

MySQL 4 Certification Study Guide Addendum

**Addendum to *MySQL Certification Study
Guide* (ISBN 0-672-32632-9)**

**MySQL 4 Certification Study Guide Addendum : Addendum
to *MySQL Certification Study Guide* (ISBN 0-672-32632-9)**

Table of Contents

Preface	v
Introduction	vi
1. Foreword	vi
2. Changes in the Exams	vi
2.1. Changes to the Core Exam	vi
2.2. Changes to the Professional Exam	vii
3. New Features in MySQL 4.1 Not on the Exam	vii
4. Acknowledgments	vii
1. Updates to the Core Exam	1
1.1. Specifying the Communication Protocol	1
1.2. CREATE TABLE	1
1.2.1. Changes to Storage Engine Specification Syntax	1
1.2.2. Changes to Storage Engine Selection Behavior	2
1.2.3. Using CREATE TABLE ... LIKE	2
1.3. The Storage Size of Character Columns	3
1.4. Using and Controlling New TIMESTAMP Behavior	4
1.4.1. Changes in the TIMESTAMP Display Format	4
1.4.2. Storing and Retrieving NULL Values in TIMESTAMP Columns	5
1.4.3. Controlling TIMESTAMP Behavior	5
1.4.4. Support for Per-Connection Time Zones	9
1.5. Subqueries	11
1.5.1. Four Types of Subqueries	12
1.5.2. Correlated Subqueries	12
1.5.3. Comparing Subquery Results to Outer Query Columns	13
1.5.4. Using IN and EXISTS	16
1.5.5. Comparison Using Row Subqueries	17
1.5.6. Using Subqueries in the FROM Clause	18
1.5.7. Subqueries as Scalar Expressions	19
1.5.8. Subqueries and Updating Statements	19
1.6. Prepared Statements	20
1.6.1. Preparing a Statement	20
1.6.2. User Variables for Prepared Statements	21
1.6.3. Executing the Prepared Statement	22
1.6.4. Deallocating Prepared Statements	23
1.7. Using Server-Side Help	23
1.8. Using INSERT ... ON DUPLICATE KEY UPDATE	24
1.9. Using the GROUP_CONCAT () Function	25
1.10. Using GROUP BY ... WITH ROLLUP	27
1.11. Exercises	29
2. Updates to the Professional Exam	47
2.1. Choosing Index Types for MEMORY tables	47
2.2. The Cluster Storage Engine	48
2.3. Using SHOW WARNINGS	48
2.4. Exercises	49
3. Errata for the MySQL Certification Study Guide	53

Preface

Copyright © 2005 MySQL AB Document revision date:

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher.

Trademarks: All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. MySQL AB cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

DISCLAIMER AND LIMITATION OF LIABILITY: EVERY EFFORT HAS BEEN MADE TO MAKE THIS BOOK AS COMPLETE AND ACCURATE AS POSSIBLE, BUT THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN "AS IS" BASIS. THERE ARE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, NON-INTERFERENCE, AND ACCURACY OF CONTENT WITH RESPECT TO ANY INFORMATION PROVIDED, NOR IS THERE ANY WARRANTY THAT THE INFORMATION WILL MEET ANY PARTICULAR NEEDS OR REQUIREMENTS. ALTHOUGH EVERY PRECAUTION HAS BEEN TAKEN IN THE PREPARATION OF THIS BOOK, THE PUBLISHER AND AUTHOR ASSUME NO RESPONSIBILITY FOR ERRORS OR OMISSIONS AND SHALL HAVE NO PATENT OR OTHER LIABILITY OR RESPONSIBILITY TO ANY PERSON OR ENTITY WITH RESPECT TO ANY LOSS OR DAMAGES ARISING FROM USE OF OR RELIANCE ON THE INFORMATION CONTAINED IN THIS BOOK.

The world sample database is Copyright Statistics Finland, <http://www.stat.fi/worldinfigures>. The world sample database may be downloaded from <http://dev.mysql.com/doc>.

Errata for this book is published at <http://www.mysql.com/certification/studyguides>.

This is a *supplement* to the "MySQL Certification Study Guide" (ISBN 0-672-32632-9). It is not a replacement for that book. For more information on how to obtain the MySQL Certification Study Guide and other titles from MySQL Press, please see <http://www.mysqlpress.com>.

About the Authors:

- *Paul DuBois* is a member of the MySQL AB documentation team and a leading author of MySQL books. His book *MySQL* is considered to be the definitive guide to using, administering, and programming MySQL.
- *Stefan Hinz* is the MySQL AB documentation team lead, a MySQL trainer and consultant, and the German translator of the *MySQL Reference Manual*.
- *Carsten Pedersen* is the MySQL AB certification manager, and has led the development of the MySQL certification program since its inception. He has also taught several MySQL courses in a number of countries.

Introduction

1. Foreword

In 2002, when MySQL AB first launched the MySQL Certification Program, we started authoring the *MySQL Certification Study Guide* (ISBN 0-672-32632-9). At that time, MySQL 4.0.x was the current production release of the MySQL server. Now, with the release of MySQL 4.1.7 in late 2004, MySQL AB has declared MySQL 4.1 as the new production version, and with that the MySQL Certification Program is being updated as well.

As of February 1st, 2005, MySQL 4 Certification exams will include questions on functionality in MySQL Server 4.1. For an itemized list of changes that will take place at that time, please read the latest version of the *MySQL Certification Candidate Guide*, available online at <http://www.mysql.com/training/certification>.

The changes involved in going from MySQL server version 4.0 to version 4.1 are certainly significant, yet, with respect to certification, fairly small in scope. That is the reasoning behind releasing this addendum rather than a completely new book (along with the fact that we do not wish to alienate those thousands of MySQL users who have already invested in the *MySQL Certification Study Guide*, but have not yet gone to an exam, by requiring them to invest in a completely new book).

The word "addendum" should be taken very literally — this booklet only covers the few subjects covered by the upgrade from MySQL 4.0 to 4.1; you will still need to read the other 600 pages of the *MySQL Certification Study Guide* to study properly for the exam.

The MySQL server, like any other software product, evolved during development from the alpha releases to the production releases. As you read this booklet and want to try out the examples and exercises, please make sure you are using the latest version of the MySQL server. Whenever you see "version 4.1" in this document, you should assume that it refers to MySQL Server 4.1.7 or later.

2. Changes in the Exams

The following presents a high-level overview of the changes that will occur on the exams as they are upgraded from MySQL 4.0 to MySQL 4.1.

2.1. Changes to the Core Exam

- *Structural changes:* The exam section *MySQL and MySQL AB* is reduced from taking up 10% of the exam questions to 5%. The section *SELECT statements* is increased from taking up 10% of the exam questions to 15%.
- In the exam section *The software making up MySQL*, questions on the *MySQL Control Center (mysqlcc)* are removed. This product is no longer under active development by MySQL AB, and has been replaced by the new GUI tools *MySQL Administrator* and *MySQL Query Browser*. Note that *MySQL Administrator* and *MySQL Query Browser* are not included in the MySQL Certification Program at this time.
- The exam section *Using Client Programs* will contain questions on using server-side help, as well as questions on specifying connection protocols for client/server communication.
- In the section *Data Definition Language*, questions on new `TIMESTAMP` behavior and storage requirements for character strings will appear.

