematical operations
- Common functions
  - TRUNCATE()
  - FLOOR()
  - CEILING()
  - ROUND()
  - ABS()
  - SIGN()
  - SIN(), COS(), TAN()

Numeric Functions (2/5)

- **ROUND** examples

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

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

Numeric Functions (3/5)

- FLOOR/CEILING examples

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

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

Numeric Functions (4/5)

- ABS/SIGN examples

```
SELECT ABS(-14.7), ABS(14.7);
+------------+-----------+
| ABS(-14.7) | ABS(14.7) |
+------------+-----------+
| 14.7       | 14.7      |
+------------+-----------+

SELECT SIGN(-14.7), SIGN(14.7), SIGN(0);
+-------------+------------+---------+
| SIGN(-14.7) | SIGN(14.7) | SIGN(0) |
+-------------+------------+---------+
|          -1 |          1 |       0 |
+-------------+------------+---------+
```

Numeric Functions (5/5)

- Trigonometric examples

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

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

String Functions (1/10)

- Perform operations on strings
- Calculate string lengths
- Find the occurrence of a string in another string
- Get a part of a string
- Combine strings to a new string
- Change the letter case of a string
- Trim or pad strings
- Etc….

String Functions (2/10)

- INSTR(), LOCATE() and POSITION()

```
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

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

String Functions (3/10)

- Perform operations on strings
- LENGTH()/CHAR_LENGTH() examples

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

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

String Functions (4/10)

- CONCAT() and CONCAT_WS() examples

```
SELECT CONCAT('See','spot','run');
+----------------------------+
| CONCAT('See','spot','run')  |
+----------------------------+
| Seespotrun                 |
+----------------------------+

SELECT CONCAT_WS(' ','See','spot','run');
+----------------------------------+
| CONCAT_WS(' ','See','spot','run') |
+----------------------------------+
| See spot run                     |
+----------------------------------+
```

String Functions (5/10)

- SUBSTRING() and SUBSTRING_INDEX()

```
SELECT SUBSTRING('Alice and Bob', 1, 5);
+------------------------------------+
| SUBSTRING('Alice and Bob', 1, 5) |
+------------------------------------+
| Alice                              |
+------------------------------------+

SELECT SUBSTRING_INDEX('Alice and Bob', 'and', 1);
+----------------------------------------------+
| SUBSTRING_INDEX('Alice and Bob', 'and', 1) |
+----------------------------------------------+
| Alice                                        |
+----------------------------------------------+

SELECT SUBSTRING_INDEX('Alice and Bob', 'and', -1);
+-----------------------------------------------+
| SUBSTRING_INDEX('Alice and Bob', 'and', -1) |
+-----------------------------------------------+
|  Bob                                          |
+-----------------------------------------------+
```

String Functions (6/10)

- LEFT() and RIGHT()

```
SELECT LEFT('Alice and Bob', 5);
+--------------------------+
| LEFT('Alice and Bob', 5) |
+--------------------------+
| Alice                    |
+--------------------------+

SELECT RIGHT('Alice and Bob', 3);
+---------------------------+
| RIGHT('Alice and Bob', 3) |
+---------------------------+
| Bob                       |
+---------------------------+
```

String Functions (7/10)

- LTRIM(), RTRIM(), and TRIM()

```
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>

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

String Functions (8/10)

- INSERT() and REPLACE()

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

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

String Functions (9/10)

- CHARSET()/COLLATE() example

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

- CONVERT() example

```
SELECT CONVERT(_latin1'Müller' USING utf8);
+-------------------------------------+
| CONVERT(_latin1'Müller' USING utf8) |
+-------------------------------------+
| Müller                              |
+-------------------------------------+
```

- CAST() example

```
SELECT CAST(_latin1'test' AS CHAR CHARACTER SET utf8);
+------------------------------------------------+
| CAST(_latin1'test' AS CHAR CHARACTER SET utf8) |
+------------------------------------------------+
| test                                           |
+------------------------------------------------+
```

String Functions (10/10)

- STRCMP() examples

```
SELECT STRCMP('abc','def'),
        STRCMP('def','def'),
        STRCMP('def','abc');
+---------------------+---------------------+---------------------+
| STRCMP('abc','def') | STRCMP('def','def') | STRCMP('def','abc') |
+---------------------+---------------------+---------------------+
|                  -1 |                   0 |                   1 |
+---------------------+---------------------+---------------------+
1 row in set (#.## sec)
```

Temporal Functions (1/5)

- Time, Date, Year
- Perform many operations
- Functions

Functions	Definition
`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)
`DAYOFMONTH()` *or* `DAY()`	*Day of the month* in integer format, per value indicated (can use **NOW()** as above)
`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.

Temporal Functions (2/5)

- View current date and time

```
SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2004-04-30 11:59:15 |
+---------------------+
1 row in set (#.## sec)
```

- View date format

```
SELECT GET_FORMAT(DATE, 'EUR');
+-------------------------+
| GET_FORMAT(DATE, 'EUR') |
+-------------------------+
| %d.%m.%Y                |
+-------------------------+
1 row in set (#.## sec)
```

Temporal Functions (3/5)

- Extracting parts of date/time examples

```
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 |
+--------------------+---------------------+--------------------------+

SELECT DAYOFYEAR('2010-04-15');
+-------------------------+
| DAYOFYEAR('2010-04-15') |
+-------------------------+
|                     105 |
+-------------------------+

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 |
+------------------+--------------------+--------------------+
```

Temporal Functions (4/5)

- Composite dates/times examples

```
SELECT MAKEDATE(2010,105);
+--------------------+
| MAKEDATE(2010,105) |
+--------------------+
| 2010-04-15         |
+--------------------+

SELECT MAKETIME(9,23,57);
+-------------------+
| MAKETIME(9,23,57) |
+-------------------+
| 09:23:57          |
+-------------------+
```

Temporal Functions (5/5)

- Current dates/times examples

```
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 |
+----------------+----------------+---------------------+
```

NULL-Related Functions (1/2)

- Specifically for use with NULL
- ISNULL()/IFNULL() examples

```
SELECT ISNULL(NULL), ISNULL(0), ISNULL(1);
+--------------+-----------+-----------+
| ISNULL(NULL) | ISNULL(0) | ISNULL(1) |
+--------------+-----------+-----------+
|            1 |         0 |         0 |
+--------------+-----------+-----------+

SELECT IFNULL (NULL,'a'), IFNULL(0,'b');
+------------------+---------------+
| IFNULL(NULL,'a') | IFNULL(0,'b') |
+------------------+---------------+
| a                | 0             |
+------------------+---------------+
```

NULL-Related Functions (2/2)

- CONCAT with NULL examples

```
SELECT CONCAT('a','b'), CONCAT('a',NULL,'b');
+-----------------+----------------------+
| CONCAT('a','b') | CONCAT('a',NULL,'b') |
+-----------------+----------------------+
| ab              | NULL                 |
+-----------------+----------------------+

SELECT CONCAT_WS('/','a','b'),
 CONCAT_WS('/','a', NULL,'b');
+------------------------+-----------------------------+
| CONCAT_WS('/','a','b') | CONCAT_WS('/','a',NULL,'b') |
+------------------------+-----------------------------+
| a/b                    | a/b
+------------------------+---------------------------
```

Comments in SQL Statements (1/2)

- MySQL supports three forms of syntax
  - `'#'`
  - `/*` or `/*!`
  - `--`
- Examples

```
/* this is a comment */

/*
  this
  is a
  comment,
  too
*/
```

Comments in SQL Statements (2/2)

- C-style comments
- Examples

```
CREATE TABLE t (i INT) /*! ENGINE = MEMORY */;

SHOW /*!50002 FULL */ TABLES;

CREATE TABLE `CountryLanguage` (
    ...
    ) ENGINE=MyISAM COMMENT 'Lists Languages Spoken'
```

Comments on Database Objects

- Table comments
  - Comments can be added to the **CREATE TABLE** statement with the **COMMENT** keyword

```
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: Chapter 7

- Comprehensive exercises

Chapter Summary

- 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

Course Content

DEVELOPER I

1. INTRODUCTION
2. MySQL CLIENT/SERVER CONCEPTS
3. MySQL CLIENTS
4. QUERYING FOR TABLE DATA
5. HANDLING ERRORS AND WARNINGS
6. DATA TYPES
7. SQL EXPRESSIONS
8. OBTAINING METADATA
9. DATABASES
10. TABLES
11. MANIPULATING TABLE DATA
12. TRANSACTIONS

DEVELOPER II

13. JOINS
14. SUBQUERIES
15. VIEWS
16. PREPARED STATEMENTS
17. EXPORTING AND IMPORTING DATA
18. STORED ROUTINES
19. TRIGGERS
20. STORAGE ENGINES
21. OPTIMIZATION
22. CONCLUSION

Learning Objectives

- 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

Metadata Access Methods

- Information about database structure is metadata
- Methods
  - INFORMATION_SCHEMA
  - SHOW
  - DESCRIBE
  - mysqlshow
- Metadata for several database aspects
- INFORMATION_SCHEMA was introduced in 5.0

INFORMATION_SCHEMA Database (1/2)

- Database/schema that serves as a central repository for metadata
- Virtual database
- Use **SELECT** to obtain information

INFORMATION_SCHEMA Database (2/2)

- Tables example

```
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                              |
...
```

INFORMATION_SCHEMA Tables (1/3)

- Table contents
  - CHARACTER_SETS -- available character sets
  - COLLATIONS -- collations for each character set
  - COLLATION_CHARACTER_SET_APPLICABILITY -- which character set applies to each collation
  - COLUMNS -- columns in tables
  - COLUMN_PRIVILEGES -- column privileges held by MySQL user accounts
  - ENGINES -- storage engines
  - EVENTS -- scheduled events
  - FILES -- the files in which MySQL NDB Disk Data tables are stored
  - KEY_COLUMN_USAGE -- constraints on key columns

INFORMATION_SCHEMA Tables (2/3)

- Table contents
  - PARTITIONS -- table partitions
  - PLUGINS -- server plugins
  - PROCESSLIST -- which threads are running
  - REFERENTIAL_CONSTRAINTS -- foreign keys
  - ROUTINES -- stored procedures and functions
  - SCHEMATA -- databases
  - SCHEMA_PRIVILEGES -- database privileges held by MySQL user accounts
  - STATISTICS -- table indexes
  - TABLES -- tables in databases
  - TABLE_CONSTRAINTS -- constraints on tables

INFORMATION_SCHEMA Tables (3/3)

- Table contents
  - TABLE_PRIVILEGES -- table privileges held by MySQL user accounts
  - TRIGGERS -- triggers in databases
  - USER_PRIVILEGES -- global privileges held by MySQL user accounts
  - VIEWS -- views in databases

Displaying INFORMATION_SCHEMA Tables (1/3)

- Specify table name

```
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    |
+-----------------+
```

Displaying INFORMATION_SCHEMA Tables (2/3)

- Can use all the normal **SELECT** features
  - Specify columns
  - Restrict rows with the WHERE clause
  - Group or Sort with GROUP BY and ORDER BY
  - Use joins, unions and subqueries
  - Can feed results in another table
  - Create views on top of INFORMATION_SCHEMA tables

Displaying INFORMATION_SCHEMA Tables (3/3)

- **SELECT** examples

```
SELECT TABLE_NAME, ENGINE FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'world';

SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME
FROM INFORMATION_SCHEMA.COLUMNS
WHERE DATA_TYPE = 'set';

SELECT CHARACTER_SET_NAME, COLLATION_NAME
FROM INFORMATION_SCHEMA.COLLATIONS
WHERE IS_DEFAULT = 'Yes';

SELECT TABLE_SCHEMA, COUNT(*)
FROM INFORMATION_SCHEMA.TABLES;
GROUP BY TABLE_SCHEMA;
```

SHOW Statements (1/8)

- MySQL supports many **SHOW** statements
- Commonly used statements
  - `SHOW DATABASES`
  - `SHOW [FULL] TABLES`
  - `SHOW [FULL] COLUMNS`
  - `SHOW INDEX`
  - `SHOW CHARACTER SET`
  - `SHOW COLLATION`

SHOW Statements (2/8)

- SHOW DATABASE example

```
SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| menagerie          |
| mysql              |
| test               |
| world              |
+--------------------+
```

SHOW Statements (3/8)

- SHOW TABLES examples

```
SHOW TABLES;
+-----------------+
| Tables_in_world |
+-----------------+
| City            |
| Country         |
| CountryLanguage |
+-----------------+
```

```
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)
```

SHOW Statements (4/8)

- SHOW COLUMNS example

```
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     |       |
+-------------+---------------+------+-----+---------+-------+
```

SHOW Statements (5/8)

- SHOW FULL COLUMNS example

```
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:
*************************** 2. row ***************************
Field: Language
Type: char(30)
Collation: latin1_swedish_ci
Null: NO
Key: PRI
...
```

SHOW Statements (6/8)

- SHOW with LIKE example

```
SHOW DATABASES LIKE 'm%';
+---------------+
| Database (m%) |
+---------------+
| menagerie     |
| mysql         |
+---------------+
```

- SHOW with WHERE example

```
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    |       |
+----------------+-------------+------+-----+---------+-------+
```

SHOW Statements (7/8)

- SHOW INDEX example

```
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:
```

SHOW Statements (8/8)

- SHOW CHARACTER SET/COLLATION examples

```
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 |
...

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 |
...
```

DESCRIBE Statements

LAB

- Equivalent to SHOW COLUMNS
- Can be abbreviated as DESC

```
DESCRIBE table_name;
DESC table_name;
SHOW COLUMNS FROM table_name;
```

- DESCRIBE does not support FROM
- Shows INFORMATION_SCHEMA table information

```
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       |       |
+----------------------+-------------+------+-----+---------+-------+
```

The mysqlshow Command (1/3)

- Client program
- Information about structure of databases and tables
  - Similar to **SHOW** Statements
- General syntax

```
mysqlshow [options] [db_name [table_name [column_name]]]
```

- Options can be standard connection parameters

The mysqlshow Command (2/3)

- Examples

```
shell> mysqlshow
+--------------------+
|     Databases      |
+--------------------+
| information_schema |
| menagerie          |
| mysql              |
| test               |
| world              |
+--------------------+
```

```
shell> mysqlshow world
Database: world
+-----------------+
|     Tables      |
+-----------------+
| City            |
| Country         |
| CountryLanguage |
+-----------------+
```

The mysqlshow Command (3/3)

- Examples *(continued)*

```
shell> mysqlshow world City

shell> mysqlshow world City CountryCode

shell> mysqlshow -u <user_name> -p "w%“
Enter Password: <password>
Wildcard: w%
+-----------+
| Databases |
+-----------+
| world     |
+-----------+
```

Chapter Summary

- 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

Course Content

Learning Objectives

- 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

Database Properties (1/2)

- MySQL manages data
  - Performing storage
  - Retrieval
  - Manipulation
- Database Directory
  - Physical data directory
  - Same name as database
  - Manages tables
  - Default character set and collation
  - Databases cannot be nested

Database Properties (2/2)

- Objects belonging to a database
  - Table data and record of relationships
  - Stored procedures
  - Triggers
  - Views
  - Events

MySQL places NO limits on the number of Databases or tables.
It is only limited by your filesystem.

~

Another word for "Database" is "Schema".

Good Design Practices

- Important things to consider in design
  - Information to be stored?
  - Types of queries required
  - Keep your business rules in mind
- MySQL good design techniques
  - Keys
  - Normalization
  - Modeling

Keys (1/2)

- Row identification
- Superkey
- Candidate key
- Primary key
- Foreign key
- Several benefits to using keys
  - Decrease lookup time
  - Enforce uniqueness identification of each row
  - A primary key cannot contain a NULL

Keys (2/2)

Electronics Equipment Rental Database

Customer Table

Primary Key

cust_id	firstname	lastname
1	Abby	Thompson
2	Jack	Guerra
3	Thorild	Svenson

Foreign Key

Account Table

acct_id	equip_id	cust_id	start_date
100	TV1	1	01/22/2000
101	ST1	1	06/07/2000
102	ENT1	2	12/26/2004
103	TV1	2	11/09/2004
104	ST2	2	11/09/2004
105	ENT1	3	03/31/1998

Foreign Key

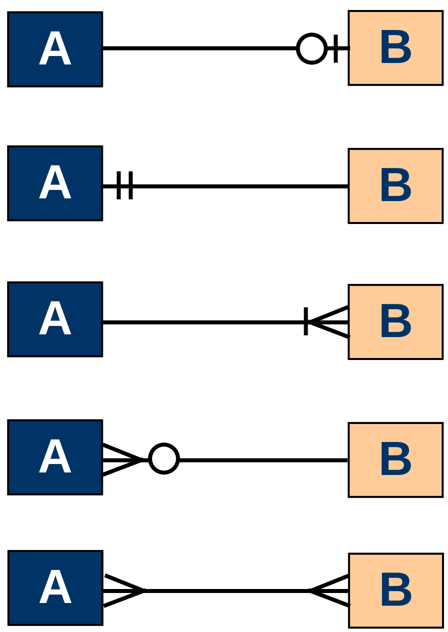
Equipment Table

equip_id	name	category
TV1	Television_36	Television
ST1	Stereo_100W	Stereo
ENT1	Oak_Entrtnmnt	Entrtnmnt_Ctr

Primary Key

Common Diagramming Systems (1/2)

- ERD
  - Entity relationship diagram
  - High-level conceptual model
  - Relationship notation



1-to-0 (or 1)

1-to-1

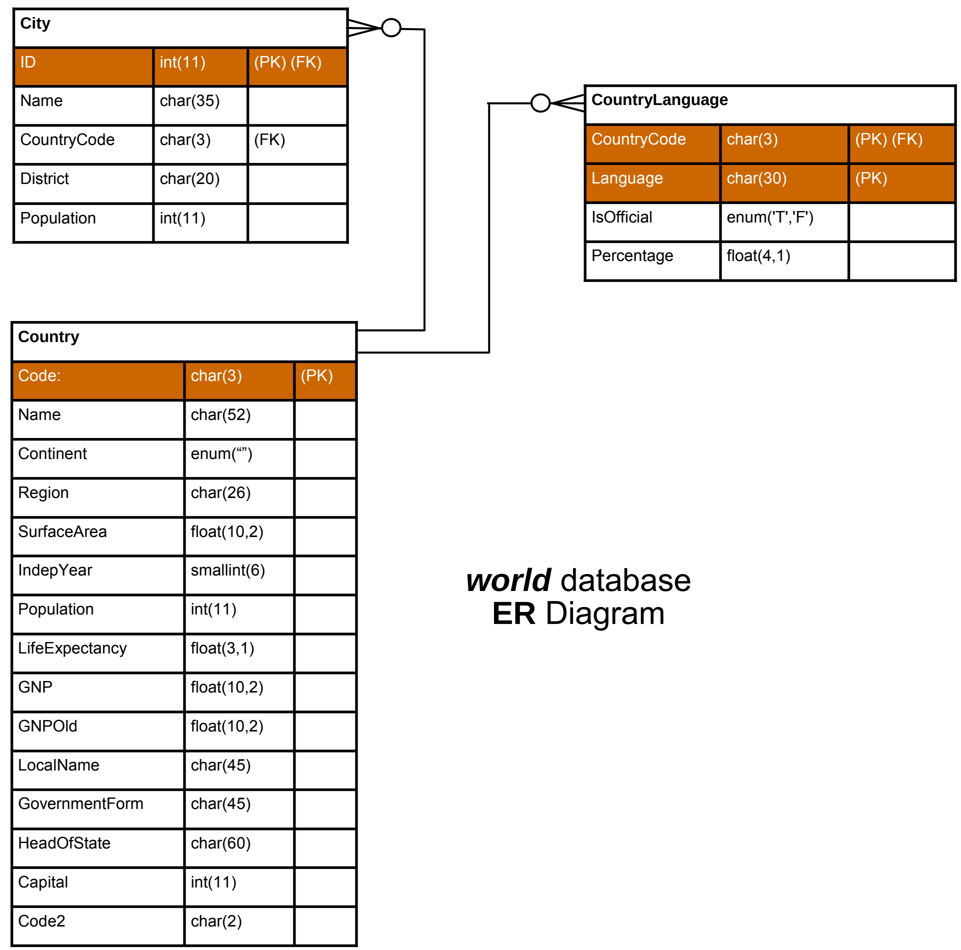
1-to-1 (or many)

0, 1 (or many)-to-1

Many-to-Many

Common Diagramming Systems (2/2)

- Example ERD
  - **world** database



world database
ER Diagram

Normalization

- Widely used as a guide in designing relational databases
- Why normalize?
  - Eliminate redundant data
  - Eliminate columns not dependent on key
  - Isolate independent multiple relationships

Advantages of Normalizing

- ER diagrams
  - Entity relationship diagrams assist in database planning
  - More effective with a normalized database
- Compact
  - Easier to modify a single object property
- Joins
  - Improved table relationships reduce data to be searched
- Optimizer
  - Retrievals and updates are more efficient due to better joins
- Updates
  - Easier to update single versus duplicate data locations

Disadvantages of Normalizing

- Numerous tables
- Maintenance
- Slower read requests with joins

Example of 'world' Normalization (1/2)

- Why are languages *not* added to Country table?
  - Large discrepancies between numbers of languages per country

(Turkey ***versus*** **Canada)**