- Questions on *subqueries*, the `GROUP_CONCAT()` function, and `GROUP BY ... WITH ROLLUP` will appear in the section *SELECT statements*.
- In the exam section *Basic SQL*, you should be prepared to answer questions on prepared statements.
- The exam section *Update Statements* will contain questions on `INSERT ... ON DUPLICATE KEY UPDATE`.

2.2. Changes to the Professional Exam

There will be no structural changes to the Professional exam. However, questions will appear on the topics described in Chapter 2, "Updates to the Professional Exam."

- Knowledge of the Cluster storage engine will be part of the exam section *MySQL Architecture*.
- The exam section *Optimizing for Query Speed* will contain questions on using different index types for MEMORY tables.
- `SHOW WARNINGS` will be part of the section *Advanced Server Features*.

3. New Features in MySQL 4.1 Not on the Exam

MySQL Server 4.1 introduces some completely new features that are important in their own right. But the area of application for these features is so new that they do not fit into the framework of general MySQL usage. We mention them here for the sake of completeness, and to clarify that they will not appear on the exam.

MySQL server now supports the *spatial data types* required for analyzing geographic features. The spatial data types allow you to store *locations* (such as lakes, cities, postcode areas, street intersections) in the database, and retrieve them using special functions. Most often, the spatial functions will, given a border, select those locations that are within the border, outside the border or which intersect the border.

Apart from briefly covering the need to be aware of potential new storage requirements, the exam does not touch upon newly introduced support for *character sets* and *character collations*.

4. Acknowledgments

We would like to thank Trudy Pelzer, who reviewed and commented on this addendum.

A big "thank you" also goes to all those keen-eyed readers that have written to us and made us aware of errors and omissions in the *MySQL Certification Study Guide*.

— Paul DuBois, Stefan Hinz, and Carsten Pedersen.

Chapter 1. Updates to the Core Exam

By far the largest number of changes in the transition from MySQL 4.0 to 4.1 has been in the available DDL (Data Definition Language) statements and the availability of subqueries when retrieving data from tables. There are numerous other changes, and to get the complete overview of these, it is necessary to look in the changelogs of the MySQL documentation for each 4.1.x version of the MySQL server.

In this chapter, we look only at the updates that are of interest to those seeking MySQL Core certification. Thus, this chapter is not a complete overview of all features changed or updated in the MySQL server and related products.

1.1. Specifying the Communication Protocol

In MySQL versions prior to 4.1, a number of details regarding the connection type created between a client and the server are implicit, based on how the host is specified. For example, specifying `localhost` as the server host when starting the `mysql` command-line client on Unix implies that a Unix socket file connection should be used. In such cases, any port specified through the `--port` option is ignored.

Starting with MySQL Server 4.1, you can explicitly select the type of protocol that should be used for a connection by specifying a `--protocol=type` option when invoking a client program. You can use `--protocol=tcp` on any platform to make a TCP/IP connection. Other protocol types are platform-specific. On Unix, `--protocol=socket` connects using a Unix socket file. On the Windows platform, where named-pipe or shared-memory connections are available, you can use `--protocol=pipe` or `--protocol=memory`. However, for the named-pipe connection protocol to be operative, you must use the `mysqld-nt` or `mysqld-max-nt` server, and the server must be started with the `--enable-named-pipe` option. Any server on Windows supports shared-memory connections, but the server must be started with the `--enable-shared-memory` option. If these options are not used when starting the server, it will not allow named-pipe or shared-memory connections. Specifying `--enable-shared-memory` at server start time has the additional effect that shared memory becomes the default connection protocol for local clients.

1.2. CREATE TABLE

There are several important changes in the `CREATE TABLE` statement:

- The keyword for specifying a table's storage engine has changed from `TYPE` to `ENGINE`. Parallel changes have been made to the names of the server startup option and the system variable that refers to the default storage engine: `--default-storage-engine` and `storage_engine` now are preferred over `--default-table-type` and `table_type`.
- The name of the storage engine for creating in-memory tables has changed from `HEAP` to `MEMORY`. `HEAP` is still recognized for backward compatibility.
- The server has a different behavior when you request the use of a storage engine that is not available. This applies to both the `CREATE TABLE` and `ALTER TABLE` statements.
- You can issue a `CREATE TABLE ... LIKE` statement to create an exact copy of an existing table's structure

1.2.1. Changes to Storage Engine Specification Syntax

In MySQL Server 4.0, you define the storage engine to use for a table by specifying the `TYPE = engine_name` table option in `CREATE TABLE` or `ALTER TABLE` statements.

In MySQL 4.0.18, the `ENGINE` keyword was added as a synonym for `TYPE`. As of MySQL Server 4.1, `ENGINE` is the preferred term. The server still recognizes `TYPE` for backward compatibility but generates a warning if you use it:

```
mysql> CREATE TABLE t (i INT) TYPE = InnoDB;
Query OK, 0 rows affected, 1 warning (0.04 sec)

mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 1287
Message: 'TYPE=storage_engine' is deprecated;
        use 'ENGINE=storage_engine' instead
1 row in set (0.00 sec)
```

Some related changes in MySQL 4.1 are that the `storage_engine` system variable is a synonym for `table_type`, and the `--default-storage-engine` option is a synonym for `--default-table-type`. The names that use "storage engine" are now preferred over those that use "table type."

Another change is that, as of MySQL 4.0.13, the name of the `HEAP` storage engine that manages in-memory tables has been changed to the `MEMORY` storage engine. The server still recognizes `HEAP` for backward compatibility.

1.2.2. Changes to Storage Engine Selection Behavior

In MySQL Server 4.0, you indicate which storage engine to use in `CREATE TABLE` or `ALTER TABLE` statements with a `TYPE = engine_name` table option. If the specified storage engine is not enabled in the server, it creates the table using the `MyISAM` storage engine instead. (A storage engine might be unavailable if it was not compiled in or was disabled at startup time.)

In MySQL Server 4.1, not only is the preferred storage engine-specification keyword `ENGINE` rather than `TYPE`, the selection behavior has also changed. If the requested storage engine is not available, the server uses the session setting of the `storage_engine` system variable to determine the default engine to use when creating a table. The default storage engine used thus is not necessarily `MyISAM`; it could be any available storage engine:

```
mysql> SET storage_engine = InnoDB;
mysql> CREATE TABLE t(i INT) ENGINE = BDB;
Query OK, 0 rows affected, 1 warning (0.01 sec)
mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 1266
Message: Using storage engine InnoDB for table 't'
```

If the server chooses to use the default storage engine rather than the one specified in the `CREATE TABLE` statement, it issues a warning.