```
+-------------+----------+
| CountryCode | Language |
+-------------+----------+
| TUR         | Arabic   |
| TUR         | Kurdish  |
| TUR         | Turkish  |
+-------------+----------+
3 rows in set (#.## sec)
```

```
+-------------+-------------------+
| 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)
```

Example of 'world' Normalization (2/2)

- Adding language to Country would result in many empty spaces
- Extra details for languages would be extraneous

```
+-------------+----------+------------+------------+
| CountryCode | Language | IsOfficial | Percentage |
+-------------+----------+------------+------------+
| TUR         | Arabic   | F          |        1.4 |
| TUR         | Kurdish  | F          |       10.6 |
| TUR         | Turkish  | T          |       87.6 |
+-------------+----------+------------+------------+
3 rows in set (#.## sec)
```

- Meets goal of normalization

Normal Forms (1/2)

- First three are most common
  - First Normal Form (1NF) -- contains no repeating groups within rows
  - Second Normal Form (2NF) -- normalized at the first level and every non-key (supporting) value is dependent on the primary key value
  - Third Normal Form (3NF) -- normalized at the first and second level, dependent *solely* on the primary key and no other non-key (supporting) value

Normal Forms (2/2)

- 1 to many relationships
  - **world** database contains some examples
    - *1* Country *to many* Languages
    - *1* Country *to many* Cities

Identifier Syntax

- Identifiers are names
  - Alias
  - Database
  - Column
  - Index
- May be quoted or unquoted
- Unquoted identifier rules

Reserved Words as Identifiers (1/2)

- Special words
  - e.g. Function names cannot be identifiers
- Error when using reserved word

```
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

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
```

Reserved Words as Identifiers (2/2)

- Proper use of quotes to avoid errors
  - Backtick quotes
  - ANSI quotes

```
CREATE TABLE t (`order` INT NOT NULL UNIQUE, d DATE NOT NULL);
Query K, 0 rows affected (#.## sec)
```

or ...

```
CREATE TABLE t (“order” INT NOT NULL UNIQUE, d DATE NOT NULL);
Query K, 0 rows affected (#.## sec)
```

- Reserved word as an alias

```
SELECT 1 AS 'INTEGER';
SELECT 1 AS “INTEGER”;
SELECT 1 AS `INTEGER`;
```

Using Qualified Names

- Identifiers in qualified form
  - Database together with table
  - Table together with column
- Examples

```
SELECT * FROM world.Country;
SELECT Country.Name FROM Country;
SELECT world.Country.Name FROM world.Country;
```

Case Sensitivity

- Affects the use of identifiers
  - Some are case sensitive some are not
- Rules to determine case sensitivity
  - Databases and table identifiers
    - O/S and filesystem dependent
    - lower_case_table_names
    - Must remain consistent throughout a given statement
  - Column, index, and stored routine identifiers
    - Not case-sensitive
  - Column aliases
    - Not case-sensitive
  - Triggers
    - O/S and filesystem dependent

Creating Databases (1/2)

- CREATE DATABASE statement
- Examples

```
CREATE DATABASE mydb;

CREATE DATABASE IF NOT EXIST mydb;
```

- Optional clauses
  - CHARACTER SET (column setting)
  - COLLATE
  - Example

```
CREATE DATABASE mydb CHARACTER SET utf8 COLLATE utf8_danish_ci;
```

Creating Databases (2/2)

- Using a database in **mysql**

```
USE mydb;
```

- Displaying a database creation

```
SHOW CREATE DATABASE world\G
*********************** 1. row **********************
       Database: world
Create Database: CREATE DATABASE `world`
          /*!40100 DEFAULT CHARACTER SET latin1 */
```

LAB

Altering Databases

- ALTER DATABASE statement
- Examples

```
ALTER DATABASE mydb COLLATE utf8_polish_ci;

ALTER DATABASE mydb CHARACTER SET latin1 COLLATE
latin1_swedish_ci;
```

- Affects new tables only

Dropping Databases

- DROP DATABASE statement
- Examples

```
DROP DATABASE mydb;

DROP DATABASE IF EXISTS mydb;
```

- Full or empty databases dropped

DROP DATABASE has no UNDO feature, so be cautious when deleting an entire database!

Further Practice: Chapter 9

- Comprehensive Exercises

Chapter Summary

- 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

Course Content

Learning Objectives

- 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

Creating a Table

- General syntax for creating a table

```
CREATE TABLE <table> (
<column name> <column type> [<column options>],
[<column name> <column type> [<column options>],…,]
[<index list>]
)[<table options>];
```

- Example

```
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';
```

Table Properties

- Add table options to CREATE TABLE statement
- Several options available
  - ENGINE
  - COMMENT
  - CHARACTER SET
  - COLLATE

COLLATE can also be used in SELECT queries.

- Example

```
CREATE TABLE CountryLanguage (
...
) ENGINE=MyISAM COMMENT='Lists Language Spoken' CHARSET
  utf8 COLLATE utf8_unicode_ci;
```

Column Options (1/2)

- Add column options to CREATE TABLE statement
- Several options available
  - NULL
  - NOT NULL
  - DEFAULT
  - AUTO_INCREMENT
- Constraints
  - Restrictions placed on one or more columns
  - Primary Key
  - Foreign Key
  - Unique

Column Options (2/2)

- Column options example

```
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 CHARSET=latin1
```

SHOW CREATE TABLE

- Viewing the exact statement used to create a table

- Example

```
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)
```

Creating Tables from Existing Tables (1/4)

- Two methods
  - `CREATE TABLE…SELECT`
  - `CREATE TABLE…LIKE`
- **`CREATE TABLE…SELECT`** will create a new table to fit and store the result set returned by the **`SELECT`**
- **`CREATE TABLE LIKE`** creates a structurally equivalent table (alas no foreign keys), but does not copy any data

Creating Tables from Existing Tables (2/4)

- **CREATE TABLE...AS SELECT** can create a table that is empty or non-empty, depending on what is returned by the **SELECT** part

```
CREATE TABLE CityCopy1 AS SELECT * FROM City;
```

- Create a table that contains a selection of the contents:

```
CREATE TABLE CityCopy2 AS SELECT * FROM City
 WHERE Population > 2000000;
```

- Create an empty copy of an existing table:

```
CREATE TABLE CityCopy3 AS SELECT * FROM City LIMIT 0;
```

- Create a table that contains only specific columns:

```
CREATE TABLE CityCopy4 AS SELECT col1,col2 FROM City;
```

Creating Tables from Existing Tables (3/4)

- LIKE examples

```
CREATE TABLE t
 (i INT NOT NULL AUTO_INCREMENT,
 PRIMARY KEY (i))
 ENGINE = InnoDB;

CREATE TABLE copy1 SELECT * FROM t WHERE 0;

CREATE TABLE copy2 LIKE t;
```

Creating Tables from Existing Tables (4/4)

- LIKE examples *(continued)*

```
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

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
```

- Some attributes (foreign keys, datadir options, etc.) not copied with **LIKE**

Temporary Tables

- For temporary use
- Only visible from within the current session
- Exist only during the current session
- May use an existing non-temporary table's name
  - Non-temporary table will be 'masked' while the temporary table still exists
- Not visible through either **SHOW TABLE** or information_schema.TABLES
- Temporary table containing cities associated with Texas:

```
CREATE TEMPORARY TABLE Texas AS
 SELECT Name FROM City WHERE District='Texas';
```

Add a Column

- Use an ALTER TABLE statement *with* ADD
- Example

```
ALTER TABLE City ADD COLUMN LocalName VARCHAR(35) CHARACTER SET utf8
 NOT NULL DEFAULT '' COMMENT 'The local name of this City';
```

- Structure Change

```
DESCRIBE City;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| ID         | int(11)     | NO   | PRI | NULL    | auto_increment |
| Name       | char(35)    | NO   |     |         |                |
...
| Population | int(11)     | NO   |     | 0       |                |
| LocalName  | varchar(35) | NO   |     |         |                |
+------------+-------------+------+-----+---------+----------------+
```

Remove a Column

- Use an ALTER TABLE statement *with* DROP
- Example

```
ALTER TABLE City DROP COLUMN LocalName;
```

Modifying Columns

- Use an ALTER TABLE statement *with* MODIFY
- Example

```
ALTER TABLE City MODIFY ID BIGINT NOT NULL AUTO_INCREMENT;
```

Changing Columns

- Use an ALTER TABLE statement *with* CHANGE
- Example
  - To change LastName column from CHAR(30) to CHAR(40)

```
ALTER TABLE HeadOfState CHANGE LastName LastName CHAR(40) NOT NULL;
```

  - To change name to Surname as well

```
ALTER TABLE HeadOfState CHANGE LastName Surname CHAR(40) NOT NULL;
```

Renaming Tables

- Use an ALTER TABLE statement *with* RENAME
- Examples

```
ALTER TABLE t1 RENAME TO t2;
RENAME TABLE t1 TO t2;
RENAME TABLE t1 TO tmp, t2 TO t1, tmp TO t2;
```

The DROP TABLE Command

- Remove a table
- Full or empty table
- **IF EXISTS** to avoid error
- **DROP TEMPORARY TABLE**
- Examples:

```
DROP TABLE table1;
DROP TABLE IF EXISTS table1;
DROP TEMPORARY TABLE EU_Countries_TEMP;
```

DROP TABLE has no UNDO feature, so be cautious when deleting an entire table!

LAB

Foreign Keys

- Distinct concepts
  - Foreign keys
    - References between rows throughout the databases
  - Relationships
    - Foreign keys are used to implement relationships between rows of data
  - Foreign key constraints
    - Used to maintain foreign keys and to ensure the references are kept consistent
  - Referential integrity
    - Foreign key constraints are used to enforce referential integrity

Foreign Keys and Relationships

- The ***world*** database
  - A row in the Country table represents a real county
  - A row in the City table represents a real city
- Real world relationship
  - Countries contain cities, and countries and cities are related to one another through this containment relationship
  - Of all the cities that belong to a country, there is one 'special' city known as the capital
    - City A resides in a particular country B
    - Country B can have a capital C which is also a city in that country
    - It is likely that city A is different from city C
  - The result is two entirely independent relationships between cities and countries

Foreign Keys Represent Relationships

- The data must represent the real world
  - 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
- Real world has a process to represent relationship
  - For the most part, all cities that have a border that is enclosed within the border of a country are cities that belong to that country
- Databases have a process to represent relationship
  - A foreign key is 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
  - A symbolic relationship using column values are used as a reference to the related row

Foreign Keys in the World DB

- The City table has a CountryCode column
  - Foreign key that relates to the Code column in the Country table
  - The Code column from the Country table will connect to 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
- The Country table contains a Capital column
  - 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

Foreign Key Example (1/2)

- Particular row from the City table

```
SELECT ID, Name, CountryCode FROM City WHERE Name LIKE 'Helsinki %';
+------+------------------------+-------------+
| ID   | Name                   | CountryCode |
+------+------------------------+-------------+
| 3236 | Helsinki [Helsingfors] | FIN         |
+------+------------------------+-------------+
```

- What country is associated with the CountryCode 'FIN'?

```
SELECT Code, Name, Capital FROM Country WHERE Code = 'FIN';
+------+---------+---------+
| Code | Name    | Capital |
+------+---------+---------+
| FIN  | Finland |    3236 |
+------+---------+---------+
```

- In which country is the city called 'Helsinki' situated?

```
SELECT ID, Name, CountryCode FROM City WHERE ID = 3236;
+------+------------------------+-------------+
| ID   | Name                   | CountryCode |
+------+------------------------+-------------+
| 3236 | Helsinki [Helsingfors] | FIN         |
+------+------------------------+-------------+
```

Foreign Key Example (2/2)

- The following diagram illustrates these relationships and the foreign keys that implement them:

City

Column Name	Value
ID	3236
Name	Helsinki [Helsingfors]
CountryCode	FIN

Country

Column Name	Value
Code	FIN
Name	Finland
Capital	3236

Referential Integrity

- What would happen if the country code would change for a particular country?
  - Suppose the country code for Romania was changed from 'ROM' to 'ROU'

```
UPDATE Country SET Code = 'ROU' WHERE Code = 'ROM';
query OK, 1 row affected (#.## sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

  - Corresponding codes in City table unchanged
    - Those cities with a country code of ROM are no longer connected to a country
    - The referential integrity has been compromised
  - According to the data, the country of Romania has no cities
    - The database itself does not enforce referential integrity

Foreign Key Constraints

- 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
  - Changes to referenced rows can either be prevented or propagated to the referencing rows
    - A foreign key constraint can be defined in such a way that any change that will cause referential integrity problems are prevented from committing
    - A foreign key constraint can be defined to automatically propagate (cascade) any changes throughout the data to maintain referential integrity

Creating Foreign Key Constraints (1/3)

- Foreign keys constraints may be specified as part of the CREATE TABLE syntax

```
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

```
ALTER TABLE City ADD FOREIGN KEY (CountryCode)
 REFERENCES Country (Code)
```

Creating Foreign Key Constraints (2/3)

- Full Foreign Key syntax

```
[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}]
```

- Mandatory elements
  - A list of referencing columns
  - The name of the referenced table
  - A list of referenced columns
  - The referenced columns should together form a PRIMARY KEY constraint

Creating Foreign Key Constraints (3/3)

- Optional elements
  - The constraint name
  - 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
    - **NO ACTION** means that a **DELETE** of a row from the referenced table must not occur if there are still referencing rows
    - **RESTRICT** means the same as **NO ACTION**
    - **SET NULL** means that the referencing columns in the referencing rows are changed to **NULL**
  - UPDATE rule - specifies what should happen to the referencing rows in case a referenced row is changed
    - Uses similar rules as those used for DELETE

Foreign Keys & Storage Engines (1/3)

- 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

```
ALTER TABLE City ENGINE = InnoDB;
```

  - When attempting to create a foreign key on a non-InnoDB table, MySQL will silently ignore the request
    - Not even a a warning will be issued
  - When attempting to create a foreign key that references a non-InnoDB table, a runtime error occurs

```
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

Foreign Keys & Storage Engines (2/3)

- InnoDB engine status

```
SHOW ENGINE InnoDB STATUS;
...
-----------------LATEST FOREIGN KEY ERROR------------------
... 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)
...
```

  - InnoDB is looking for a table name near the occurrence of (Code) in the DDL statement, but can't seem to find one
    - The Country table is not a InnoDB table
    - InnoDB doesn't know anything about the existence of any non-InnoDB tables
    - The storage engine of the Country table would have to be changed to InnoDB before a foreign key constraint can be created

Foreign Keys & Storage Engines (3/3)