The default setting of `storage_engine` is `MyISAM`. However, depending on how MySQL was installed or configured, `storage_engine` might be set to a different storage engine. Make sure to double-check the setting to ensure that it is really what you expect.

1.2.3. Using `CREATE TABLE ... LIKE`

In MySQL Server 4.1, you can use the `LIKE` keyword to create an empty table based on the definition of another table. The result is a new table with a definition that includes all column attributes and indexes of the original table. This differs from the result of using `CREATE TABLE ... SELECT` to create an empty table.

Example: Suppose that table `t` looks like this:

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

Either of the following statements will create an empty copy of the table:

```
mysql> CREATE TABLE copy1 SELECT * FROM t WHERE 0;
mysql> CREATE TABLE copy2 LIKE t;
```

However, the resulting copies differ in the amount of information retained from the original table structure:

```
mysql> SHOW CREATE TABLE copy1\G;
***** 1. row *****
      Table: copy1
Create Table: CREATE TABLE `copy1` (
  `i` int(11) NOT NULL default '0'
) ENGINE=MyISAM DEFAULT CHARSET=latin1
mysql> SHOW CREATE TABLE copy2\G;
***** 1. row *****
      Table: copy2
Create Table: CREATE TABLE `copy2` (
  `i` int(11) NOT NULL auto_increment,
  PRIMARY KEY (`i`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

The `CREATE TABLE ... SELECT` statement copied the column name and data type from the original table, but did not retain the `PRIMARY KEY` index information or the `AUTO_INCREMENT` column attribute information. The new table also uses the default storage engine, rather than the storage engine utilized by table `t`. The copy created with `CREATE TABLE ... LIKE` has none of these problems.

Some table attributes are not copied, even when issuing `CREATE TABLE ... LIKE`. The most notable examples are:

- If the original table is a `MyISAM` table for which the `DATA DIRECTORY` or `INDEX DIRECTORY` table options are specified, those options are not copied to the new table. The data and index files for the new table will reside in the database directory for the chosen database.
- Foreign key definitions in the original table are not copied to the new table. If you wish to retain the foreign key definitions, they must be re-specified with `ALTER TABLE` after creating the copy.

1.3. The Storage Size of Character Columns

When you create or alter a table, the sizes of character columns in column definitions now are interpreted as the number of *characters* that may be stored in the column, rather than the number of *bytes* that may be stored. This change applies to the `CHAR`, `VARCHAR`, and `TEXT` data types.

In MySQL 4.0, all lengths in character columns and index declarations are interpreted in byte units. The following table definition tells the server to set aside 32 bytes for storing data in the `c1` column, and to index the column using the first six bytes of each column value:

```
CREATE TABLE t
(
  c1 CHAR(32),
  INDEX (c1(6))
);
```

In MySQL 4.1, this has changed. Lengths in character columns are now interpreted as a number of *character units*, not byte units. In other words, in 4.1, `CHAR(32)` now means that the server sets aside enough space to store 32 *characters* in `c1`, and the index uses the first six characters of each column value.

With single-byte character sets, this change in meaning makes no difference, because each character uses exactly one byte for storage. However, with multi-byte character sets, this change can make a big difference. Suppose that a fixed-length `CHAR(32)` column is defined like this:

```
c1 CHAR(32) CHARACTER SET utf8
```

In this case, MySQL must allocate 32 character units per value, based on the size in bytes of the widest character in the character set. Characters in the `utf8` character set can take up to three bytes, so each value requires 96 bytes (32×3 bytes) of storage.

Similarly, an index on `c1` that is defined as `INDEX (c(6))` requires 6 characters (18 bytes), not 6 bytes.

Note that even though the sizes of character column and index values now depend upon the chosen character set, limits on the maximum size of index entries are not affected. The size of an index entry is still counted in bytes, not characters. For example, the limit for `MyISAM` tables is 1000 bytes per index entry. This means that the maximum number of characters in an index entry is less than 1000 if the index contains columns that use a multi-byte character set.

1.4. Using and Controlling New `TIMESTAMP` Behavior

The behavior of the `TIMESTAMP` data type has changed in a number of important ways in MySQL Server 4.1:

- The display format has changed.
- `TIMESTAMP` columns may be defined to accept and store `NULL` values.
- You have more control over the default value and update behavior of `TIMESTAMP` columns.
- It is possible to use connection-specific time zone settings.

1.4.1. Changes in the `TIMESTAMP` Display Format

The way that MySQL displays and returns `TIMESTAMP` values has changed. In MySQL 4.0, `8:15:16` on Jan 4, 2005 is displayed as `20050104081516`. In MySQL 4.1, the value is displayed using the same format as `DATETIME` values instead, that is, `'2005-01-04 8:15:16'`:

```
mysql> CREATE TABLE ts_test (ts TIMESTAMP);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO ts_test VALUES (NULL);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM ts_test;
+-----+
| ts                |
+-----+
| 2005-01-04 08:15:16 |
+-----+
1 row in set (0.00 sec)
```

A side effect of these changes is that the server ignores any display width specification that you use when creating or altering a `TIMESTAMP` column. For example, the server treats `TIMESTAMP(4)` as just `TIMESTAMP`.

1.4.2. Storing and Retrieving `NULL` Values in `TIMESTAMP` Columns

MySQL Server 4.0 disallows the storage of `NULL` values in `TIMESTAMP` columns. Assigning a `NULL` value to a `TIMESTAMP` column sets the value to the current timestamp. With MySQL Server 4.1, it is possible to store `NULL` values in `TIMESTAMP` columns.

However, to maintain backward compatibility, the server, by default, still defines `TIMESTAMP` columns as `NOT NULL` and stores the current timestamp in the column if you assign it a value of `NULL`. If you want to be able to store `NULL` in a `TIMESTAMP` column, you must explicitly write the column definition to allow `NULL` when creating or altering the column:

```
mysql> CREATE TABLE ts_null (ts TIMESTAMP NULL);
Query OK, 0 rows affected (0.04 sec)

mysql> DESCRIBE ts_null;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| ts    | timestamp | YES  |    | NULL    |      |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.10 sec)
```

Previous versions of MySQL Server would also allow you to include `NULL` in the column definition, but would ignore it: Inserting `NULL` into the column sets it to the current timestamp.

Note that specifying `NULL` for a `TIMESTAMP` column implicitly changes its default value from `CURRENT_TIMESTAMP` to `NULL` if no explicit default value is given. (`CURRENT_TIMESTAMP` is discussed in section 1.4.3, "Controlling `TIMESTAMP` Behavior.")

1.4.3. Controlling `TIMESTAMP` Behavior

The initialization and update behavior of `TIMESTAMP` columns can now be controlled in a more fine-grained manner. In MySQL Server 4.0, the first (and only the first) `TIMESTAMP` column in a table is automatically initialized and updated whenever the record changes. In MySQL Server 4.1, you can specify that any single `TIMESTAMP` column in a table should be initialized with the current timestamp when the record is created with `INSERT` or `REPLACE`, updated with the current timestamp when the record is changed with `UPDATE`, or both.

To control the initialization and update behavior of a `TIMESTAMP` column, you add either or both of the `DEFAULT CURRENT_TIMESTAMP` and `ON UPDATE CURRENT_TIMESTAMP` attributes to the column definition when creating the table with `CREATE TABLE` or changing it with `ALTER TABLE`.

The `DEFAULT CURRENT_TIMESTAMP` attribute causes the column to be initialized with the current timestamp at the time the record is created. The `ON UPDATE CURRENT_TIMESTAMP` attribute causes the column to be updated with the current timestamp when the value of another column in the record is changed from its current value.

For backward compatibility with older versions of MySQL, if you do not specify either of the `DEFAULT CURRENT_TIMESTAMP` or `ON UPDATE CURRENT_TIMESTAMP` attributes when creating a table, the MySQL server automatically assigns both attributes to the first `TIMESTAMP` column:

```
mysql> CREATE TABLE ts_test1 (
->   ts1 TIMESTAMP,
->   ts2 TIMESTAMP,
->   data CHAR(30)
-> );
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> DESCRIBE ts_test1;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default                | Extra |
+-----+-----+-----+-----+-----+-----+
| ts1   | timestamp     | YES  |     | CURRENT_TIMESTAMP      |       |
| ts2   | timestamp     | YES  |     | 0000-00-00 00:00:00    |       |
| data  | char(30)      | YES  |     | NULL                   |       |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

```
mysql> INSERT INTO ts_test1 (data) VALUES ('original_value');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM ts_test1;
+-----+-----+-----+
| ts1           | ts2           | data           |
+-----+-----+-----+
| 2005-01-04 14:45:51 | 0000-00-00 00:00:00 | original_value |
+-----+-----+-----+
1 row in set (0.00 sec)
```

mysql> ... *time passes* ...

```
mysql> UPDATE ts_test1 SET data='updated_value';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> SELECT * FROM ts_test1;
+-----+-----+-----+
| ts1           | ts2           | data           |
+-----+-----+-----+
| 2005-01-04 14:46:17 | 0000-00-00 00:00:00 | updated_value  |
+-----+-----+-----+
1 row in set (0.00 sec)
```

The same behavior occurs if you specify both `DEFAULT CURRENT_TIMESTAMP` and `ON UPDATE CURRENT_TIMESTAMP` explicitly for the first `TIMESTAMP` column. It is also possible to use just of the attributes. The following example uses `DEFAULT CURRENT_TIMESTAMP`, but omits `ON UPDATE CURRENT_TIMESTAMP`. The result is that the column is initialized automatically, but not updated when the record is updated:

```
mysql> CREATE TABLE ts_test2 (
  ->   created_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  ->   data CHAR(30)
  -> );
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO ts_test2 (data) VALUES ('original_value');
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM ts_test2;
+-----+-----+
| created_time          | data          |
+-----+-----+
| 2005-01-04 14:46:39  | original_value |
+-----+-----+
1 row in set (0.00 sec)

mysql> ... time passes ...

mysql> UPDATE ts_test2 SET data='updated_value';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM ts_test2;
+-----+-----+
| created_time          | data          |
+-----+-----+
| 2005-01-04 14:46:39  | updated_value |
+-----+-----+
1 row in set (0.00 sec)
```

Note that even though the record is updated, the `created_time` column is not. In previous versions of MySQL Server, the `UPDATE` statement would have caused the `created_time` column to be updated as well.

The next example demonstrates how to create a `TIMESTAMP` column that is not set to the current timestamp when the record is created, but only when it is updated. In this case, the column definition includes `ON UPDATE CURRENT_TIMESTAMP` but omits `DEFAULT CURRENT_TIMESTAMP`:

```
mysql> CREATE TABLE ts_test3 (
  ->   updated_time TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  ->   data CHAR(30)
  -> );
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO ts_test3 (data) VALUES ('original_value');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM ts_test3;
+-----+-----+
| updated_time          | data          |
+-----+-----+
| 0000-00-00 00:00:00  | original_value |
+-----+-----+
1 row in set (0.00 sec)

mysql> UPDATE ts_test3 SET data='updated_value';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM ts_test3;
+-----+-----+
| updated_time          | data          |
+-----+-----+
| 0000-00-00 00:00:00  | updated_value |
+-----+-----+
1 row in set (0.00 sec)
```

```

| updated_time          | data          |
+-----+-----+
| 2005-01-04 14:47:10 | updated_value |
+-----+-----+
1 row in set (0.00 sec)

```

Note that you can choose to use `CURRENT_TIMESTAMP` with neither, either, or both of the attributes for a single `TIMESTAMP` column, but you cannot use `DEFAULT CURRENT_TIMESTAMP` with one column and `ON UPDATE CURRENT_TIMESTAMP` with another:

```

mysql> CREATE TABLE ts_test4 (
->   created TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
->   updated TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
->   data CHAR(30)
-> );
ERROR 1293 (HY000): Incorrect table definition; there can be
only one TIMESTAMP column with CURRENT_TIMESTAMP in DEFAULT
or ON UPDATE clause

```

Nevertheless, you can achieve the effect of having one column with the creation time and another with the time of the last update. To do this, create two `TIMESTAMP` columns. Define the column that should hold the creation time with `DEFAULT 0` and explicitly set it to `NULL` whenever you `INSERT` a new record. Define the column that should hold the updated time with `DEFAULT CURRENT_TIMESTAMP`:

```

mysql> CREATE TABLE ts_test5 (
->   created TIMESTAMP DEFAULT 0,
->   updated TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
->   data CHAR(30)
-> );
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO ts_test5 (created, data)
-> VALUES (NULL, 'original_value');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM ts_test5;
+-----+-----+-----+
| created          | updated          | data          |
+-----+-----+-----+
| 2005-01-04 14:47:39 | 0000-00-00 00:00:00 | original_value |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> ... time passes ...

mysql> UPDATE ts_test5 SET data='updated_value';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM ts_test5;
+-----+-----+-----+
| created          | updated          | data          |
+-----+-----+-----+
| 2005-01-04 14:47:39 | 2005-01-04 14:47:52 | updated_value |
+-----+-----+-----+
1 row in set (0.00 sec)

```

As mentioned in section 1.4.2, "Storing and Retrieving `NULL` Values in `TIMESTAMP` Columns," specifying `NULL` for a `TIMESTAMP` column implicitly changes its default value from `CURRENT_TIMESTAMP` to `NULL` if no explicit default value is given.

1.4.4. Support for Per-Connection Time Zones

In MySQL Server 4.1, it is now possible to set the current time zone on a per-connection basis.