- InnoDB implementation of foreign keys
  - 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
  - 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
  - MySQL accepts the syntax for 'inline' foreign key constraints (foreign key constraint definitions at the column level)
    - However, they are silently discarded

Further Practice: Chapter 10

- Comprehensive exercises

Chapter Summary

- 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

Course Content

Learning Objectives

- 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

The INSERT Statement

- The INSERT statement is a common method for adding new rows of data into a table

```
INSERT INTO table_name (column_list) VALUES(row_list);
```

  - *table_name* identifies the table to which new rows are to be added
  - *column_list* is optional – if present;
    - Comma-separated list of column names from the specified table, enclosed in parentheses
    - Order of the columns dictates the order of the values as they appear in the rows that are to be added to the table
  - *row_list* (row constructors) 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
    - There must be at least one row constructor, but multiple rows may be specified provided they are separated by a comma

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

INSERT ... SET

- The INSERT ... SET clause can also be used to indicate column names and values

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

  – The above example can also be written with SET as follows;

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

INSERT ... SELECT

- 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
```

  - *table_name* and *column_list* work the same way as they do for `INSERT ... VALUES` statements
  - *query_expression* is mandatory
    - If the *column_list* is specified, this query must produce exactly as many columns as specified by the *column_list*

```
INSERT INTO Top10Cities (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

INSERT with LAST_INSERT_ID()

- LAST_INSERT_ID() retrieves the last AUTO_INCREMENT value
- INSERT/ LAST_INSERT_ID() example

```
INSERT INTO City (name, countrycode)
  VALUES ('Sarah City', 'USA');
Query OK, 1 row affected (#.## sec)

SELECT LAST_INSERT_ID();
+-------------------+
| LAST_INSERT_ID() |
+-------------------+
|              4080 |
+-------------------+
1 row in set (#.## sec)
```

The DELETE Statement (1/2)

- Emptying a table completely

```
DELETE FROM table_name
```

- Remove specific rows of data

```
DELETE FROM table_name [WHERE where_condition][ORDER BY…]
  [LIMIT row_count];
```

- Example

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

- The DELETE statement removes entire rows
  - Does not include a specification of columns

The DELETE Statement (2/2)

- DELETE supports ORDER BY and LIMIT clauses, which provide finer control over the way records are deleted
  - LIMIT can be useful to remove only some instances of a given set of records

    ```
    DELETE FROM people WHERE name='Emily' LIMIT 4;
    ```

    - 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

    ```
    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

The UPDATE Statement (1/4)

- Modifies contents of existing rows

```
UPDATE table_name SET column=expression(s)
 [WHERE where_condition][ORDER BY...][LIMIT row_count];
```

- Use with the SET clause for column assignments
- Optionally use WHERE
- Example

```
UPDATE Country SET Population = Population * 1.1;
Query OK, 232 rows affected (#.## sec)
Rows matched: 239 Changed: 232 Warnings:0
```

The UPDATE Statement (2/4)

- Effects subject to column constraints
  - An attempt to update a column to a value that doesn't match the column definition will be converted or truncated by the server
- Updates can have no effect
  - Matches no records
  - No change to column values

The UPDATE Statement (3/4)

- Inconsistent ordering by default
- Use ORDER BY and LIMIT to control order/count
- Pre-updated rows

```
SELECT * FROM people;
+--------------------+
| id | name   | age |
+--------------------+
| 1  | Victor | 21  |
| 2  | Susan  | 15  |
| 3  | Victor | 31  |
+--------------------+
3 rows in set (#.## sec)
```

The UPDATE Statement (4/4)

LAB

- Examples

```
UPDATE people
SET id=id-1;
```

Does not put the id's in order After subscription occurs (4 to 3, 3 to 2) ...

```
+-------------------+
| id | name   | age |
+-------------------+
| 2  | Susan  | 15  |
| 1  | Victor | 21  |
| 3  | Victor | 31  |
+-------------------+
```

```
UPDATE people
SET id=id-1
ORDER BY id;
```

Solves ordering issue…

```
+-------------------+
| id | name   | age |
+-------------------+
| 1  | Victor | 21  |
| 2  | Susan  | 15  |
| 3  | Victor | 31  |
+-------------------+
```

```
UPDATE people
SET name='Vic'
WHERE name='Victor'
LIMIT 1;
```

After id renumbering is finalized, this update changes one name and limits output to only changed row…

```
+-----------------+
| id | name   | age |
+-----------------+
| 1  | Vic    | 21  |
+------------------+
1 row in set (#.## sec)
```

The REPLACE Statement (1/2)

- MySQL extension to SQL standard
- Exactly the same as INSERT
  - Except when it is a PRIMARY KEY or UNIQUE constraint
- General syntax

```
REPLACE INTO table_name (column_list) VALUES(value_list);
```

- Example

```
REPLACE INTO people (id,name,age) VALUES(12,'Bruce',25);
```

- *Only* useful with PRIMARY KEY or UNIQUE

The REPLACE Statement (2/2)

- Returns sum of rows deleted and inserted
- REPLACE algorithm
  - Try to insert the new row into the table
  - 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

LAB

INSERT with ON DUPLICATE KEY UPDATE Instead of REPLACE

- ON DUPLICATE KEY is like REPLACE but "nicer"
  - REPLACE
    - New row is added to the table, the old row is discarded
  - ON DUPLICATE KEY UPDATE
    - The old row is preserved, the new row is discarded

The TRUNCATE TABLE Statement

- Always removes all records
- General syntax

```
TRUNCATE TABLE table_name;
```

- DELETE vs. TRUNCATE TABLE

DELETE	TRUNCATE TABLE
Can delete specific rows with WHERE	Cannot delete specific rows, deletes *all* rows
Usually executes more slowly	Usually executes more quickly
Returns a true row count	May return a row count of zero
Transactional	May reset AUTO_INCREMENT
	Not Transactional

Further Practice: Chapter 11

- Comprehensive exercises

Chapter Summary

- 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

Course Content

Learning Objectives

- Use transaction commands to run multiple SQL statements concurrently
- Describe and use the ACID transaction rules
- Isolate one transaction from another

What is a Transaction? (1/2)

- In database programming, a transaction is a collection of data manipulation execution steps that are treated as a single unit of work
  - Execution steps are performed as if there were a single specialized command that accomplishes exactly that combination of actions

Non-Transactional Executions	Transactional Executions
Remove $1000 from account #10001	Remove $1000 from account #10001
Write to database	Deposit $1000 into account #10243
Deposit $1000 into account #10243	Write to database
Write to database	

What is a Transaction? (2/2)

- All of the data manipulation steps must be carried out
- If any portion fails, action must be taken to:
  - Permanently retain those operations that did succeed
    - **- or -**
  - Disregard those operations that did succeed

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

ACID

- **A**tomic
  - All statements execute successfully or are canceled as a *unit*
- **C**onsistent
  - Database that is in a consistent state when a transaction begins, is left in a consistent state by the transaction
- **I**solated
  - One transaction does *not* affect another
- **D**urable
  - All changes made by transaction that complete successfully are recorded properly in database--Changes are *not* lost

Transaction Control Statements

- START TRANSACTION (or BEGIN)
  - 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

AUTOCOMMIT Mode (1/2)

- Determines how and when new transactions are started
- Autocommit enabled
  - A single SQL statement implicitly starts a new transaction by default
    - The transaction is automatically committed if the statement executes successfully
    - If the statement does not execute successfully, the transaction is automatically rolled back
  - Transactions can still be started explicitly using the START TRANSACTION statement
- Autocommit disabled
  - Transactions span multiple statements by default
  - Transactions can be explicitly committed or rolled back
  - A new transaction is implicitly started after termination of previous

AUTOCOMMIT Mode (2/2)

- Autocommit disabled
  - Transactions span multiple statements by default
  - Transactions must be explicitly committed or rolled back
  - A new transaction is implicitly started after successful termination of previous transaction
  - Unsuccessful statements will result in any potential changes by that statement being undone
    - The transaction continues to remain open until committed or rolled back as a whole

Controlling AUTOCOMMIT Mode (1/2)

- The autocommit mode can be controlled through the server variable AUTOCOMMIT
- Session level control

  `SET AUTOCOMMIT = OFF` ... or ... `SET SESSION AUTOCOMMIT = OFF`

  ... or ... `SET @@autocommit := 0` (set to 1 to enable)

- Determining current autocommit setting

```
SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
|            0 |
+--------------+
```

Controlling AUTOCOMMIT Mode (2/2)

- By default, autocommit is enabled
  - Disable if transactions that span multiple statements are required
- Server configuration default behavior can be changed

  `SET AUTOCOMMIT = OFF` ... or ... `SET SESSION AUTOCOMMIT = OFF`

- Option files
  - `my.cnf` or `my.ini` option file
  - Solution does not work for users with the SUPER privilege (including root)

Implicit COMMIT's

- COMMIT explicitly commits the current transaction
- Other statements that cause commit's
  - **`START TRANSACTION`**
  - **`SET AUTOCOMMIT`** = 1 (or ON)
- Statements that have the potential to cause commit's
  - Data definition statements (**`ALTER`**, **`CREATE`**, **`DROP`**)
  - Data access and user management statements (**`GRANT`**, **`REVOKE`**, **`SET PASSWORD`**)
  - Locking statements (**`LOCK TABLES`**, **`UNLOCK TABLES`**)
- DML statements that cause implicit commit's
  - **`TRUNCATE TABLE`**, **`LOAD DATA INFILE`**

Transaction Demo: ROLLBACK

```
START TRANSACTION;

SELECT name FROM City WHERE id=3803;
+----------+
| name     |
+----------+
| San Jose |
+----------+

DELETE FROM City WHERE id=3803;
Query OK, 1 row affected (#.## sec)

SELECT name FROM City WHERE id=3803;
Empty set (#.## sec)

ROLLBACK;

SELECT name FROM City WHERE id=3803;
+----------+
| name     |
+----------+
| San Jose |
+----------+
```

View Available Storage Engines

- Check for a Transactional Storage Engine

```
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
...
```

Isolation Levels

- Concurrent Transactions Can Cause Problems
- Storage Engines Implement Isolation Levels
  - Controls level of visibility between transactions
  - May vary per database servers
- Three Common Problems
  - “Dirty” Read
  - Non-Repeatable Read
  - Phantom Row

Isolation Levels (1/3)

- InnoDB Implements Four Isolation Levels
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE
- Serializable versus Repeatable Read
- Levels Only Relevant with Simultaneous Transactions

Isolation Levels (2/3)

Transaction Isolation Level Characteristics			
	Dirty Read	**Non-Repeatable Read**	**Phantom Read**
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible (not for InnoDB)
Serializable	Not Possible	Not Possible	Not Possible

- Setting the Level
  - Use the `--transaction-isolation` option

```
[mysqld]
transaction-isolation = [READ-UNCOMMITTED | READ-COMMITTED
| REPEATABLE READ | SERIALIZABLE]
```

  - mysql server commands

```
SET [SESSION] TRANSACTION ISOLATION LEVEL isolation_level;
SET GLOBAL TRANSACTION ISOLATION LEVEL isolation_level;
```

Isolation Levels (3/3)

- Clients can modify transaction isolation levels for their own sessions
- Changing the default transaction isolation level globally requires the SUPER privilege
- View the current isolation level

```
SELECT @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
```

- View global and session levels

```
SELECT @@global.tx_isolation, @@session.tx_isolation;
+-----------------------+------------------------+
| @@global.tx_isolation | @@session.tx_isolation |
+-----------------------+------------------------+
| READ-UNCOMMITTED      | REPEATABLE-READ        |
+-----------------------+------------------------+
```

Transaction Demo: Isolation (1/2)

<table>
<tr><th>Sesion 1</th><th>Session 2</th></tr>
<tr><td><pre>mysql> PROMPT s1>

s1> SET GLOBAL TRANSACTION
 -> ISOLATION LEVEL READ COMITTED;

s1> SELECT @@tx_isolation;
+-----------------+
| @@tx_isolation |
+-----------------+
| READ-COMMITTED |
+-----------------+</pre></td><td></td></tr>
<tr><td></td><td><pre>mysql> PROMPT s2>

s2> INSERT INTO City
 -> (Name, CountryCode,
 -> population)
 -> VALUES ('Sakila', 'SWE', 1);</pre></td></tr>
<tr><td><pre>s1> SELECT Name, CountryCode
 -> FROM City
 -> WHERE Name = 'Sakila';</pre>Gives an empty set as the current isolation level prevents this transaction from seeing uncommitted changes.</td><td></td></tr>
</table>

Transaction Demo: Isolation (2/2)

Sesion 1	Session 2
	s2> COMMIT;
s1> SELECT Name, CountryCode -> FROM City -> WHERE Name = 'Sakila'; +--------+-------------+ | Name | CountryCode | +--------+-------------+ | Sakila | SWE | +--------+-------------+	
s1> PROMPT	s2> PROMPT

Locking Concepts

- A Locking Mechanism Prevents Problems with Concurrent Data Access
- Locks are Managed By the Server
  - Allows access to one client and locks others out
- Locking Depends on Access Type
  - READ vs. WRITE

Locking Reads

- InnoDB Supports Two Types of Locking
  - LOCK IN SHARE MODE -- locks each row with a *shared lock*
  - FOR UPDATE -- locks each row with an *exclusive lock*

- LOCK IN SHARE MODE Example

```
SELECT * FROM Country WHERE Code='AUS' LOCK IN SHARE MODE\G
```

- FOR UPDATE Example

```
SELECT counter_field FROM child_codes FOR UPDATE;
UPDATE child_codes SET counter_field = counter_field + 1;
```

Transaction Demo: Deadlock

Sesion 1	Session 2

`START TRANSACTION;`
 	
`UPDATE Country`
`SET name='Sakila'`
` WHERE code='SWE';`	

	`START TRANSACTION;`
	
	`UPDATE Country`
	`SET name='World Cup Winner'`
	`WHERE code='ITA';`

`DELETE FROM Country`
`WHERE code='ITA';`
- This will hang, waiting for lock	

	`UPDATE Country SET`
	`population=1 WHERE`
	`code='SWE';`
	- DEADLOCK detected!

DELETE stops hanging and executes successfully	

Further Practice: Chapter 12

- Comprehensive exercises

Chapter Summary

- Use transaction commands to run multiple SQL statements concurrently
- Describe and use the ACID transaction rules
- Isolate one transaction from another

Course Content

Learning Objectives

- Describe the concept of joining tables
- Understand the construction and properties of the Cartesian product
- Utilize the syntax and application of different join types
- Understand the need to use qualified column references and table aliases to avoid ambiguity
- Join a table to itself
- Utilize multi-table **UPDATE** and **DELETE** statements

Joins

- What is a join operation?
  - A join is an operation upon two tables
  - Creates new rows by combining (joining) rows from two tables
  - Combined rows form a new table

Single Table Query Limitation

- Multiple step processes to see all the data
  - Locating all the cities with the name “London”

```
SELECT * FROM City WHERE Name = 'London';
+------+--------+-------------+----------+------------+
| ID   | Name   | CountryCode | District | Population |
+------+--------+-------------+----------+------------+
| 456  | London | GBR         | England  | 7285000    |
| 1820 | London | CAN         | Ontario  | 339917     |
+------+--------+-------------+----------+------------+
```

  - Determining which country they are located in

```
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 |
+------+----------------+---------------+------------+
```

Combining Two Simple Tables

- Keep it simple for practice
  - A simple City table

```
CREATE TABLE SimpleCity AS
SELECT Name AS CityName, CountryCode, Population AS CityPop
FROM City WHERE Name LIKE 'London';
```

  - A simple Country table

```
CREATE TABLE SimpleCountry AS
SELECT Code, Name AS CountryName, Population AS CountryPop
FROM Country WHERE Code IN ('CAN', 'GBR')
```

Combining Two Simple Tables

SimpleCity

CityName	CountryCode	CityPop
London	GBR	7285000
London	CAN	339917

SimpleCountry

Code	CountryName	CountryPop
CAN	Canada	34261700
GBR	United Kingdom	65585740

Result

CityName	CountryCode	CityPop	Code	CountryName	CountryPop
London	GBR	7285000	CAN	Canada	7285000
London	CAN	339917	GBR	United Kingdom	339917

- The column layout of the result is good
- The row combinations are not good!
- Another approach is needed ...

Combining Rows: Cartesian Product

- All possible pairs of rows from two tables
  - "product", "cross product" or "Cartesian product"

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

Cartesian Product: SimpleCity Loop

```
goto_first_row(SimpleCity)
while has_rows(SimpleCity) do
    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
```

Cartesian Product: SimpleCountry Loop

```
goto_first_row(SimpleCity)
while has_rows(SimpleCity) do
    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
```

Cartesian Product: SimpleCountry Loop

combine rows

```
goto_first_row(SimpleCity)
while has_rows(SimpleCity) do
    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
```

Cartesian Product

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**			

Cartesian Product

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

Cartesian Product

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**			

Cartesian Product

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**

Cartesian Product

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**			

Cartesian Product

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**

Cartesian Product

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**			

Cartesian Product

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**

Dimensions of the Cartesian Product

- #columns: sum #columns of constituent tables
- #rows: multiply #rows of constituent tables
- SimpleCity:
  - 3 columns, 2 rows
- SimpleCountry:
  - 3 columns, 2 rows
- Product of SimpleCity and SimpleCountry:
  - 3 + 3 = 6 columns
  - 2 * 2 = 4 rows
- Cartesian products easily grow very large

Order of tables is not of real importance

- When creating a Cartesian product, table processing order influences the order of columns and rows
- This is not that important though:
  - The row order is not of importance from a relational point of view
  - The column order is not that important as long as each column can still be identified
- Changing the order in which tables are processed does not change the information content of the product table

A Cartesian product is itself a table

- A Cartesian product is not a 'real', physical table, however, the result has a tabular form
- Most operations that can be applied to a table can in principle be applied to a Cartesian product
- Because the product is similar to a 'real' table, we can process it as a table in a subsequent product operation
- This allows products of more than two tables to be conceived as a series of products of two tables:

T1 * T2 * T3 = (T1 * T2) * T3

Cartesian Product of more than 2 Tables

- Continuing to keep it simple for practice
  - A simple Language table

```
CREATE TABLE SimpleLanguage AS
SELECT CountryCode, Language FROM CountryLanguage
WHERE CountryCode IN ('CAN','GBR') AND IsOfficial = 'T';
```

- "Glueing" SimpleLanguage to the other two tables
  - All columns from SimpleCity (3), SimpleCountry (3) and SimpleLangauge (2) yields 3 + 3 + 2 = 8 columns
  - Multiplication of the # of rows in the SimpleCity (2), SimpleCountry (2) and SimpleLanguage (3) tables yields 2 * 2 * 3 = 12 rows

Cartesian Product of Three Tables

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

Filtering Out Undesired Rows/Columns

- Discard unwanted rows (filter):

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~~

- Discard unwanted columns (projection):

SimpleCity			SimpleCountry		
CityName	~~CountryCode~~	~~CityPop~~	~~Code~~	CountryName	~~CountryPop~~
London	~~GBR~~	~~7285000~~	~~GBR~~	United Kingdom	~~65585740~~
London	~~CAN~~	~~339917~~	~~CAN~~	Canada	~~34261700~~

Final Result and Recapitulation

- City names and name of respective country:

SimpleCityCountry	
CityName	**CountryName**
London	United Kingdom
London	Canada

- Recapitulation of the join operation
  - Construct the Cartesian product
  - Remove unwanted rows
  - Remove unwanted columns

Joins and Foreign Keys

- In many cases, rows are joined according to a foreign key
  - In the example, rows were retained in case the **`CountryCode`** column in the **`SimpleCity`** table matched the **`Code`** column in the **`SimpleCountry`** table
- Joining based on a foreign key is a very common pattern
  - For each row in the referencing table, the join operation 'looks up' data in the referenced table

FK Join and Denormalization (1/2)

- Usually, base tables are in 3rd normal form
- The result of joining based on a foreign key is a denormalized table (1st normal form)
  - The join result has the 'primary key' of the referencing table
- In the join result, the columns 'borrowed' from the referenced table are:
  - Functionally dependent upon the foreign key columns
  - Functionally dependent upon the columns that originate from any unique keys in the referenced table

FK Join and Denormalization (2/2)

Country (3rd normal form)	
Code (PK)	CountryName
CAN	Canada
GBR	United Kingdom

1:n

City (3rd normal form)		
CityID (PK)	**CountryCode (FK)**	CityName
456	GBR	London
1820	CAN	London

Join Result (1st normal form)				
CityID (PK)	CountryCode	CityName	Code	CountryName
456	GBR	London	GBR	United Kingdom
1820	CAN	London	CAN	Canada

Functional dependencies on FK:

Joining in SQL using a Cartesian product

- Cartesian product using the 'comma join'
- Separate multiple table names with a comma (“,“)

Comma

```
SELECT *
FROM   SimpleCity, SimpleCountry;
```

```
+--------+----------+-------------+------+----------------+---------+
| CityID | CityName | CountryCode | Code | CountryName    | Capital |
+--------+----------+-------------+------+----------------+---------+
|    456 | London   | GBR         | CAN  | Canada         |    1822 |
|   1820 | London   | CAN         | CAN  | Canada         |    1822 |
|    456 | London   | GBR         | GBR  | United Kingdom |     456 |
|   1820 | London   | CAN         | GBR  | United Kingdom |     456 |
+--------+----------+-------------+------+----------------+---------+
```

Using WHERE to retain matching rows

- The WHERE clause can be used to retain only those rows that satisfy a condition
  - We can write a condition to require matching SimpleCity and SimpleCountry rows

```
SELECT *
FROM   SimpleCity, SimpleCountry
WHERE  CountryCode = Code
```

```
+--------+----------+-------------+------+----------------+---------+
| CityID | CityName | CountryCode | Code | CountryName    | Capital |
+--------+----------+-------------+------+----------------+---------+
|   1820 | London   | CAN         | CAN  | Canada         |    1822 |
|    456 | London   | GBR         | GBR  | United Kingdom |     456 |
+--------+----------+-------------+------+----------------+---------+
```

The Join Condition

- The WHERE is 'just' an ordinary WHERE clause
  - The WHERE clause can contain any condition
  - requiring matching rows is 'just' a condition
  - Still, we like to consider the condition special
- A *join condition* is the condition that compares the columns of two joined tables

SimpleCity

SimpleCountry

```
SELECT *
FROM   SimpleCity, SimpleCountry
WHERE  CountryCode = Code
AND    CityPop < 1000000
```

some condition

Join condition

Ambiguous Column Names (1/2)

- Potential ambiguity when joining tables
  - A joined table may contain a column that has a name identical to that of a column in the table it is joined with
- Column name alone may not be enough to identify a column:

Country

City

???

ambiguous

```
SELECT Continent,
       CountryCode,
       Name
FROM   City, Country
WHERE  CountryCode = Code
```

Ambiguous Column Names (2/2)

- Avoid ambiguity by ***qualifying*** column names
- Separate table name and column name with a dot

table name (qualifier)

dot

column name

```
SELECT Continent,
       CountryCode,
       Country.Name,
       City.Name
FROM   City, Country
WHERE  CountryCode = Code
```

qualified column names

- Qualified columns can appear almost anywhere
- You may also qualify unambiguous columns

Table Aliases

- In SQL statements, tables can be given an ***alias***
  - Alternative name for local use in the statement
- When qualifying a column of an aliased table, the alias must be used as qualifier – not the table name
- Alias follows after the table name
- Optionally, separate table name and alias with the keyword **AS**: <***table-reference***> [**AS**] <***alias***>

alias as qualifier
AS keyword (optional)
alias

```
SELECT Capital.Name, Country.Name
FROM   Country,
       City AS Capital
WHERE  Capital = Capital.ID
```

Aliased table

Reasons for Using Table Aliases

- Alias may be shorter than the full table name
  - more convenient to type
- Queries becomes more resilient to table name changes
  - Only the table names need to be changed; all columns use the alias as qualifier
- Resolve ambiguity of table names
  - Needed whenever a statement uses two instances of the same table
- May clarify the purpose of the table in the statement

```
SELECT Capital.Name, Country.Name
FROM Country, City AS Capital
WHERE Capital = Capital.ID;
```

Common Alias Error

- If a table has been given a table alias, the alias must be used instead of the table name when qualifying a column

```
SELECT Country.Name, City.Name
FROM   Country, City AS Capital
WHERE  Capital = Capital.ID;
```

- Second expression in the SELECT list attempts to refer to the Name column of the City table
  - City table has been given the table alias Capital, locally renaming that particular instance of the City table


```
ERROR 1054 (42S22): Unknown column
 'City.Name' in 'field list'
```

Basic Join Syntax

- SQL offers the **`JOIN`** syntax
  - Allows separation of the join condition from other conditions
- Syntax:

```
<table-ref> [<join-type>] JOIN <table-ref>
 ON <join-condition>
```

- Example:

```
SELECT *
FROM   SimpleCity JOIN SimpleCountry
ON     CountryCode = Code
WHERE  CityPop < 1000000
```

Join condition

Non-join condition

- **`ON`** clause still allows non-join conditions
  - Better to put those in the **`WHERE`**

LAB

INNER JOIN

- The inner join operation is characterized by the fact that its result contains only rows for which the join condition is satisfied
  - Previous comma join and **`JOIN`** examples are all inner joins
- Explicit syntax for the inner join operation:
  - Use **`INNER`** keyword before the **`JOIN`** keyword
  - If the <*join-type*> is omitted, **`INNER`** is implied
- Example:

```
SELECT *
FROM   SimpleCity INNER JOIN SimpleCountry
ON     CountryCode = Code
```

INNER JOIN Pseudocode

```
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
```

Ommitting the Join Condition

- For the **`INNER JOIN`** syntax, MySQL allows the **`ON`** clause to be omitted
  - This has the effect of constructing a Cartesian product

```
SELECT * FROM SimpleCity INNER JOIN SimpleCountry;
```

... Is equivalent to ...

```
SELECT * FROM SimpleCity, SimpleCountry;
```

- The defining characteristic of an inner join operation is to produce only the rows that satisfy the join condition
  - This implies a join condition **should** be present
- In most cases, it does not make much sense to deliberately omit the join condition
  - For **`INNER JOIN`**, always write a join condition

Outer Joins

- The result of an outer join contains all rows for which the join condition was satisfied.
- In addition, the outer join result contains some rows for which the join condition could not be satisfied

Two Simple Tables (1/2): SimpleCountry

```
CREATE TABLE SimpleCountry
AS
SELECT Code, Name AS CountryName, Capital
FROM Country
WHERE Code IN ('CAN', 'GBR');

+------+----------------+---------+
| Code | CountryName    | Capital |
+------+----------------+---------+
| CAN  | Canada         |    1822 |
| GBR  | United Kingdom |     456 |
+------+----------------+---------+
```

Two Simple Tables (2/2): SimpleCity

```
CREATE TABLE SimpleCity
AS
SELECT ID as CityID, Name AS CityName, CountryCode
FROM City
WHERE Name LIKE 'London';

+--------+----------+-------------+
| CityID | CityName | CountryCode |
+--------+----------+-------------+
|    456 | London   | GBR         |
|   1820 | London   | CAN         |
+--------+----------+-------------+
```

Inner Join to Find the Capital City

```
SELECT CountryName, CityName
FROM   SimpleCountry INNER JOIN SimpleCity
ON     Capital = CityID;

+----------------+----------+
| CountryName    | CityName |
+----------------+----------+
| United Kingdom | London   |
+----------------+----------+
```

The row for `'Canada'` is missing

Inner Join Discards the Unmatched Row

- The **Capital** column for the '**Canada**' row in **SimpleCountry** does not match any **CityID** column in **SimpleCity**
  - The join condition is not satisfied
  - The '**Canada**' row is discarded and does not appear in the join result

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

Outer Join Operation

- What if we want a list of *all* countries, and if possible, the capital city?
  - Retain the row from **`SimpleCountry`** even if no corresponding capital was found in **`SimpleCity`**
- An outer join operation achieves exactly that

Outer join operation: pseudocode

```
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
```

The LEFT OUTER JOIN Syntax

- Syntax:

```
<left-table> LEFT [OUTER] JOIN <right-table>
ON <join-condition>
```

- Note that the **OUTER** keyword is optional
  - Usually omitted
- The **LEFT OUTER JOIN**:
  - Returns all rows that match the join condition
  - Retains unmatched rows from *<left-table>*
  - Substitutes **NULL** for *<right-table>* columns for each unmatched row from *<left-table>*

LEFT OUTER JOIN Example

- Example query:

```
SELECT    CountryName, CityName
FROM      SimpleCountry
LEFT JOIN SimpleCity
ON        Capital = CityID;

+----------------+----------+
| CountryName    | CityName |
+----------------+----------+
| Canada         | NULL     |
| United Kingdom | London   |
+----------------+----------+
```

RIGHT OUTER JOIN Syntax

- Syntax:
  `<left-table> RIGHT [OUTER] JOIN <right-table>`
  `ON <join-condition>`

- Same as **LEFT OUTER JOIN** syntax except that the keyword **RIGHT** is used instead of **LEFT**
- The **RIGHT OUTER JOIN**:
  - Returns all rows that match the join condition
  - Returns unmatched rows from *<right-table>*
  - Substitutes **NULL** for *<left-table>* columns for each unmatched row in *<right-table>*

Equivalent LEFT and RIGHT JOIN Syntax

- RIGHT and LEFT join are not really different types
- The original **LEFT JOIN**:

```
SELECT CountryName,
       CityName
FROM   SimpleCountry LEFT JOIN SimpleCity
ON Capital = CityID;
```

- Equivalent **RIGHT JOIN**

```
SELECT CountryName,
       CityName
FROM   SimpleCity RIGHT JOIN SimpleCountry
ON     CityID = Capital;
```

Outer Joins: The Join Condition

- The outer join syntax does not allow the join condition to be omitted
  - For the inner join, it is possible to accidentally omit the join condition
- In the join condition, the order of appearance of columns has no effect on the evaluation of the condition
  - It is a good idea to use the same order of appearance for the columns as for the tables
  - Write columns as near as possible to the table from which they originate

Outer joins: `ON` vs `WHERE`

- For outer joins, it matters whether conditions are placed in the **`ON`** or in the **`WHERE`** clause
  - The **`ON`** clauses are processed before the **`WHERE`** clause
- Example: "*All cities, and in case the city is the capital of an independent country, the country name and year of independence*"
  - Join City and Country over the Capital foreign key
  - "*All cities...*", capitals or not, so we need an outer join
  - Some additional complexity for "*...independent country...*"

Outer Joins: ON vs WHERE

- Basic query:

```
SELECT City.Name,
       Country.Name,
       Country.IndepYear
FROM   City LEFT JOIN Country
ON     City.ID = Country.Capital
```

- This yields: “All cities, and the country name and year of independence in case it's a capital”
  - Still need to check if the country is independent

Outer Joins: ON vs WHERE

- **WHERE** clause:

```
SELECT City.Name,
       Country.Name,
       Country.IndepYear
FROM   City LEFT JOIN Country
ON     City.ID = Country.Capital
WHERE  Country.IndepYear IS NOT NULL
```

- **ON** clause:

```
SELECT City.Name,
       Country.Name,
       Country.IndepYear
FROM   City LEFT JOIN Country
ON     City.ID = Country.Capital
AND    Country.IndepYear IS NOT NULL
```

Choosing Between Inner and Outer Joins

- With inner joins, you may lose rows
- Don't settle blindly for outer joins
  - Outer joins are usually slower than inner joins
  - An outer join usually implies dealing with `NULL` values in expressions elsewhere in the statement
- Typical cases that require an outer join:
  - Nullable columns in the join condition
  - Joining to another outer joined table
  - Solving a 'not exists' problem
  - Aggregating related rows

Nullable Columns in the Join Condition

- If the columns used in the join condition are nullable, an inner join may result in missing rows
  - Use an outer join if this is not acceptable
- Typical case: optional foreign key columns
- Example: `Capital` column in `Country`
  - Some `Country` rows have a `NULL` in `Capital`
  - These rows will be discarded when using an inner join
  - Any query that needs to show ***all*** countries with their capital should consider using an outer join

Joining to Another Outer Joined Table (1/2)

- Variation on theme "nullable column in join condition"
- Example: Assume a **`LEFT`** join to retrieve all cities, and if applicable, the country of which it is the capital. If the city is a capital, also list all languages spoken in the entire country
  - The condition to join **`Country`** and **`CountryLanguage`** is based on non-nullable columns
  - However, the **`Country`** columns may be **`NULL`** anyway because it was **`LEFT`** joined
  - **`CountryLanguage`** **must** also be **`LEFT`** joined

Joining to Another Outer Joined Table (2/2)

```
SELECT    Ci.Name,
          Co.Name AS `Capital of`,
          Cl.Language
FROM      City    Ci
LEFT JOIN Country Co
ON        Ci.ID = Co.Capital
LEFT JOIN CountryLanguage Cl
ON        Co.Code = Cl.CountryCode
```

columns of **Co** may be **NULL**

- **Country.Code** is defined to be **NOT NULL**...
  ....but **Co.Code** is nullable!!!
- **CountryLanguage** must be **LEFT** joined too

Solving 'Not Exists' Problems

- The outer join operation produces **NULL** values in case no row is found that satisfies the join condition
- With a **WHERE** clause, we can filter to find these **NULL** values
  - Test a column that is defined as **NOT NULL** in order to distinguish between a 'real' **NULL** and one generated by the outer join
- This can be used to solve queries like:
  - "Which cities are not a capital?"
  - "Which countries do not have any cities"
  - "Which cities do not belong to any country"

Aggregating related rows

- Typical case: Calculating the number of referencing rows in a related table
- Example: how many cities are situated in a country
  - with an inner join, countries without cities are discarded, whereas we'd like to see 0 (zero)

```
SELECT   Country.Code,
         COUNT(City.ID)
FROM     Country LEFT JOIN City
ON       Code = CountryCode
WHERE    Country.Name = 'Antarctica'
GROUP BY Country.Code
```

Aggregation

Other types of joins

- Joins on identically named columns:
  - **NATURAL** join
  - Joins with **USING** (named columns join)
- The **CROSS JOIN** (explicit Cartesian product):
  - *<table-ref>* **CROSS JOIN** *<table-ref>*
- Join condition categories
  - The equi join
  - The non-equi join
    - The **BETWEEN...AND** join
- The autojoin (self-join)

The NATURAL JOIN

- Does not allow an explicit join condition
- Instead, applies an implicit join condition based on an equality comparison of all identically named columns present in the two joined tables
- When using an 'all columns' wildcard (*) in the **SELECT** list, identically named columns are reported only once
- Syntax:

```
<table-reference>
NATURAL [<outer-join-type>] JOIN
<table-reference>
```

NATURAL JOIN Example

```
SELECT       *
FROM         SimpleLanguage
NATURAL JOIN SimpleCity
```

NATURAL Join

- The **CountryCode** column occurs in both tables
  - An implicit join condition is applied requiring equality
  - The 'all columns' wildcard will expand to include only one **CountryCode** column
- Equivalent ordinary statement:

Common columns merged

Common columns must be equal

```
SELECT     COALESCE(l.CountryCode,
                    c.CountryCode) AS CountryCode
                 l.Language, c.CityID, c.CityName
FROM       SimpleLanguage l
INNER JOIN SimpleCity     c
ON         l.CountryCode = c.CountryCode
```

Joins with USING (Named Columns Join)

- Like **NATURAL JOIN**, joins on an equality comparison of all identically named columns
- Unlike **NATURAL JOIN**, join columns can be specified
- Like with **NATURAL JOIN**, an 'all columns' wildcard (*) reports identically named columns only once
- Syntax:

```
<table-reference> [<join-type>]
JOIN
<table-reference>
USING (colum1[,..., columN])
```