To discuss time zones, we must first introduce a number of concepts:

- *UTC* is "Coordinated Universal Time" and is the common reference point for time measurement. For purposes of this discussion, UTC is the same as Greenwich Mean Time (GMT), although time zone aficionados get into long discussions about astronomical observations, atomic clocks, "Universal Time" vs. "Greenwich Mean Time" vs. "Coordinated Universal Time," and much else.
- There are three *time zone formats* available to use with MySQL:
 - The *signed hour/minute offset* of a time zone is expressed as '+hh:mm' or '-hh:mm', where *hh* and *mm* stand for two-digit hours and minutes, respectively. UTC is, in this format, commonly expressed as '+00:00'. Each time zone bases its offset according to the distance between it and the UTC time zone. Berlin, Germany, is one hour ahead of Greenwich, England (for example, the sun rises in Berlin approximately one hour before it does in Greenwich), so the hour/minute offset for Berlin is expressed as '+01:00'. In New York, where the sun rises some 5 hours after it does in Greenwich, the hour/minute offset is expressed as '-05:00'.
 - The *named time zone* for a given location is defined by a string such as 'US/Eastern', which is translated into the correct time zone by the server. MySQL supports named time zones through a set of time zone tables in the `mysql` database.
 - The third format is the `SYSTEM` time zone. This stands for the time zone value that the MySQL server retrieves from the server host. The server uses this value as its default time zone setting when it begins executing.

The exact details of support for named time zones differ slightly from one operating system to the next, and are not covered in any detail on the certification exam. However, knowing how to use time zone support using signed offsets *is* mandatory.

Time zone settings are determined by the `time_zone` system variable. The server maintains a global `time_zone` value, as well as a session `time_zone` value for each client that connects. The session value is initialized for a given client, from the current value of the global `time_zone` variable, when the client connects.

The default setting for the global value is `SYSTEM`, which thus also becomes each client's initial session `time_zone` value. The global and session time zone settings can be retrieved with the following statement:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+-----+-----+
| @@global.time_zone | @@session.time_zone |
+-----+-----+
| SYSTEM             | SYSTEM              |
+-----+-----+
1 row in set (0.00 sec)
```

MySQL Server stores `TIMESTAMP` values internally in UTC. It converts `TIMESTAMP` values from the server's current time zone for storage, and converts back to the current time zone for retrieval. The standard setting for both server and the per-client connection is to use the `SYSTEM` setting, which the server retrieves from the host at startup.

If the time zone setting is the same for both storage and retrieval, you will get back the same value you store. If you store a `TIMESTAMP` value, and then change the time zone to a different value, the returned

TIMESTAMP value will be different from the one you stored.

The following examples demonstrate how to change the session time zone settings to store and retrieve TIMESTAMP data. First, we set both the global and the session time zones to UTC, that is, '+00:00':

```
mysql> SET time_zone = '+00:00';
Query OK, 0 rows affected (0.00 sec)
```

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

```
+-----+
| @@session.time_zone |
+-----+
| +00:00              |
+-----+
1 row in set (0.00 sec)
```

Next, we create a simple table containing just a TIMESTAMP column named `ts` and insert one record that assigns the current time to `ts`. Then we retrieve the record:

```
mysql> CREATE TABLE ts_test (ts TIMESTAMP);
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> INSERT INTO ts_test (ts) VALUES (NULL);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM ts_test;
```

```
+-----+
| ts                |
+-----+
| 2005-01-04 20:50:18 |
+-----+
1 row in set (0.00 sec)
```

Finally, we change the session time zone twice, each time retrieving the value after the change. This demonstrates that, even though we're retrieving the same TIMESTAMP value, the change in time zone setting causes the "localized" display value to be different each time:

```
mysql> SET time_zone = '+02:00';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT * FROM ts_test;
```

```
+-----+
| ts                |
+-----+
| 2005-01-04 22:50:18 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SET time_zone = '-05:00';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT * FROM ts_test;
```

```
+-----+
| ts                |
+-----+
| 2005-01-04 15:50:18 |
+-----+
1 row in set (0.00 sec)
```

The per-connection time zone settings also influence other aspects of the MySQL server that depend on the current time, most notably the function `NOW()`.

MySQL Server 4.1 also introduces the function `CONVERT_TZ()`, which performs time zone conversions of `datetime` values:

```
mysql> SELECT CONVERT_TZ('2005-01-27 13:30:00', '+01:00', '+03:00');
+-----+
| CONVERT_TZ('2005-01-27 13:30:00', '+01:00', '+03:00') |
+-----+
| 2005-01-27 15:30:00 |
+-----+
1 row in set (0.00 sec)
```

`CONVERT_TZ()` assumes that the given `datetime` value has the time zone represented by the first hour/minute offset argument, and converts it to a value in the time zone represented by the second offset argument. The result is that you get the same `datetime` value, from the point of view of a different time zone.

1.5. Subqueries

One of the most important new features of MySQL Server 4.1 is the ability to use *subqueries*. A subquery is a `SELECT` statement that is placed inside another SQL statement.

As a matter of fact, versions of MySQL Server prior to 4.1 also had subquery support, albeit to a very limited extent. You are already familiar with the two constructs `INSERT INTO ... SELECT` and `CREATE TABLE ... SELECT`. In both of these, the `SELECT` is a fully qualified subquery. However, there's a lot more to subqueries than these simple usages. Many of the examples that follow use the tables in the `world` database, which is also used in the *MySQL Certification Study Guide*. (The `world` database was compiled by and is copyright Statistics Finland, <http://www.stat.fi/worldinfigures>. You can download a copy of the `world` sample database from <http://dev.mysql.com/doc>.)

The following example shows how a simple subquery works. We use the two tables `Country` and `CountryLanguage` from the `world` database to find the languages spoken in Finland:

```
mysql> SELECT Language
-> FROM CountryLanguage
-> WHERE CountryCode = (SELECT Code
->                        FROM Country
->                        WHERE Name='Finland');
+-----+
| Language |
+-----+
| Estonian |
| Finnish  |
| Russian  |
| Saame    |
| Swedish  |
+-----+
5 rows in set (0.00 sec)
```

The following query could not have been written as a single, efficient statement in previous versions of MySQL Server. It determines which country has the most populous city in the world:

```
mysql> SELECT Country.Name
-> FROM Country, City
-> WHERE Country.Code = City.CountryCode
-> AND City.Population = (SELECT MAX(Population)
->                        FROM City);
+-----+
| Name |
```

```
+-----+
| India |
+-----+
1 row in set (0.01 sec)
```

However, as you will undoubtedly notice in many of the descriptions and examples in this section, most uses of subqueries can be rewritten to completely equivalent (and often more efficient) queries using joins. Nonetheless, subqueries are preferred by many as an alternative way of specifying relations that otherwise require complex joins or unions. Some users insist on using subqueries simply because they find them much more readable and easier to maintain than queries involving complex joins.

1.5.1. Four Types of Subqueries

In this discussion, we divide subqueries into four general categories, which affect the contexts in which they can be used:

- *Scalar subqueries* return a single value, that is, one row with one column of data. The previous query is an example of a scalar subquery.
- *Row subqueries* return a single row with one or more columns of data.
- *Column subqueries* return one or more rows of data, in a single column.
- *Table subqueries* return a result with one or more rows containing one or more columns of data.

1.5.2. Correlated Subqueries

Another distinction between subqueries is whether or not they are *correlated*. A correlated subquery is a subquery that contains references to the values in the *outer query*. In the following example, we calculate which country on each continent has the largest population. The value of the column `Continent`, which appears in the outer query, is used to limit the number of rows considered for the `MAX()` calculation in the subquery.