Joins with USING Example

```
SELECT *
FROM   SimpleLanguage l INNER JOIN SimpleCity c
USING  (CountryCode)
```

named columns join

- The **CountryCode** column occurs in both tables
  - The **CountryCode** columns from both tables will be compared using an equality comparison
  - The 'all columns' wildcard will expand to include only one **CountryCode** column
- Equivalent ordinary statement:

Named columns merged

Named columns required to be equal

```
SELECT COALESCE(l.CountryCode,
                c.CountryCode) AS CountryCode
       l.Language, c.CityID, c.CityName
FROM   SimpleLanguage l INNER JOIN SimpleCity c
ON     l.CountryCode = c.CountryCode
```

Cross Join

- A join operation that computes a Cartesian product
- Syntax:

```
<table-reference>
JOIN
<table-reference>
```

- Syntax model does not contain a join-condition
  - 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
  - Precedence
    - Comma has the lowest precedence
    - Cross Join is on the same precedence level as all other join types

Equijoin and Non-equijoin

- Equijoin:
  - join condition contains only column comparisons using the equals operator
- Non-equijoin
  - Anything that is not an equijoin
- **BETWEEN...AND** join

```
SELECT Employee.ID, Bonus.Amount
FROM   Employee INNER JOIN Bonus
ON     Employee.Salary
       BETWEEN Bonus.LowerSalaryBound
       AND     Bonus.UpperSalaryBound
```

BETWEEN...AND

Joins in DELETE and UPDATE Statements

- In MySQL, join operations are not confined to **SELECT** statements
  - Supported in **UPDATE** and **DELETE** statements
  - An **UPDATE** or **DELETE** statement containing a join is called a ***multi-table*** **UPDATE** / **DELETE** statement
- The join result can be used to make a specific selection to **UPDATE** or **DELETE**
- Multi-table **UPDATE** / **DELETE** statements can modify the contents of multiple tables in a single statement

Multi-table UPDATE statements

- Syntax:

```
UPDATE <joined-tables>
SET    <column-assignments>
[WHERE <condition>]
```

- *<column-assignments>* is a comma separated list of column assignments
- Column assignment:

```
<column-reference> = <value-expression>
```

Multi-table UPDATE statements: Example

- Add 20,000 to the population of 'Rio de Janeiro'

SET keyword

Column (assignment target)

assignment operator

value expression

comma

```
UPDATE       City       Ci
INNER JOIN   Country    Co
ON           Ci.CountryCode = Co.Code
SET          Ci.Population  = Ci.Population
                            + 20000,
             Co.Population  = Co.Population
                            + 20000
WHERE        Co.Name        = 'Brazil'
AND          Ci.Name        = 'Rio de Janeiro'
```

Multi-table DELETE Statements

- Syntax:

```
DELETE <table-list>
FROM   <joined-tables>
[WHERE <condition>]
```

- Alternative syntax:

```
DELETE
FROM   <table-list>
USING  <joined-tables>
[WHERE <condition>]
```

- *<table-list>* is a comma-separated list of tables from which to delete rows selected by the *<joined-tables>*

Multi-table DELETE Statements: Example

- Remove the country with the code `'NLD'` as well as all its cities and languages from the `world` database:

```
DELETE      Country, City, CountryLanguage
FROM        Country
LEFT JOIN   City
ON          Code = City.CountryCode
LEFT JOIN   CountryLanguage
ON          Code = CountryLanguage.CountryCode
WHERE       Code = 'NLD'
```

Multi-table DELETE / UPDATE Advantages

- Packing multiple **UPDATE** or **DELETE** statements into a single multi-table **UPDATE** or **DELETE** statement reduces roundtrips
- Rewrite poorly performing **UPDATE** or **DELETE** statements involving subqueries to multi-table **UPDATE** or **DELETE** statements
- Logical grouping of related statements

Multi-table DELETE / UPDATE Limitations

- Multi-table **`UPDATE`** or **`DELETE`** statements are not guaranteed to execute atomically
- No support for **`ORDER BY`** and **`LIMIT`**
- Cascading delete / update rules on InnoDB foreign key constraints may interfere with intended actions
- Non-standard syntax

Further Practice: Chapter 13

- Comprehensive Exercises

Chapter Summary

- Describe the concept of a join
- Connect data from multiple tables using various join statements
- Resolve name clashes when joining tables
- Join a table to itself
- Join tables with UPDATE and DELETE statements

Course Content

DEVELOPER I		DEVELOPER II	
1.	INTRODUCTION	13.	JOINS
2.	MySQL CLIENT/SERVER CONCEPTS	14.	SUBQUERIES
3.	MySQL CLIENTS	15.	VIEWS
4.	QUERYING FOR TABLE DATA	16.	PREPARED STATEMENTS
5.	HANDLING ERRORS AND WARNINGS	17.	EXPORTING AND IMPORTING DATA
6.	DATA TYPES	18.	STORED ROUTINES
7.	SQL EXPRESSIONS	19.	TRIGGERS
8.	OBTAINING METADATA	20.	STORAGE ENGINES
9.	DATABASES	21.	OPTIMIZATION
10.	TABLES	22.	CONCLUSION
11.	MANIPULATING TABLE DATA		
12.	TRANSACTIONS		

Learning Objectives

- 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

Subquery Overview (1/2)

- Query Nested Inside Another Query
- Enclosed in Parenthesis ()
- Example

```
SELECT   Language                         -- outer SELECT expression
FROM     CountryLanguage
WHERE    CountryCode = (                  -- left parenthesis - starts subquery
             SELECT Code                  -- subquery SELECT expression
             FROM   Country
             WHERE  Name = 'Finland'
         )                                -- right parenthesis – ends subquery
+----------+
| Language |
+----------+
| Estonian |
| Finnish  |
| Russian  |
| Saame    |
| Swedish  |
+----------+
```

Subquery Overview (2/2)

- Two-step example
  - Subquery retrieves the value of the Code column from the Country table for the country called “Finland”

```
SELECT Code FROM Country WHERE Name = 'Finland';
+------+
| Code |
+------+
| FIN  |
+------+
```

  - Substituting result of subquery

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

Scalar Subqueries (1/3)

- Scalar subqueries act as simple, singular value expressions (scalars)
  - Query expression is executed and expected to retrieve at most one row having exactly one column
  - Always evaluates to a single value expression
    - If no row is retrieved, the subquery evaluates to the NULL value
- Scalar subqueries have the same status as literals, function calls, column references and the like

```
SELECT Country.Name,
 100 * Country.Population /
  (SELECT SUM(Population) FROM Country) AS pct_of_world_pop
FROM  Country;
```

Scalar Subqueries (2/3)

- Semantical error occurs when the subquery happens to contain more than one column

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

- Semantical error occurs when the subquery happens to contain more than one row

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

- No such problem occurs if the parenthesized query expression happens to yield the empty set

```
SELECT (SELECT Name FROM world.Country LIMIT 0);
+-------------------------------------------+
| (SELECT Name FROM world.Country LIMIT 0) |
+-------------------------------------------+
| NULL                                      |
+-------------------------------------------+
```

Scalar Subqueries (3/3)

- Scalar subqueries are very useful when calculating aggregates of multiple unrelated details

```
SELECT Name,
       (SELECT COUNT(*) FROM City
         WHERE CountryCode = Code) AS Cities,
       (SELECT COUNT(*) FROM CountryLanguage
         WHERE CountryCode = Code) AS Languages
FROM   Country;
```

Row Subqueries (1/3)

- Row subqueries are treated as a single row containing at least 2 columns
- Can be used as operands
  - Equality operators =, <>, !=, <=>
    - Two rows are equal only if all corresponding column values are equal
  - Comparison operators <, >, >=, and <=
    - Order of the columns has significance
      - The row (SELECT 2, 1) is considered to be larger than the row (SELECT 1, 100) because 2 is larger than 1
      - The row (SELECT 1, 2) is considered to be smaller than (SELECT 100,1) because 1 is smaller than 100

Row Subqueries (2/3)

- Equality examples

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

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 |
+----------+
```

Row Subqueries (3/3)

- Handling invalid results examples
  - Empty set

```
SELECT (NULL, NULL) <=> (SELECT ID, Name FROM City LIMIT 0);
+-------------------------------------------------------+
| (NULL, NULL) <=> (SELECT ID, Name FROM City LIMIT 0) |
+-------------------------------------------------------+
|                                                     1 |
+-------------------------------------------------------+
```

  - More than one row returned

```
SELECT ('London', 'GBR') = (SELECT Name, CountryCode FROM City);
ERROR 1242 (21000): Subquery returns more than 1 row
```

  - Different number of columns

```
SELECT (456, 'London', 'GBR') = (SELECT Name, CountryCode FROM City);
ERROR 1241 (21000): Operand should contain 3 column(s)
```

Table Subqueries (1/3)

- Table subqueries act as (readonly) tables
  - Evaluate to a result set containing zero or more rows with one or more columns
  - They can appear in the following contexts:
    - In the **FROM** clause of an enclosing query
    - As right-hand operands to the logical operators **IN** and **EXISTS**
    - As a right hand operator to a regular comparison operator (=, !=, <>, <, >, <= or >=) quantified with **ALL**, **ANY** and **SOME**
  - The number of selected columns or the number of returned rows does not affect the subquery's status as a table subquery

Table Subqueries (2/3)

- Subqueries in the FROM clause
  - 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

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

  - 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
```

Table Subqueries (3/3)

- Subqueries in the FROM clause are especially useful for calculating aggregates of aggregates
  - Example: average of the sums of the population of each continent

```
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

Table Subquery Operators

- Only specific operators can accept a table subquery as right hand operand
  - The logical operators IN and EXISTS
  - The regular comparison operators =, !=, <>, <, >, >= and <=
    - One of the quantifiers ALL, ANY or SOME must be used to specify how the operator must be applied to the subquery
- Return boolean results
- Nothing particularly special about these comparison operators except they require a table subquery at their right hand side
- Usage of a table subquery as a right hand operand for these operators is confined mostly to the WHERE clause

IN Operator (1/3)

- 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
  - Evaluates to false otherwise
- If the table subquery selects only a single column the left hand operand must be a scalar value expression

```
SELECT  * FROM City WHERE CountryCode
   IN (SELECT Code FROM Country WHERE Continent = 'Asia');
```

  - Subquery retrieves all possible country codes for Asian countries
  - 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
    - The left hand operand row is checked for equality using pairwise column comparisons with the rows in the result set

IN Operator (2/3)

- Pairwise column comparison usage

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

  – Subquery retrieves all possible country codes for Asian countries
  – Outer query retrieves cities and compares their country code and city name to each of the country code and name rows
  – Statement returns only those City rows for which the name is equal to the name of its respective country
- Using NOT IN
  – IN operator can be negated by prefixing the IN keyword with the NOT keyword
  – NOT IN evaluates to TRUE in case the result set is known <u>not</u> to contain an entry that is equal to the left operand
    - Evaluates to FALSE if result set does contain an entry

IN Operator (3/3)

- IN operator and NULL
  - 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
      - 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

EXISTS Operator (1/2)

- 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

```
SELECT * FROM City
   WHERE EXISTS (SELECT NULL FROM Country
                  WHERE Capital = ID);
```

    - The outer query retrieves cities
    - The EXISTS operator is used to find out if there is a country of which the city happens to be the capital
  - 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

EXISTS Operator (2/2)

- Using NOT EXISTS
  - The effect of EXISTS can be negated by placing the NOT keyword before the EXISTS keyword
    - Evaluates to TRUE only if the argument subquery evaluates to the empty set, and is FALSE otherwise

```
SELECT * FROM Country
   WHERE NOT EXISTS (SELECT NULL FROM CountryLanguage
                      WHERE CountryCode = Code AND
                      Language = 'English');
```

    - Outer query retrieves all countries
    - For each Country row, the corresponding rows from CountryLanguage are retrieved in the subquery

ALL, ANY and SOME (1/4)

- ALL, ANY and SOME are quantifiers
  - Used in conjunction with the regular comparison operators (=, !=, <>, <, >, <=, and >=)
  - They 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
- Used to correct errors

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

  - The = operator cannot be used directly to compare the scalar value 'Finland' to the set of values returned by the subquery
  - The purpose of the quantifiers ALL, ANY and SOME is to 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

ALL, ANY and SOME (2/4)

- Effects of different quantifiers
  - 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 (or SOME) 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
- Quantifiers are used by placing them between the operator and the subquery:

```
<left-hand-operand> <comparison-operator> <quantifier> <table-subquery>
```

- The following example uses ANY to see if 'Finland' is a valid country name:

```
SELECT 'Finland' = ANY (SELECT Name FROM world.Country);
```

ALL, ANY and SOME (3/4)

- ALL example:

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

  – Retrieves countries only when the country's population exceeds the population of all of the cities

- SOME versus ANY

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

  - Query in English: “Retrieve all cities that have a population larger than <u>any</u> country's population”
    – All the rows from the City table are returned
  - Rethinking the query: “For each city, check if there is <u>any</u> country at all for which the city's population exceeds that of the country”
  - ... or ... “Retrieve all cities that have a population larger than <u>some</u> country's population”
  - ... or ... “Retrieve all cities with a population larger than that of <u>some</u> countries”

ALL, ANY and SOME (4/4)

- Alternatives to ANY and ALL
  - SQL queries with > ANY for which the English translation included the word “any” comes across as all rows returned by the subquery
  - SQL queries 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 to confusion, many people like to rewrite queries in order to avoid the quantifiers

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

Categories of Subqueries

- Correlated
  - References outer query
  - Cannot stand alone
- Non-Correlated
  - Does not reference outer query
  - Can stand alone

Non-Correlated Subqueries

- A subquery is *non-correlated* if the parenthesized query expression does not refer to an expression that depends on the enclosing statementReferences outer query

```
SELECT * FROM City
   WHERE CountryCode IN (SELECT Code FROM Country
                          WHERE Continent = 'Europe');
```

  - The subquery is not dependent upon any expressions derived from outside the parenthesized query expression and its result can be computed independently from the remainder of the statement
  - A non-correlated subquery is self-contained, the query expression can be executed as a standalone query:

```
SELECT Code FROM Country WHERE Continent = 'Europe';
```

  - The subquery is itself a valid SELECT statement

Correlated Subqueries (1/2)

- A subquery is *correlated* 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

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

  - Code (in the subquery) refers to a column that would be derived from the query outside the parenthesis
  - The subquery can be evaluated only when the Code column value is known already, thus the subquery is dependent upon the outer query
  - Attempting to execute the subquery independent of the query proves this

```
SELECT NULL FROM City WHERE CountryCode = Code;
ERROR 1054 (42S22): Unknown column 'Code' in 'where clause'
```

Correlated Subqueries (2/2)

- Scope in correlated subqueries

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

  – Both the City table used in the outer query as well as the Country table used in the subquery contain a Name column

    - Column references are resolved using the nearest possible scope
    - The Name column can be resolved in the local scope of the parenthesized query expression

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

Rewriting IN to INNER JOIN (1/2)

- Subqueries that finds matches between tables often can be rewritten as a join

```
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

```
SELECT Name FROM Country INNER JOIN CountryLanguage
   ON Code = CountryCode WHERE Language = 'Spanish';
```

Rewriting IN to INNER JOIN (2/2)

- There are cases where the join query may return multiple copies of the same row as compared to the original query containing the IN subquery

```
SELECT Name FROM Country
   WHERE Code IN (SELECT CountryCode FROM CountryLanguage);
```

  - If converted to a join query the rows from the Country and CountryLanguage tables would be *combined*, causing each Country row to appear just as often as there are corresponding CountryLanguage rows
  - This can be corrected using the DISTINCT keyword to the SELECT list in order to eliminate the duplicate rows

```
SELECT DISTINCT Name FROM Country INNER JOIN CountryLanguage
   ON Code = CountryCode;
```

Rewriting NOT IN to an Outer Join

- It is possible to rewrite a subquery using NOT IN to a LEFT or RIGHT JOIN

```
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

```
SELECT City.* FROM City LEFT JOIN Country
   ON ID = Capital WHERE Capital IS NULL;
```

Limitations to Rewriting Subqueries to Joins (1/3)

- Aggregating Aggregates using FROM clause subqueries
  - A subquery in the FROM clause can be conveniently used to compute an aggregate
  - The result is essentially treated as just another table, allowing the outer query to again apply an aggregate function to the subquery result
  - Creating a hierarchical overview of countries

```
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) AS Countries
          GROUP BY Continent) AS Regions\G
```

Limitations to Rewriting Subqueries to Joins (2/3)

- Reporting aggregates of distinct child tables
  - Calculate multiple aggregates on different tables can be beneficial

```
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;
```

  - This type of query implies a (partial) Cartesian product, better yet ...

```
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;
```

Limitations to Rewriting Subqueries to Joins (3/3)

- Reporting aggregates of distinct child tables (cont.)
  - Use the original query design, but this time use pre-aggregated data via subqueries

```
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;
```

Further Practice: Chapter 14

- Comprehensive exercises

Chapter Summary

- 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

Course Content

Learning Objectives

- Define views
- List the reasons for using views
- Create a view
- Check a view
- Alter and remove a view
- Set privileges for views

What are Views? (1/2)

- View descriptions
  - Database Object Defined in Terms of a SELECT Statement
  - Virtual Table
  - Selected from Base Tables or Views
  - Updatable
- Benefits
  - Access to data becomes simplified
    - Can be used to perform a calculation and display its result
    - Can be used to select a restricted set of rows
    - Can be used for selecting data from multiple tables

What are Views? (2/2)

- Operations performed 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
  - Structure your tables to accommodate certain applications, a view can preserve the appearance of the original table structure to minimize disruption to other applications
  - Views can assist with structure changes that need to be made to tables to accommodate certain applications

The CREATE VIEW Statement

- Define a view
- General syntax

```
CREATE [OR REPLACE] [ALGORITHM = algorithm_type]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

- Optional parts of a CREATE VIEW statement
  - OR REPLACE
  - ALGORITHM
  - WITH CHECK OPTION

CREATE VIEW with SELECT (1/2)

- Example

```
CREATE VIEW CityView AS SELECT ID, Name FROM City;

SELECT * FROM CityView;
+------+-----------------------------------+
| ID   | Name                              |
+------+-----------------------------------+
|    1 | Kabul                             |
|    2 | Qandahar                          |
|    3 | Herat                             |
...
```

CREATE VIEW with SELECT (2/2)

- Column list examples

```
CREATE VIEW v AS SELECT Country.Name, City.Name
 FROM Country, City WHERE Code = CountryCode;
ERROR 1060 (42S21): Duplicate column name 'Name'

CREATE VIEW v AS SELECT Country.Name AS CountryName,
 City.Name AS CityName FROM Country, City WHERE Code = CountryCode;

CREATE VIEW v (CountryName, CityName) AS SELECT Country.Name,
 City.Name FROM Country, City WHERE Code = CountryCode;

CREATE VIEW CountryLangCount (Name, LanguageCount) AS
 SELECT Name, COUNT(Language) FROM Country, CountryLanguage
 WHERE Code = CountryCode GROUP BY Name;
```

Updatable Views (1/3)

- Can use UPDATE and DELETE
- Must be one-to-one relationship
- Updatability examples

```
CREATE VIEW EuropePop AS
 SELECT Name, Population FROM Country
 WHERE Continent = 'Europe';
Query OK, 0 rows affected (#.## sec)
```

Updatable Views (2/3)

- Showing the update

```
SELECT * FROM EuropePop WHERE Name = 'San Marino';
+------------+------------+
| Name       | Population |
+------------+------------+
| San Marino |      27000 |
+------------+------------+
1 row in set (#.## sec)

UPDATE EuropePop SET Population = Population + 1
 WHERE Name = 'San Marino';
Query OK, 1 row affected (#.## sec)
Rows matched: 1  Changed: 1  Warnings: 0

SELECT * FROM EuropePop WHERE Name = 'San Marino';
+------------+------------+
| Name       | Population |
+------------+------------+
| San Marino |      27001 |
+------------+------------+
1 row in set (#.## sec)
```

Updatable Views (3/3)

- Showing the update and delete

```
SELECT * FROM Country WHERE Name = 'San Marino';
+------------+------------+-----------+
| Name       | Population | Continent |
+------------+------------+-----------+
| San Marino |      27001 | Europe    |
+------------+------------+-----------+
1 row in set (#.## sec)

DELETE FROM EuropePop WHERE Name = 'San Marino';
Query OK, 1 row affected (#.## sec)

SELECT * FROM EuropePop WHERE Name = 'San Marino';
Empty set (#.## sec)

SELECT * FROM Country WHERE Name = 'San Marino';
Empty set (#.## sec)
```

LAB

Insertable Views (1/2)

- An Updateable View Can Be Insertable
  - Must meet additional requirements
  - No duplicate view column names
  - Must contain all columns from base table without default value
  - Cannot be derived columns
- A view that has a mix of simple column references and derived columns is not insertable
  - Updatable if updating only those columns that are not derived

```
CREATE VIEW v AS SELECT col1, 1 AS col2 FROM t;
```

- Updatable example

```
UPDATE v SET col1 = 0;
```

- Non-updatable example

```
UPDATE v SET col2 = 0;
```

Insertable Views (2/2)

- 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)
- Tables with AUTO_INCREMENT columns
  - 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()

WITH CHECK OPTION (1/2)

- Places constraint on allowable modifications
- Checks the WHERE conditions for updates
- Examples

```
CREATE VIEW LargePop AS
 SELECT Name, Population FROM Country
 WHERE Population >= 100000000
 WITH CHECK OPTION;
Query OK, 0 rows affected (#.## sec)

SELECT * FROM LargePop;
+--------------------+------------+
| Name               | Population |
+--------------------+------------+
| Bangladesh         |  129155000 |
| Brazil             |  170115000 |
...
| Russian Federation |  146934000 |
| United States      |  278357000 |
+--------------------+------------+
10 rows in set (#.## sec)
```

WITH CHECK OPTION (2/2)

- Update examples

```
UPDATE LargePop SET Population = Population + 1
 WHERE Name = 'Nigeria';
Query OK, 1 row affected (#.## sec)
Rows matched: 1  Changed: 1  Warnings: 0

SELECT * FROM LargePop WHERE Name = 'Nigeria';
+---------+------------+
| Name    | Population |
+---------+------------+
| Nigeria |  111506001 |
+---------+------------+
1 row in set (#.## sec)

UPDATE LargePop SET Population = 99999999
 WHERE Name = 'Nigeria';
ERROR 1369 (HY000): CHECK OPTION failed 'world.LargePop'
```

Checking Views

- Any object referenced by a view must exist
- Use CHECK TABLE command

```
CREATE TABLE t1 (i INT);
Query OK, 0 rows affected (#.## sec)

CREATE VIEW v AS SELECT i FROM t1;
Query OK, 0 rows affected (#.## sec)

RENAME TABLE t1 TO t2;
Query OK, 0 rows affected (#.## sec)

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)
```

Altering Views

- Changing the definition of an existing view
- Use ALTER VIEW statement
- Example

```
ALTER VIEW LargePop AS
SELECT Name, Population FROM Country
WHERE Population >= 100000000;
```

- Can also use CREATE VIEW to change a view

Dropping Views

- Deletes one or more views
- Use DROP VIEW statement
  - IF EXISTS clause
- Example

```
DROP VIEW IF EXISTS v1, v2;
Query OK, 0 rows affected, 1 warning (#.## sec)

SHOW WARNINGS;
+-------+------+---------------------------+
| Level | Code | Message                   |
+-------+------+---------------------------+
| Note  | 1051 | Unknown table 'world.v2' |
+-------+------+---------------------------+
1 row in set (#.## sec)
```

INFORMATION_SCHEMA

- VIEWS table in database
- Example

```
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
```

SHOW Statements (1/2)

- Display metadata
- SHOW CREATE VIEW specifically for views
- Example

```
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`
```

SHOW Statements (2/2)

- SHOW and DESCRIBE statements for views
  - DESCRIBE
  - SHOW TABLE STATUS
  - SHOW TABLES
  - SHOW FULL TABLES
- Example

```
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       |
+-------------------+------------+
```

Further Practice: Chapter 15

- Comprehensive exercises

Chapter Summary

- Define views
- List the reasons for using views
- Create a view
- Check a view
- Alter and remove a view
- Set privileges for views

Course Content

DEVELOPER I

1. INTRODUCTION
2. MySQL CLIENT/SERVER CONCEPTS
3. MySQL CLIENTS
4. QUERYING FOR TABLE DATA
5. HANDLING ERRORS AND WARNINGS
6. DATA TYPES
7. SQL EXPRESSIONS
8. OBTAINING METADATA
9. DATABASES
10. TABLES
11. MANIPULATING TABLE DATA
12. TRANSACTIONS

DEVELOPER II

13. JOINS
14. SUBQUERIES
15. VIEWS
16. PREPARED STATEMENTS
17. EXPORTING AND IMPORTING DATA
18. STORED ROUTINES
19. TRIGGERS
20. STORAGE ENGINES
21. OPTIMIZATION
22. CONCLUSION

Learning Objectives

- List the reasons for using prepared statements
- Using prepared statements with **`mysql`**
- Preparing, executing, and de-allocating prepared statements

Why Use Prepared Statements?

- Useful for running multiple similar queries
- Can ise same structure and change data values
- Enhanced performance
  - Statement parsed only once by server
  - May require fewer conversions
  - Less traffic between server and client

Prepared Statements from `mysql` (1/2)

- Aids in testing and debugging
- Session-bound
- User defined variables pass values from one statement to another
  - Connection specific
  - Also known as '@' variables
  - Use **SET** statement to define
  - Example syntax

```
SET @var_name = expr [, @var_name = expr] ...
```

  - Expression can evaluate to a integer, real, string or NULL value
  - Coercibility is implicit

Prepared Statements from `mysql` (2/2)

- Example

```
PREPARE my_stmt FROM
 'SELECT COUNT(*) FROM CountryLanguage WHERE CountryCode= ?';

SET @code = 'ESP'; EXECUTE my_stmt USING @code;
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
1 row in set (#.## sec)

SET @code = 'RUS'; EXECUTE my_stmt USING @code;
+----------+
| COUNT(*) |
+----------+
|       12 |
+----------+
1 row in set (#.## sec)

DEALLOCATE PREPARE my_stmt;
```

Preparing a Statement (1/3)

- PREPARE statement defines SQL for later execution
- Takes two arguments
  - Name
  - Statement text
- Use question mark (**?**) when data values are unknown

Preparing a Statement (2/3)

- Examples

```
PREPARE namepop FROM
 'SELECT Name, Population FROM Country WHERE Code = ?';
Query OK, 0 rows affected (#.## sec)
Statement prepared

PREPARE error FROM
 'SELECT NonExistingColumn FROM Country WHERE Code = ?';
ERROR 1054 (42S22): Unknown column 'NonExistingColumn' in
   'field list'
```

Preparing a Statement (3/3)

- Not all SQL statements can be prepared
- Limited to the following
  - **SELECT**
  - Data modification: **INSERT**, **REPLACE**, **UPDATE**, **DELETE**
  - **SET**, **DO** and many **SHOW** statements
- Several added with **5.1**
  - See list in guide
- Prepared statements exist per session only

Executing a Prepared Statement (1/2)

- Once prepared a statement can be executed
- **EXECUTE** and **USING** keywords
- Examples

```
PREPARE namepop FROM
 'SELECT Name, Population FROM Country
  WHERE Code = ?';
Query OK, 0 rows affected (#.## sec)
Statement prepared

SET @var1 = 'USA';
Query OK, 0 rows affected (#.## sec)
```

```
EXECUTE namepop USING @var1;
+---------------+------------+
| Name          | Population |
+---------------+------------+
| United States |  278357000 |
+---------------+------------+
1 row in set (#.## sec)
```

Executing a Prepared Statement (2/2)

- **EXECUTE** and **USING** examples *(continued)*

```
SET @var2 = 'GBR';
Query OK, 0 rows affected (#.## sec)

EXECUTE namepop USING @var2;
+----------------+------------+
| Name           | Population |
+----------------+------------+
| United Kingdom |   59623400 |
+----------------+------------+
1 row in set (#.## sec)

SELECT @var3 := 'CAN';
+----------------+
| @var3 := 'CAN' |
+----------------+
| CAN            |
+----------------+
1 row in set (#.## sec)
```

```
EXECUTE namepop USING @var3;
+--------+------------+
| Name   | Population |
+--------+------------+
| Canada |   31147000 |
+--------+------------+
1 row in set (#.## sec)

EXECUTE namepop USING @var4;
Empty set (#.## sec)
```

Deallocating a Prepared Statement

- Usually dropped automatically
- When explicit drop is required use the **DEALLOCATE PREPARE** Statement
  - **DROP PREPARE** can be used also
  - Reallocating the prepared statement will cause the previous prepared statement to be "dropped"
- Examples

```
DEALLOCATE PREPARE namepop;
Query OK, 0 rows affected (#.## sec)
```

Further Practice: Chapter 16

- Comprehensive exercises

Chapter Summary

- List the reasons for using prepared statements
- Using prepared statements with **`mysql`**
- Preparing, executing, and de-allocating prepared statements

Course Content

DEVELOPER I

1. INTRODUCTION
2. MySQL CLIENT/SERVER CONCEPTS
3. MySQL CLIENTS
4. QUERYING FOR TABLE DATA
5. HANDLING ERRORS AND WARNINGS
6. DATA TYPES
7. SQL EXPRESSIONS
8. OBTAINING METADATA
9. DATABASES
10. TABLES
11. MANIPULATING TABLE DATA
12. TRANSACTIONS

DEVELOPER II

13. JOINS
14. SUBQUERIES
15. VIEWS
16. PREPARED STATEMENTS
17. EXPORTING AND IMPORTING DATA
18. STORED ROUTINES
19. TRIGGERS
20. STORAGE ENGINES
21. OPTIMIZATION
22. CONCLUSION

Learning Objectives

- 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

Export Data Using SELECT/INTO OUTFILE (1/3)

- **`SELECT`** with **`INTO OUTFILE`**
  - Writes result set directly into a file
  - MySQL assumes filepath to be in database **data** directory, unless otherwise specified
- Example:

```
SELECT * INTO OUTFILE 'C:/City.txt' FROM City;
```

  - 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
...
```

Export Data Using SELECT/INTO OUTFILE (2/3)

- Utilizing with Windows
  - MySQL treats the backslash as the escape character in strings
  - Best to use '/' or as '\\'

```
SELECT * INTO OUTFILE 'C:\\City.txt' FROM City
```

- No path file identified
  - MySQL assumes that the file should be placed in the database data directory
- **INTO OUTFILE** changes **SELECT** operation
  - File written to server host, instead of over the network to client
  - Causes server to write a new file on the server host
  - File is created with filesystem access permissions
  - File contains one line per row select by the statement

Using Data File Format Specifiers (1/2)

- **SELECT** with **INTO OUTFILE** default specifiers
  - Assumes Tab delimited and newline terminators
- Can change specifiers for all columns
  - Syntax

```
FIELDS
    TERMINATED BY 'string'
    ENCLOSED BY 'char'
    ESCAPED BY 'char'
LINES TERMINATED BY 'string'
```

  - **FIELDS** clause defines data values within a line
  - **LINES** clause indicate where record boundaries occur
  - **TERMINATED BY**, **ENCLOSED BY**, **ESCAPED BY** use defaults if not specified

Using Data File Format Specifiers (2/2)

- Terminator definitions
- Line terminator specifiers
  - Newline character is the default
- CSV format text file example

```
SELECT * INTO OUTFILE 'C:/City.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\r' FROM City;

...
"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"
...
```

Sequence	Meaning
`\N`	NULL value
`\0`	NUL (zero) byte
`\b`	Backspace
`\n`	Newline (linefeed)
`\r`	Carriage return
`\s`	Space
`\t`	Tab
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash

Import LOAD DATA INFILE (1/4)

- Use **LOAD DATA INFILE**
- Example

```
LOAD DATA INFILE 'C:/City.txt' INTO TABLE City;
```

- Similar clauses and format specifiers as **SELECT...INTO OUTFILE**
- MySQL assumes file is located on server host
  - In database data directory

Import LOAD DATA INFILE (2/4)

- Tab delimited or comma separated files
- Characteristics to know about input file
- CSV example

```
LOAD DATA INFILE 'C:/City.txt' INTO TABLE City
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

Import LOAD DATA INFILE (3/4)

- Specifying data file location as client host

```
LOAD DATA LOCAL INFILE 'C:/City.txt' INTO TABLE City;
```

- Skipping or transforming column values

```
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
```

  – Reduced from original 2233 rows

Import LOAD DATA INFILE (4/4)

- **LOAD DATA INFILE** and duplicate records
  - Can control duplicate records with **INSERT** and **REPLACE**
  - Behavior differs slightly
  - **IGNORE** and **REPLACE** are mutually exclusive
- Information provided by **LOAD DATA INFILE**

```
Records: 174  Deleted: 0  Skipped: 3  Warnings: 14
```

  - Records -- number of input records
  - Deleted -- number of records replaced
  - Skipped -- number of records ignored
  - Warnings -- number of problems found in input file

Export with 'mysqldump' (1/3)

- MySQL utility to export (dump) table contents
  - Full structure
  - Data only
  - Table structure only
  - In standard format
  - MySQL specifics for optimized speed
  - Compressed
- Three ways to invoke **mysqldump**:

```
shell> mysqldump [options] db_name [tables]
shell> mysqldump [options] --databases db_name1
   [db_name2 db_name3...]
shell> mysqldump [options] --all-databases
```

Export with 'mysqldump' (2/3)

- Export a database …

```
mysqldump world
```

- Export multiple tables …

```
mysqldump world City Country
```

- Export multiple databases …

```
mysqldump –all-databases (or –A)
mysqldump --databases world db2
```

If you do not specify a table name, the entire database will be dumped.

Export with 'mysqldump' (3/3)

- Export to a text file using the redirect operator

```
mysqldump -uroot -p<password> world > C:/world_dump.sql
```

- File contains commands needed to recreate tables and data
- Export to a specific table in a database

```
mysqldump -uroot -p<password> world CountryLanguage
 > C:/CountryLanguage.sql
```

Existing files with the same name will be overwritten.

Import with 'mysqlimport' (1/3)

- MySQL utility to load data files into tables
- Command line interface to **`LOAD DATA INFILE`**
- General syntax:

```
shell> mysqlimport options db_name input_file ...
```

- Matches file name with table name
- Tables must already exist

Import with 'mysqlimport' (2/3)

- `mysqlimport` options

  `--help`

  `--lines-terminated-by=`*string*

  `--fields-terminated-by=`*string*

  `--fields-enclosed-by=`*char*

  `--ignore` or `--replace`

  `--local`

Import with 'mysqlimport' (3/3)

- Examples:

```
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
```

Import with MySQL Command Files

- Import table data
  - Example

```
shell> mysql -u root world < CountryLanguage.sql
```

- Loading the data
  - Examples

```
SOURCE C:/CountryLanguage.sql
```

Default is to *not* write over existing database file.

Further Practice: Chapter 17

- Comprehensive exercises

Chapter Summary

- 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

Course Content

Learning Objectives

- 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

What is a Stored Routine?