```
mysql> SELECT Continent, Name, Population
-> FROM Country c
-> WHERE Population = (SELECT MAX(Population)
->                      FROM Country c2
->                      WHERE c.Continent=c2.Continent
->                      );
```

Continent	Name	Population
Oceania	Australia	18886000
South America	Brazil	170115000
Asia	China	1277558000
Africa	Nigeria	111506000
Europe	Russian Federation	146934000
North America	United States	278357000
Antarctica	Antarctica	0
Antarctica	Bouvet Island	0
Antarctica	South Georgia and the South Sandwich Islands	0
Antarctica	Heard Island and McDonald Islands	0
Antarctica	French Southern territories	0

```
11 rows in set (0.07 sec)
```

The 5 lines of output for Antarctica are quite correct. Although the continent of Antarctica has no coun-

tries in reality, it does consist of a number of administered territories. The world database recognizes 5 of these as countries. So there are indeed 5 "countries" in Antarctica, all of which have a population of 0. Thus, they are all contenders for the maximum population in Antarctica, which is also 0.

Note how the table qualifiers `c` and `c2` are used in the example. This is necessary because the columns that are used to correlate values from the inner and outer queries come from different copies of the same table and thus have the same name.

1.5.3. Comparing Subquery Results to Outer Query Columns

The previous sections showed two examples of scalar subqueries that use the `=` equality operator to compare a single column to the value returned by the subquery. But you are not limited to using the `=` equality operator. When comparing the values in the outer query with those returned by a scalar subquery, you can make use of all the usual comparison operators such as `=`, `<`, `>`, `<>`, and `>=`.

You can use a scalar subquery wherever you would otherwise use a literal value (such as `3` or `'abc'`), a function value (such as `RAND()`), or any column value. A more thorough discussion on using scalar subqueries appears in 1.5.7, "Subqueries as Scalar Expressions."

When using scalar subqueries in comparisons, you must ensure that the subquery returns only a single value. Suppose that we wanted to find out whether there is a country that has a city with a population of less than 100, using the following subquery:

```
mysql> SELECT Code c, Name
-> FROM Country
-> WHERE 100 > (SELECT Population
-> FROM City
-> WHERE CountryCode = c);
ERROR 1242 (21000): Subquery returns more than 1 row
```

The subquery returns more than one value, so the statement fails.

To perform a comparison between a scalar value and a subquery that returns several rows of data in a single column (a column subquery), we must use a *quantified comparison*. The quantifier keywords `ALL`, `ANY`, and `SOME` let us consider multiple-row results in our comparisons.

Using the `ALL` keyword when doing a comparison with a column subquery limits the result set to only those records where the comparison is true for *all* values produced by the subquery. Consider the following query, which tells us the average country population for each of the world's continents:

```
mysql> SELECT Continent, AVG(Population)
-> FROM Country
-> GROUP BY Continent;
```

Continent	AVG(Population)
Asia	72647562.7451
Europe	15871186.9565
North America	13053864.8649
Africa	13525431.0345
Oceania	1085755.3571
Antarctica	0.0000
South America	24698571.4286

```
7 rows in set (0.00 sec)
```

Now, suppose that we would like to know all the countries in the world where the population is larger than the average country population of all of the world's continents. To get this information, we can use ALL in conjunction with the > operator to compare the value of the country population with every average continent population from the preceding result:

```
mysql> SELECT Name, Population
-> FROM Country
-> WHERE Population > ALL (SELECT AVG(Population)
-> FROM Country
-> GROUP BY Continent)
-> ORDER BY Name;
```

Name	Population
Bangladesh	129155000
Brazil	170115000
China	1277558000
Germany	82164700
India	1013662000
Indonesia	212107000
Japan	126714000
Mexico	98881000
Nigeria	111506000
Pakistan	156483000
Philippines	75967000
Russian Federation	146934000
United States	278357000
Vietnam	79832000

14 rows in set (0.00 sec)

Note that Continent has been removed from the subquery's SELECT clause, because a quantified subquery can produce only a single column of values. If the subquery is written to select both the Continent column and the calculated column, MySQL cannot tell which one to use in the comparison and issues a complaint:

```
mysql> SELECT Name
-> FROM Country
-> WHERE Population > ALL (SELECT Continent, AVG(Population)
-> FROM Country
-> GROUP BY Continent)
-> ORDER BY Name;
```

ERROR 1241 (21000): Operand should contain 1 column(s)

The keyword ANY (as well as the other quantified comparison keywords) is not limited to working with the = operator. Any of the standard comparison operators (=, <, >, <>, >=, and so forth) may be used for the comparison.

Comparisons using the word ANY will, as the name implies, succeed for any values in the column of data found by the subquery which succeed in the comparison. The following example finds the countries on the European continent, and, for each one, tests whether the country is among the world-wide list of countries where Spanish is spoken:

```
mysql> SELECT Name
-> FROM Country
-> WHERE Continent = 'Europe'
-> AND Code = ANY (SELECT CountryCode
-> FROM CountryLanguage
-> WHERE Language = "Spanish")
-> ORDER BY Name;
```

```

+-----+
| Name   |
+-----+
| Andorra|
| France |
| Spain  |
| Sweden |
+-----+
4 rows in set (0.00 sec)

```

Compare that query to the following one using ALL: We run the same query, changing ANY to ALL to see if the European continent covers all those countries where Spanish is spoken:

```

mysql> SELECT Name
-> FROM Country
-> WHERE Continent = 'Europe'
-> AND Code = ALL (SELECT CountryCode
-> FROM CountryLanguage
-> WHERE Language = 'Spanish')
-> ORDER BY Name;
Empty set (0.00 sec)

```

Because the result is empty, we can conclude that the European continent is not the only one where Spanish is spoken.

The word SOME is an alias for ANY, and may be used anywhere that ANY is used. The SQL standard defines these two words with the same meaning to overcome a limitation in the English language. Consider the following statement, in which we use <> ANY to negate the sense of the previous ANY example. As you read the example, try to form a sentence in your head to describe the output you would expect from the query (the output has been reduced to enhance readability):

```

mysql> SELECT Name
-> FROM Country
-> WHERE Continent = 'Europe'
-> AND Code <> ANY (SELECT CountryCode
-> FROM CountryLanguage
-> WHERE Language = 'Spanish')
-> ORDER BY Name;

```