- Set of SQL statements that can be stored in server
- Types
  - Stored procedures
    - A procedure is invoked using a **CALL** statement, and can only pass back values using output variables
  - Stored functions
    - A function can be called from inside a statement and can return a scalar value

Stored Routine Uses

- Client applications
  - One application
  - One programming language
- Security
  - Minimal data access
  - Single location processing
- Performance
- Function libraries

Stored Routine Issues

- Increased server load
- Limited development tools
- Limited language functionality and speed
- Limited debugging/profiling capabilities

Creating Stored Routines

- Create procedure

```
CREATE PROCEDURE procedure_name procedure_statement
```

  – Single statement example

```
CREATE PROCEDURE world_record_count ()
BEGIN
  SELECT 'country count ', COUNT(*) FROM Country;
END//
```

- Create function

```
CREATE FUNCTION function_name RETURNS return_type
    function_statement
```

  – Single parameter example

```
CREATE FUNCTION ThankYou (s CHAR(20)) RETURNS CHAR(50)
RETURN CONCAT('Thank You, ',s,'!');
```

LAB

Compound Statements

- **DELIMITER**
- **BEGIN ... END**

```
DELIMITER //
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//
DELIMITER ;
```

Blocks can be nested.
And can have labels like;
WHILE/REPEAT/LOOP

Assign Variables

- **DECLARE**
  - Declaring
  - Scope

```
DELIMITER //
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//
DELIMITER ;
```

Assign Variables
SELECT ... INTO

- **SELECT ... INTO**
  - Session variables

```
SELECT SUM(population) FROM country INTO @WorldPop;
```

... is equivalent to ...

```
SELECT SUM(population) INTO @WorldPop FROM country;
```

  - Local variables (**DECLARE** statement recommended)

```
SELECT COUNT(*) FROM city INTO Total_Cities;
```

... is equivalent to ...

```
SELECT COUNT(*) INTO Total_Cities FROM city;
```

Assign Variables
SET

- **SET**
  - The **SET** statement allows the user to assign a value to a user defined variable using either **=** or **:=** as the assignment operator

```
DELIMITER //
CREATE FUNCTION final_bill
  (total_charge FLOAT(9,2), tax_rate FLOAT (3,2))
RETURNS FLOAT(10,2)
BEGIN
  DECLARE Fbill FLOAT(10,2);
  SET Fbill=total_charge + total_charge * tax_rate;
  RETURN Fbill;
END//
DELIMITER ;
```

Variable Scope

Recommendation:
Use different prefixes for
Different variable types.

- Local variable
- Routine parameter
- Local variable in an *inner* block
- Local variable in an *outer* block

```
DELIMITER //
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//
DELIMITER ;
```

Parameter Declarations

- Stored procedures
  - **`IN`** (Default)
    - Indicates an input parameter which is passed in from the caller to the procedure
  - **`OUT`**
    - Indicates an output parameter which is set by the procedure and passed to the caller after the procedure terminates
  - **`INOUT`**
    - Indicates a parameter that can act as an **`IN`** and an **`OUT`** parameter
- Stored functions
  - Stored functions can not have **`OUT`** or **`INOUT`** parameters. Consequently it is neither necessary nor allowed to use the **`IN`** keyword for function parameter declarations

Execute Stored Routines

- Executing procedures
- Executing functions
- Implications of database association
  - USE *Database*
  - Qualify names
  - Routines deleted when database deleted
- **SELECT** statements
  - Stored procedures only
  - Result set sent directly to client

Examine Stored Routines

- **SHOW CREATE PROCEDURE** / **FUNCTION**
  - MySQL specific
  - Returns exact code string
- **SHOW PROCEDURE** / **FUNCTION STATUS**
  - MySQL specific
  - Returns characteristics of routines
- INFORMATION_SCHEMA.ROUTINES
  - Standard SQL
  - Returns a combination of the **SHOW** commands

Delete Stored Routines

- **DROP PROCEDURE**
  - Syntax

```
DROP PROCEDURE [IF EXISTS] procedure_name
```

  - Example

```
DROP PROCEDURE proc_1;
```

- **DROP FUNCTION**
  - Syntax

```
DROP FUNCTION [IF EXISTS] function_name
```

  - Example

```
DROP FUNCTION IF EXISTS func_1;
```

Flow Control Statements

- Statements and constructs that control order of operation execution
- Common flow controls
  - Choices
    - **`IF`** and **`CASE`**
  - Loops
    - **`REPEAT`**, **`WHILE`** and **`LOOP`**

IF

- The most basic of all choice flow controls or conditional constructs

```
IF (test condition) THEN
...
ELSEIF (test condition) THEN
...
ELSE
...
END IF
```

CASE

- **CASE** provides a means of developing complex conditional constructs
- **CASE** works on the principle of comparing a given value with specified constants and acting upon the first constant that is matched

```
CASE case_value
  WHEN when_value THEN
  ...
  ELSE
  ...
END CASE
```

OR ...

```
CASE
  WHEN test_condition THEN
  ...
  ELSE
  ...
END CASE
```

REPEAT

- The **REPEAT** statement repeats the statements between the **REPEAT** and **UNTIL** keywords until the condition after the **UNTIL** keyword becomes **TRUE**
- A **REPEAT** loop always iterates at least once
- Optional Labels
  – Begin
  – End

```
my_label: REPEAT
...
UNTIL test_condition
END REPEAT my_label;
```

WHILE

- **WHILE** repeats the statements between the **DO** and **END WHILE** keywords as long as the condition appearing after the **WHILE** keyword remains **TRUE**
- A **WHILE** loop may never iterate (if the condition is initially **FALSE**)

```
my_label: WHILE test_condition DO
...
END WHILE my_label;
```

LOOP

- The statements between the **LOOP** and **END LOOP** keywords are repeated.
- The loop must be explicitly exited, and usually this is accomplished with a **LEAVE** statement.
- A valid label must appear after the **LEAVE** keyword.

```
my_label: LOOP
   ...
   LEAVE my_label;
END LOOP my_label;
```

Other Flow Control Constructs

- **LEAVE**
  - This statement is used to exit any labeled construct
  - **LEAVE** can may be used to exit **BEGIN ... END** compound statements as well as the various loops (**LOOP**, **REPEAT** and **WHILE**) as long a the block is labeled.
- **ITERATE**
  - This statement simply means “do the **LOOP** (or **REPEAT** or **WHILE**) again”

DECLARE Statement Syntax

- The **DECLARE** Statement Defines Items Local to a Routine
  - Local variables
  - Conditions and handlers
  - Cursors
- **DECLARE** Allowed Only Inside a **BEGIN ... END**
- Declarations Must Follow a Specific Order
  - Variables
  - Conditions
  - Cursors
  - Handlers

DECLARE must be Immediately following the BEGIN, before any other statements.

DECLARE CONDITION

- **DECLARE CONDITION**

```
DECLARE condition_name CONDITION FOR condition_value;
```

  - SQLSTATE Condition Value

```
DECLARE null_not_allowed CONDITION FOR SQLSTATE '23000';
```

  - MySQL Error Code Condition Value

```
DECLARE null_not_allowed CONDITION FOR 1048;
```

DECLARE HANDLER

- **`DECLARE CONTINUE HANDLER`**

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000'  SET @x = 1;
```

- **`DECLARE EXIT HANDLER`**
- Additional Condition Values
  - Declared Conditions
  - SQLWARNING
  - NOT FOUND
  - SQLEXCEPTION

Cursors (1/3)

- A Control Structure within Stored Routines for Record Retrieval
  - One row at a time
- Cursor is Short for **CUR**rent **S**et **O**f **R**ecords
- Mostly Used in Loops That Fetch and Process Rows
- Asensitive
- Read-Only
- Non-Scrolling

Cursors (2/3)

- Declaring cursors and handlers
  - Cursors must be declared before declaring handlers

```
DECLARE cursor_name CURSOR FOR select_statement;
```

  - Handling end of records

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
   SET done = 'yes'
```

- **OPEN**
  - Opens a previously declared cursor

Cursors (3/3)

- **FETCH**
  - Obtains the next row using the specified open cursor, and advances the cursor pointer
  - When there is no next row, an error will result
- **CLOSE**
  - 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

Cursor Limitations

- Noteworthy Limitations
  - Read-only
  - Updatable cursors are not supported
  - Asensitive
  - Non-scrollable
  - Not named
  - Only cursor per prepared statement
  - Statement result must be in prepared mode
  - Work on a row base
  - Cannot skip rows

Further Practice: Chapter 18

- Incremental exercises

Chapter Summary

- 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

Course Content

Learning Objectives

- Describe triggers
- Create new triggers
- Delete existing triggers

What Are Triggers?

- Named database objects
- Activated when table data is modified
- Bring a level of power and security to table data
- Trigger scenario using the world database
  - What would you do after changing the Country table code column?
  - Since the code is stored in all three world database tables, it is best to change all 3 at once
  - A trigger can accomplish this task
- Trigger features

Creating Triggers

- Syntax

```
CREATE TRIGGER trigger_name
  { BEFORE | AFTER }
  { INSERT | UPDATE | DELETE }
  ON table_name
  FOR EACH ROW
  triggered_statement
```

- Example

```
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)
```

Trigger Events (1/2)

- Before Time
  - `BEFORE INSERT`
  - `BEFORE UPDATE`
  - `BEFORE DELETE`
- After Time
  - `AFTER INSERT`
  - `AFTER UPDATE`
  - `AFTER DELETE`

Trigger Events (2/2)

- Trigger procedure
  - Create and Use the CityAd Table

1) First, check to see if a trigger of this type/name already exists.
2) A table should be created to contain the data collected from the trigger.
3) Create the actual trigger, using the appropriate name according to the type of event.
4) Confirm that the City_AD trigger now exists.
5) Perform a query to confirm the existence of trigger data.
6) Perform a **DELETE** that will create results to be placed in the DeletedCity trigger table.
7) Perform another query to confirm the *deletion* of the same trigger data.
8) Perform a query on the City_AD trigger table. Notice that Dallas has been added to the table.

Trigger Error Handling

- MySQL handles errors during trigger execution as follows:
  - Failed **BEFORE** triggers
    - Operation on corresponding row is not performed
  - **AFTER** trigger execution
    - **BEFORE** trigger events and the row operation must execute successfully
  - Transactional tables
    - Rollback of all changes made by the statement

Delete Triggers

- **DROP TRIGGER**

```
DROP TRIGGER trigger_name;

DROP TRIGGER schema_name.trigger_name;
```

If you drop a table, the triggers are automatically dropped also.

Restrictions on Triggers

- Disallowed statements
  - SQL prepared statements
  - Explicit or implicit **`COMMIT`** or **`ROLLBACK`**
  - Return a result set
  - **`FLUSH`**
  - Recursive

Further Practice: Chapter 19

- Comprehensive exercises

Chapter Summary

- Describe triggers
- Create new triggers
- Delete existing triggers

Course Content

DEVELOPER I

DEVELOPER II

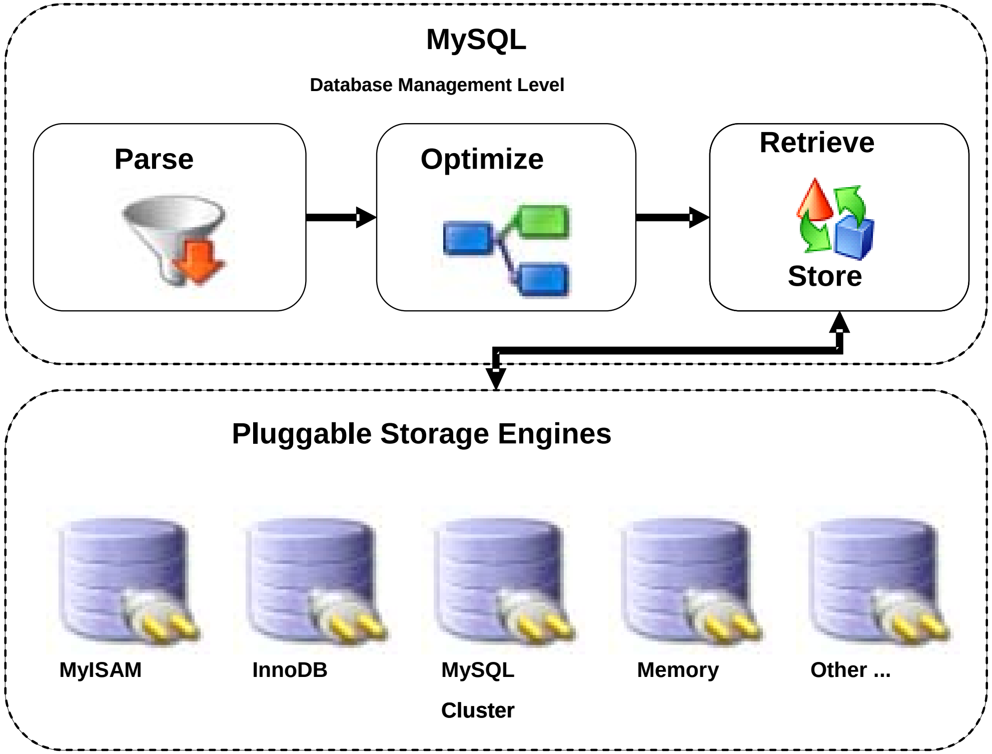
Learning Objectives

- 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

SQL Parser and Storage Engines

- Client sends requests to the server as SQL
- Two-tier processing
  - Upper tier includes SQL parser and optimizer
  - Lower tier comprises a set of storage engines
- SQL tier free of dependency on storage engine
  - Engine setting does not effect processing
  - Some Exceptions

Storage Engine Breakdown (1/2)



Storage Engine Breakdown (2/2)

- Storage medium
- Transactional capabilities
- Locking
- Backup and recovery
- Optimization
- Special features
- MySQL server operates same for all storage engines
  - SQL commands independent of engine

Storage Engines and MySQL

- Can choose specific storage engine when creating a table
- Best fit for your application
- Each have different characteristics and implications

Available Storage Engines

- MySQL provides and maintains several storage engines
- Also compatible with many third party engines
- MySQL developed
  - MyISAM
  - MEMORY
  - BLACKHOLE
  - Falcon
  - ARCHIVE
  - CSV
  - NDB/Cluster
- Third party engines
  - InnoDB
  - InfoBright- BrightHouse
  - PBXT
  - solidDB
  - Nitro

Common Storage Engines

- MyISAM
  – Fast
  – Data stored in table
  – Table-level locking
- InnoDB
  – Transactional
  – Foreign keys
  – Row-leveling locking
  – Backups
- Memory
  – Data is in memory ONLY

Storage Engines Available on Server

- View available storage engines…

```
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
...
```

Setting the Storage Engine

- Specify engine using **CREATE TABLE**
  - MySQL uses system default engine if not specified
- Change engine for existing table with **ALTER TABLE**
- Examples

```
CREATE TABLE t (i INT) ENGINE = InnoDB;
ALTER TABLE t ENGINE = MEMORY;
```

Displaying Storage Engines (1/3)

- View current engine setting
  - SHOW CREATE TABLE
  - SHOW TABLE STATUS

- Examples

```
SHOW CREATE TABLE City\G
********************** 1. row ***********************
   Table: CityCreate 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)
```

Displaying Storage Engines (2/3)

- Examples *(continued)*

```
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
   ...
```

Displaying Storage Engines (3/3)

- Using INFORMATION_SCHEMA

```
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)
```

- Table management is engine independent
- Storage engine implementation “Lower Tier”
- Knowing the engine can enable efficient use
- Engine must be compiled and enabled

The MyISAM Storage Engine

- MyISAM is the MySQL Default
- Manages tables with specific characteristics
  - Represented by three files
  - Most Flexible AUTO_INCREMENT
  - Fast, compressed, read-only tables save space
  - Manages contention between queries
  - Portable storage format
  - Specify number of rows for a table
  - Disable updating of non-unique indexes and enable the indexes
  - Tables take up very little space

MyISAM Row Storage Formats

- Three row storage formats
  - Fixed-row format
  - Dynamic-row format
  - Compressed format

Compressing MyISAM Tables

- Tables must be deliberately compressed
- Compressed tables are read-only
  - Tables must be decompressed to modify
- Using the **myisampack** utility
  - Includes a mixture of “True” compression and a set of optimizations
  - Each record compressed separately with small cost to decompress
- Use **myisamchk** afterward to update the indexes
- Always backup tables prior to running utilities

MyISAM Locking (1/2)

- Table level locking
- Acquiring locks
- Tables with *no* holes support concurrent inserts
- Tables *with* holes do not support concurrent inserts by default
- Can change priority of statements that retrieve or modify data
- Write request not processed until current readers finished

MyISAM Locking (2/2)

- Scheduling modifiers for changing request priority
  - **LOW_PRIORITY** for updating tables
  - **HIGH_PRIORITY** with **SELECT** to move ahead
  - **DELAYED** may be used with **INSERT** and **REPLACE**
- Scheduling modifiers for changing query priority
  - **SELECT HIGH PRIORITY** moves ahead of **INSERT**
  - **LOW_PRIORITY** or **DELAYED** reduces logging priority
- There are many points to keep in mind when using **DELAYED**

The InnoDB Storage Engine

- Manages tables with specific characteristics
  - Represented on disk by a `.frm` format file as well as data and index storage
  - Supports transactions
  - ACID compliant
  - Auto-recovery after a crash
  - MVCC and non-locking reads
  - Supports foreign keys and referential integrity
  - Supports consistent and online logical backup

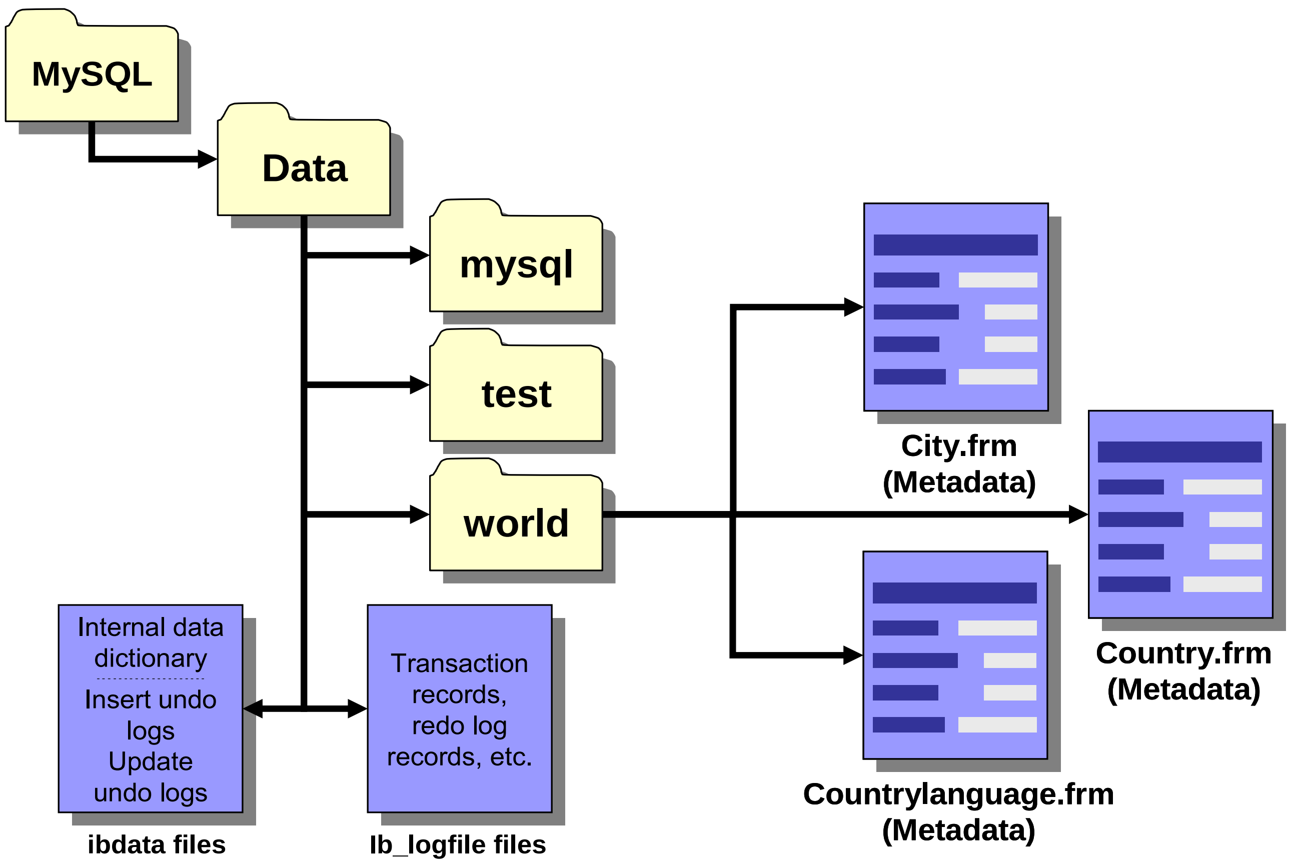
The InnoDB Tablespace and Logs (1/2)

- Tablespace for storing table contents
- Log files for recording transaction activity
- Format file (`.frm`)
- Logical storage area can contain multiple files
- Table-specific file (`.ibd`)

  `--innodb-file-per-table`
- Manages InnoDB-specific log files
- Log files used for auto-recovery

The InnoDB Tablespace and Logs (2/2)

- File locations



The InnoDB ACID Compliance and Locking

- Satisfies ACID conditions
- General locking properties
  - Does not need to set locks to achieve consistent reads
  - Uses row-level locking per concurrency properties
  - May acquire row locks as necessary to improve concurrency
  - Deadlock is possible
- Supports two locking modifiers
  - Convert non-locking into locking reads
  - `LOCK IN SHARE MODE` places a shared lock on each selected row
  - `FOR UPDATE` places an exclusive lock on selected rows
- REPEATABLE READ isolation level allows modifiers

The MEMORY Storage Engine

- Uses tables stored in memory
  - Tables are temporary
- Fixed-length rows
- Manages tables with specific characteristics
- Formerly HEAP engine
- MEMORY indexing options
  - Uses HASH indexes by default
  - BTREE is preferrable for some operators

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

Other Storage Engines

- Optional storage engines
- Select when configuring MySQL
- Many more engines available
  - Falcon
  - NDB
  - EXAMPLE
  - ARCHIVE
  - CSV
  - BLACKHOLE

To reduce memory Usage, do not configure unneeded Storage engines into the server.

Further Practice: Chapter 20

- Comprehensive exercises

Chapter Summary

- 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

Course Content

Learning Objectives

- 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

Overview of Optimization Principles

- Why be concerned about optimization?
  - Server processes queries more efficiently and performs better
- Several optimization strategies
  - Use indexing properly
  - Well-written queries
  - Generating summary tables
  - Choose best matching storage engine

Using Indexes for Optimization

- Large tables require indexing for efficiency
- Benefits of indexes
  - Contain sorted values
  - Use less disk I/O
  - Enforce uniqueness constraints
- Downside of indexing
  - Uses additional space
  - Can slow down some data manipulations

Types of Indexes (1/2)

- Four general types
  - PRIMARY KEY
  - FOREIGN KEY
  - UNIQUE
  - NON-UNIQUE
- Specialized types
  - FULLTEXT
  - SPATIAL

Types of Indexes (2/2)

```
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'
```

	MyISAM	MEMORY	InnoDB
Indexing	B-tree / Full text / R-tree	Hash / B-tree	B-tree

Creating Indexes (1/4)

- Create indexes with table creation
- Table *without* index

```
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);
```

Creating Indexes (2/4)

- Table *with* index

```
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));
```

- Table with composite index

```
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));
```

Creating Indexes (3/4)

- Table with multiple indexes

```
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));
```

- Table with **UNIQUE INDEX**

```
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));
```

Creating Indexes (4/4)

- Primary key versus unique index
  - Primary key cannot contain NULL
  - Only one primary key is allowed per table
  - Primary key is a type of unique index
  - Unique Index is not always a primary key

Naming Indexes

- Include name just before column list

```
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));
```

- Default name
- Primary key always named **PRIMARY**

Adding Indexes to Existing Tables (1/2)

- Use **ALTER TABLE** or **CREATE INDEX** statements
- **ALTER TABLE** examples
  - Adding Indexes to **HeadOfState** table (created without index)

```
ALTER TABLE HeadOfState ADD PRIMARY KEY (ID);

ALTER TABLE HeadOfState ADD INDEX (LastName,FirstName);

ALTER TABLE HeadOfState ADD PRIMARY KEY (ID),
    ADD INDEX (LastName,FirstName);
```

Adding Indexes to Existing Tables (2/2)

- **CREATE INDEX** examples
  - Must provide name for index
  - Only single index per statement

```
CREATE UNIQUE INDEX IDIndex ON HeadOfState (ID);

CREATE INDEX NameIndex ON HeadOfState (LastName,FirstName);
```

- **Using index prefixes**
  - Several column types
  - Use only specified, leading part of column values
  - Example

```
CREATE INDEX part_of_name ON customer (name(10));
```

Dropping Indexes

- With **ALTER TABLE**, use a **DROP** clause and name the index to be dropped
  - Dropping a **PRIMARY KEY** is easy

```
ALTER TABLE HeadOfState DROP PRIMARY KEY
```

  - To drop another kind of index, you must specify its name

```
ALTER TABLE HeadOfState DROP INDEX NameIndex;
```

  - Adding an index back after it is dropped

```
ALTER TABLE HeadOfState
  ADD INDEX NameIndex (LastName, FirstName);
```

- **DROP INDEX** examples

```
DROP INDEX NameIndex ON t;
DROP INDEX `PRIMARY` ON t;
```

Principles for Index Creation

- Use NOT NULL if possible
- Over-indexing slows down table updates
- Estimates whether indexing is efficient
- Choose unique and non-unique indexes appropriately
- Index column prefix rather than entire column
- Avoid creating multiple indexes
- Optimize index creation process
- Determine whether hash or tree indexes are better

Indexing Column Prefixes (1/2)

- Short index values processed more quickly

- Examples

```
CREATE TABLE t
(
    name CHAR(255),
    INDEX (name)
);
```

...and with prefix length...

```
CREATE TABLE t
(
    name CHAR(255),
    INDEX (name(15))
);
```

Indexing Column Prefixes (2/2)

- Determine the number of duplicate indexes

```
SELECT
    COUNT(*) AS 'Total Rows',
    COUNT(DISTINCT name) AS 'Distinct Values',
    COUNT(*) - COUNT(DISTINCT name) AS 'Duplicate Values'
FROM t;
```

...and with prefix length...

```
SELECT
    COUNT(DISTINCT LEFT(name,n)) AS 'Distinct Prefix Values',
    COUNT(*) - COUNT(DISTINCT LEFT(name,n)) AS 'Duplicate
   Prefix Values'
FROM t;
```

Leftmost Index Prefixes (1/2)

- Used for composite index
- Consists of one or more of the initial columns
- Composite indexes allow quick lookup

Leftmost Index Prefixes (2/2)

- Display composite index information

```
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
    Sub_part: NULL
      Packed: NULL
        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
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
```

FULLTEXT Indexes (1/3)

- MySQL supports full-text indexing and searching
  - **FULLTEXT** index type
  - Only supported by MyISAM
  - Definition can be given within **CREATE TABLE**, **ALTER TABLE**, **CREATE INDEX**
  - Create index after table creation for large datasets
- Use **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
  }
```

FULLTEXT Indexes (2/3)

- Three types of FULLTEXT searches
  - Boolean, Natural Language and Query Expansion
- Example table

```
CREATE TABLE books (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  author VARCHAR(64),
  title VARCHAR(128),
  FULLTEXT (title)
) ENGINE=MyISAM;
```

FULLTEXT Indexes (3/3)

- Examples

```
SELECT title FROM books WHERE MATCH(title)
 AGAINST('prince');
+-------------------------------------------+
| title                                     |
+-------------------------------------------+
| The Little Prince                         |
| Harry Potter and the Half-Blood Prince    |
+-------------------------------------------+
2 rows in set (#.## sec)

SELECT title, author FROM books WHERE MATCH(title)
 AGAINST('green +Anne' IN BOOLEAN MODE);
+-----------------------+--------------------+
| title                 | author             |
+-----------------------+--------------------+
| Anne of Green Gables  | Kathleen Olmstead  |
+-----------------------+--------------------+
1 row in set (#.## sec)
```

Using EXPLAIN to Analyze Queries

- Use **EXPLAIN** to determine query processing
- Returns useful information
  - Shows if index is required
  - Shows if index is being used
  - Analyzes query rewrites
- Can use **EXPLAIN** with **SELECT**
  - And indirectly with **UPDATE** and **DELETE**

How EXPLAIN Works (1/2)

- Place **EXPLAIN** keyword in front of **SELECT**

- Examples

```
EXPLAIN SELECT * FROM t WHERE YEAR(d) >= 1994\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 867038
        Extra: Using where
```

How EXPLAIN Works (2/2)

- Examples *(continued)*

```
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: 70968
        Extra: Using where
```

EXPLAIN Output Columns (1/2)

- Produces one row of output for each table
- Meaning of the output columns
  - `id`
  - `select_type`
  - `table`
  - `type`

EXPLAIN Output Columns (2/2)

- Output columns *(continued)*
  - **possible_keys**
  - **Key**
  - **key_len**
  - **ref**
  - **rows**
  - **Extra**

EXPLAIN for Joins (1/2)

- Joins tend to increase amount of server processing
- **`EXPLAIN`** can help reduce server impact
- The **`type`** column indicates the join type
- **`type`** column output
  - **`system`**
  - **`const`**
  - **`eq_ref`**
  - **`ref`**

EXPLAIN for Joins (2/2)

- **`type`** column output *(continued)*
  - `ref_or_null`
  - `index_merge`
  - `unique_subquery`
  - `index_subquery`
  - `range`
  - `index`
  - `ALL`

EXPLAIN for Table Processing (1/2)

- The `Extra` column provides process information
  - Can indicate an efficient or inefficient query
- Efficient query `Extra` output
  - `Using index`
  - `Where used`
  - `Distinct`
  - `Not exists`

EXPLAIN for Table Processing (2/2)

- Inefficient query **`Extra`** output
  - **`Using filesort`**
  - **`Using temporary`**
  - **`Range checked for each record`**
- Rewrite query and run **`EXPLAIN`** again, if needed

Query Rewriting Techniques (1/2)

- Efficiency principles
  - No indexed columns within an expression

    Bad Practice `SELECT * FROM t WHERE YEAR(d) >= 1994;`

    Good Practice `SELECT * FROM t WHERE d >= '1994-01-01';`

  - Beneficial for joins that compare columns from two tables

```
SELECT * FROM Country JOIN CountryLanguage
   ON Country.Code = CountryLanguage.CountryCode;
```

  - Use same value as column data type

    Recommended `WHERE id = 18`

    Good Practice `WHERE id = '18'`

Query Rewriting Techniques (2/2)

- Efficiency principles *(continued)*
  - Pattern matching

```
WHERE name LIKE 'de%'
WHERE name >= 'de' AND name < 'df'
WHERE name LIKE '%de%'
```

  - Rewrite the query and use a trigger that maintains the additional column

```
WHERE LENGTH(column)=5
```

...rewrite as...

```
WHERE column_length=5
```

Optimizing Queries by Limiting Output

- Reduce amount of output a query produces
- Use the **LIMIT** clause
  - Reduces information going over the network
  - Allows server to terminate query processing earlier

```
SELECT * FROM t LIMIT 10;
```

- Use the **WHERE** clause

Good Practice `SELECT * FROM Country WHERE Name LIKE 'M%';`

Recommended `SELECT Name FROM Country WHERE Name LIKE 'M%';`

- More improvement with index or column
  - `'Name' in the above example is better as an index`

Using Summary Tables (1/4)

- Select records to generate summaries
- Summary table strategy
- Several benefits to this strategy
- Table-locking table will be available more
- Consider making a MEMORY table

Using Summary Tables (2/4)

- Creating a summary table…

```
CREATE TABLE ContinentGNP
 SELECT Continent, AVG(GNP) AS AvgGNP
 FROM Country GROUP BY Continent;

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 |
+---------------+---------------+
```

Using Summary Tables (3/4)

- Compare summary table to original table…

```
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 |
| Asia      | Maldives      |     199.00 |   150105.725490 |
| Asia      | Mongolia      |    1043.00 |   150105.725490 |
| Europe    | Andorra       |    1630.00 |   206497.065217 |
| Europe    | Faroe Islands |       0.00 |   206497.065217 |
| Europe    | Gibraltar     |     258.00 |   206497.065217 |
...
```

Using Summary Tables (4/4)

- Disadvantages of summary tables
  - Values are only good until changed
  - Storing data twice

Optimizing Updates (1/2)

- Techniques for updating tables
  - Use **DELETE** and **UPDATE** same as **SELECT**
  - **EXPLAIN** can be use with **SELECT**
  - Multi-row **INSERT**

```
INSERT INTO t (id, name) VALUES(1,'Bea');
INSERT INTO t (id, name) VALUES(2,'Belle');
INSERT INTO t (id, name) VALUES(3,'Bernice');
```

...or...

```
INSERT INTO t (id, name)
 VALUES (1,'Bea'),(2,'Belle'),(3,'Bernice');
```

Optimizing Updates (2/2)

- Techniques for updating tables *(continued)*
  - Better Performance with a transaction

```
START TRANSACTION;
INSERT INTO t (id, name) VALUES(1,'Bea');
INSERT INTO t (id, name) VALUES(2,'Belle');
INSERT INTO t (id, name) VALUES(3,'Bernice');
COMMIT;
```

- **`LOAD DATA INFILE`** faster than multi-row **`INSERT`**
- Use **`REPLACE`** rather than **`DELETE`** plus **`INSERT`**

Choosing Appropriate Storage Engines (1/2)

- Decide query types during table creation
- Choose storage engine with locking level needed
- InnoDB good for a mix of retrievals and updates
- MyISAM table structure dependent on the higher priority between speed or disk
  - Choose fixed-length or variable-length according to need
  - Use read-only tables

Choosing Appropriate Storage Engines (2/2)

- CHAR columns take more space than VARCHAR
  - No speed advantage for InnoDB as for MyISAM
- MEMORY for temporary data

Chapter Summary

- 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

Course Content

DEVELOPER I

1. INTRODUCTION
2. MySQL CLIENT/SERVER CONCEPTS
3. MySQL CLIENTS
4. QUERYING FOR TABLE DATA
5. HANDLING ERRORS AND WARNINGS
6. DATA TYPES
7. SQL EXPRESSIONS
8. OBTAINING METADATA
9. DATABASES
10. TABLES
11. MANIPULATING TABLE DATA
12. TRANSACTIONS

DEVELOPER II

13. JOINS
14. SUBQUERIES
15. VIEWS
16. PREPARED STATEMENTS
17. EXPORTING AND IMPORTING DATA
18. STORED ROUTINES
19. TRIGGERS
20. STORAGE ENGINES
21. OPTIMIZATION
22. CONCLUSION

Learning Objectives

- 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

Training and Certification Website

http://www.mysql.com/training

- Overview
- Certification
- Curriculum Paths
- Catalog
- Schedule
- Delivery Options
- Savings and Promotions
- Training Bundles
- Training Partners
- Testimonials
- FAQ

We Need Your Evaluation!

- Please Take Time to Give us Your Opinions
  - http://www.mysql.com/training/evaluation.php
  - Get course code from instructor

THANK YOU!!!

- We Appreciate Your Attendance and Participation!
- Contact Us Regarding Training Issues @...
  - **Website: http://www.mysql.com/training/**
  - **Email: training@mysql.com**
  - **Phone: USA Toll Free: 1-866-697-7522**
    Worldwide: 1-208-514-4780

Q&A Session

- Question and Answers
- More Questions After Class?
  - Get answers on our Reference Manual FAQ online
  - http://dev.mysql.com/doc/refman/5.1/en/faqs.html
- Want To Do the Labs On Your Own?
  - Download the '**world**' database from our website
  - http://dev.mysql.com/doc/ Under "**Example Databases**"