```

+-----+
| Name   |
+-----+
| Albania|
| Andorra|
| Austria|
| ..... |
| Finland|
| France |
| Germany|
| ..... |
| Svalbard and Jan Mayen|
| Sweden |
| Switzerland|
| Ukraine |
| United Kingdom|
| Yugoslavia|
+-----+
46 rows in set (0.01 sec)

```

You probably expected this query to find "all the countries on the European continent where Spanish is not spoken", or something similar. Yet the query actually finds every single country on the European continent.

In the English language, we expect "not any" to mean "none at all". However, in SQL, `<> ANY` means "one or more do not match". In other words, the statement is really saying "return all the countries, where there are *some* people that do not speak Spanish". In our example, for all of the four countries where there are Spanish speakers, we do in fact also find speakers of other languages.

To alleviate the confusion that might arise from the use of `<> ANY`, the SQL standard includes the `SOME` keyword, as a synonym for `ANY`. Using the `<> SOME` construct makes it easier to understand the expected outcome of the SQL statement:

```
SELECT Name
FROM Country
WHERE Continent = 'Europe'
      AND Code <> SOME (SELECT CountryCode
                       FROM CountryLanguage
                       WHERE Language = 'Spanish')

ORDER BY Name;
```

1.5.4. Using IN and EXISTS

In addition to the quantified comparison predicates formed with `ALL`, `ANY`, and `SOME`, there are two further predicates supported by MySQL Server: `IN` and `EXISTS`.

You are already familiar with the variant of `[NOT] IN` that may be used in an expression, as shown in the following example:

```
mysql> SELECT Name
      -> FROM Country
      -> WHERE Code IN ('DEU', 'USA', 'JPN');
```

Name
Germany
Japan
United States

```
3 rows in set (0.00 sec)
```

However, in this case using `IN` is merely a shorthand for writing `WHERE Code='DEU' OR Code='USA' OR Code='JPN'`. It has nothing to do with subqueries.

When `IN` is used with a subquery, it is functionally equivalent to `= ANY` (note that the `=` sign is part of the equivalence). Many consider `IN` to be more readable than `= ANY`, because what you really want to know is "does this value appear in the subquery?" As an example, consider the equivalent `IN` version of the `= ANY` example shown in the previous section:

```
SELECT Name
FROM Country
WHERE Continent = 'Europe'
      AND Code IN (SELECT CountryCode
                  FROM CountryLanguage
                  WHERE Language = 'Spanish')

ORDER BY Name;
```

`IN` cannot be combined with any comparison operators such as `=` or `<>`.

`NOT IN` is another "shorthand." However, it is *not* an alias of `<> ANY` as you might otherwise expect. It is an alias of `<> ALL`. In other words, `NOT IN` is only true if none of the records of the subquery can be matched by the outer query. In the example for `SOME`, we demonstrated that `<> ANY` would return re-

cords of countries where some people didn't speak Spanish. The same query, using `NOT IN` (that is, `<> ALL`) will return only those countries where Spanish is not spoken at all. Although it may seem logically flawed that `IN` and `NOT IN` are aliases of two very different statements, it fits better with the way that we usually understand the equivalent English terms.

The `EXISTS` predicate performs a simple test: It tells you whether the subquery finds any rows. It does not return the actual values found in any of the rows, it merely returns `TRUE` if any rows were found and `FALSE` if no rows were found. As does one of our previous examples, the following example finds countries on the European continent where Spanish is spoken. But with this query, no actual comparison is made between the data in the outer query and the rows found in the inner query.

```
mysql> SELECT Code c, Name
-> FROM Country
-> WHERE Continent = 'Europe'
-> AND EXISTS (SELECT *
->              FROM CountryLanguage
->              WHERE CountryCode = c
->              AND Language = 'Spanish');
```

c	Name
AND	Andorra
ESP	Spain
FRA	France
SWE	Sweden

```
4 rows in set (0.00 sec)
```

The use of `SELECT *` in the subquery is purely by tradition. You can use a different column list as long as the subquery is syntactically correct. No column values are ever needed for comparison, so MySQL never actually evaluates the column list given in the subquery `SELECT`. For example, you could replace the `*` with a constant value such as `1`, `0`, or even `NULL`.

`EXISTS` can be negated using `NOT EXISTS`, which, as the name implies, returns `TRUE` for subquery result sets with no rows. Replacing `EXISTS` with `NOT EXISTS` in the previous example shows those 42 countries on the European continent in which Spanish is not spoken at all.

1.5.5. Comparison Using Row Subqueries

For row subqueries, we can perform an equality comparison for all columns in a row. The subquery must return a single row. This method of comparison is not often used, but can provide some convenience for certain comparison operations. In the following example, we find the name of the capital of Finland. The query makes use of the fact that the city's name is stored in the `City` table, whereas the ID of a country's capital city is stored in the `Country` table:

```
mysql> SELECT City.Name
-> FROM City
-> WHERE (City.ID, City.CountryCode) =
->        (SELECT Capital, Code
->         FROM Country
->         WHERE Name='Finland');
```

Name
Helsinki [Helsingfors]

```
1 row in set (0.02 sec)
```

Notice the use of the construct `(City.ID, City.CountryCode)`. This is known as a *row con-*

structor. An equivalent method of defining a row is using `ROW()`, to underscore the fact that the values are used to construct a row of data for comparison. In this case, we would have written `ROW(City.ID, City.CountryCode)`.

Trying to compare a tuple created by the row constructor with a subquery that returns several rows at once produces an error. The following example is similar to the preceding one, but does not work because there is no limit on the number of rows returned by the subquery:

```
mysql> SELECT City.Name
-> FROM City
-> WHERE (City.ID, City.CountryCode) =
->       (SELECT Capital, Code
->        FROM Country);
ERROR 1242 (21000): Subquery returns more than 1 row
```

Row constructors can be used only for equality comparison using the `=` operator. You may not use other comparison operators such as `<`, `>`, or `<>`; nor may you use special words such as `ALL`, `ANY`, or `IN`.

Row constructors are commonly used with row subqueries, but they can be used in other contexts, and they may contain any type of scalar expression. For example, the following is a legal statement:

```
mysql> SELECT Name, Population
-> FROM Country
-> WHERE (Continent, Region) = ('Europe', 'Western Europe');
```

Name	Population
Netherlands	15864000
Belgium	10239000
Austria	8091800
Liechtenstein	32300
Luxembourg	435700
Monaco	34000
France	59225700
Germany	82164700
Switzerland	7160400

```
9 rows in set (0.01 sec)
```

In practice, row constructors are often inefficient when used like this, so it is more common to write the equivalent expression using `AND`. The query optimizer performs better if you write the `WHERE` clause like this:

```
SELECT Name, Population
FROM Country
WHERE Continent = 'Europe' AND Region = 'Western Europe';
```

1.5.6. Using Subqueries in the FROM Clause

Subqueries may be used in the `FROM` clause of a `SELECT` statement. In the following query, we find the average of the sums of the population of each continent:

```
mysql> SELECT AVG(cont_sum)
-> FROM (SELECT Continent, SUM(Population) AS cont_sum
->        FROM Country
->        GROUP BY Continent
->        ) AS _t;
```

AVG(cont_sum)

```
+-----+
| 868392778.5714 |
+-----+
1 row in set (0.13 sec)
```

Because every table that appears in a FROM clause must have a name, a subquery in the FROM clause must be followed by a table alias.

The SELECT in the FROM clause can be a table subquery, even if not all of its values are used by the outer query. This is shown by the preceding example, where the Continent column selected by the subquery is not used by the outer query.

Subqueries in FROM clauses may not be correlated with the outer statement.

1.5.7. Subqueries as Scalar Expressions

Scalar subqueries can appear anywhere that a scalar value is allowed by the SQL syntax. This means that you can use subqueries as function parameters, use mathematical operators on subqueries that contain numeric values, and so forth. The following example shows how to use a scalar subquery as a parameter to the CONCAT() function:

```
mysql> SELECT CONCAT('The country code for Finland is: ',
-> (SELECT Code
-> FROM Country
-> WHERE Name='Finland')) AS s1;
+-----+
| s1 |
+-----+
| The country code for Finland is: FIN |
+-----+
1 row in set (0.02 sec)
```

Notice that the subquery must be enclosed in parentheses here, just as in any other context where a subquery may appear.

The next example shows the use of scalar subqueries in a mathematical expression that calculates the ratio of the people living in cities to that of the world population:

```
mysql> SELECT (SELECT SUM(Population) FROM City) /
-> (SELECT SUM(Population) FROM Country) AS ratio;
+-----+
| ratio |
+-----+
| 0.24 |
+-----+
1 row in set (0.02 sec)
```

1.5.8. Subqueries and Updating Statements

Use of subqueries is not limited to SELECT statements. Any SQL statement that includes a WHERE clause or scalar expression may use subqueries. For example, to create a new table containing every North American city, and then later remove all cities located in countries where the life expectancy is less than 70 years, use these statements:

```
mysql> CREATE TABLE nacities
-> SELECT * FROM City
-> WHERE CountryCode IN (SELECT Code
-> FROM Country
```

```
->                               WHERE Continent='North America');
Query OK, 581 rows affected (0.11 sec)
Records: 581  Duplicates: 0  Warnings: 0

mysql> DELETE FROM nacies
-> WHERE CountryCode IN (SELECT Code
->                         FROM Country
->                         WHERE LifeExpectancy < 70.0);
Query OK, 26 rows affected (0.02 sec)
```

Although subqueries can be used to retrieve or aggregate data from other tables for updating statements (such as UPDATE, DELETE, INSERT, and REPLACE) MySQL does not allow the table being updated to appear in any subquery of the statement. For example, the following statement yields an error:

```
mysql> DELETE FROM nacies
-> WHERE ID IN (SELECT ID
->              FROM nacies
->              WHERE Population < 500);
ERROR 1093 (HY000): You can't specify target table 'nacies'
for update in FROM clause
```

1.6. Prepared Statements

A new feature of MySQL Server 4.1 is support for *prepared statements*. Prepared statements are useful when you want to run several queries that differ only in very small details.

Besides being a convenience, prepared statements also offer enhanced performance, because the complete statement is parsed only once by the server. Once the parse is complete, the server and client may make use of a new protocol that makes fewer data conversions (and usually makes for less traffic between the server and client) than in earlier versions of MySQL.

MySQL Server 4.1 does not allow every type of SQL statement to be prepared. The types of SQL statement that may be prepared are limited to the following:

- SELECT statements
- Updating statements: INSERT, REPLACE, UPDATE, and DELETE
- CREATE TABLE statements

In most circumstances, statements are prepared and executed using the application programming interface that you normally use with MySQL. However, to aid in testing and debugging, it is possible to define and use prepared statements from within the `mysql` command-line client. For purposes of certification, prepared statement use from within the `mysql` command-line client is the context used to explain prepared statements here. It is also the context in which questions on prepared statements will appear on the exam.

1.6.1. Preparing a Statement

The PREPARE statement is used to define an SQL statement that will be executed later. PREPARE takes two arguments: the text of an SQL statement, and a name to assign to the statement once it has been prepared. The statement may not be complete, because data values that are unknown at preparation time are represented by question mark ('?') characters that serve as parameter markers. At the time the statement is executed, you provide specific data values. The server replaces the markers with the values to complete the statement. Different values can be used each time the statement is executed.

In the following example, we prepare a statement named `namepop`. When executed later with a country code as a parameter value, the statement will return a result set containing the corresponding country name and population from our database.

```
mysql> PREPARE namepop FROM '  
'> SELECT Name, Population  
'> FROM Country  
'> WHERE Code = ?  
'> ;  
Query OK, 0 rows affected (0.02 sec)  
Statement prepared
```

The message `Statement prepared` tells us that the server is ready to execute the `namepop` statement. On the other hand, if the server finds a problem as it parses the statement during a `PREPARE`, it returns an error and does not prepare the statement:

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

If you `PREPARE` a statement using a statement name that already exists, the server first discards the prepared statement currently associated with the name, and then prepares the new statement. If the new statement contains an error and cannot be prepared, the result is that no statement with the given name will exist.

A prepared statement exists only for the duration of the session in which it is created, and is visible only to the session in which it is created. When a session ends, all prepared statements for that session are discarded.

1.6.2. User Variables for Prepared Statements

After a statement has been prepared, it can be executed. If the statement contains any '?' parameter markers, a data value must be supplied for each of them by means of *user variables*. (The certification exam does not cover user variables in depth. Here, as well as on the exam, we cover them only to the extent required to demonstrate their use with prepared statements.)

User variables are written as `@var_name` and may be set to an integer, real, string, or NULL value. To assign a value to a variable in a `SET` statement, you can use either `=` or `:=` as the assignment operator:

```
mysql> SET @var1 = 'USA';  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> SET @var2 := 'GBR';  
Query OK, 0 rows affected (0.00 sec)
```

In other contexts, such as inside a `SELECT` statement, use the `:=` assignment operator (not `=`), and omit the `SET` keyword:

```
mysql> SELECT @var3 := 'CAN';  
+-----+  
| @var3 := 'CAN' |  
+-----+  
| CAN            |  
+-----+
```

1 row in set (0.00 sec)

If you refer to an uninitialized variable that has not been assigned a value explicitly, its value is NULL:

```
mysql> SELECT @var1, @var2, @var3, @var4;
+-----+-----+-----+-----+
| @var1 | @var2 | @var3 | @var4 |
+-----+-----+-----+-----+
| USA   | GBR   | CAN   | NULL  |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

User variables exist only for the duration of the connection in which they are used. When the connection ends, all user variables are lost.

1.6.3. Executing the Prepared Statement

To execute a prepared statement, initialize any user variables needed to provide parameter values, and then issue an EXECUTE ... USING statement. Here's a complete example that prepares a statement and then executes it several times using different data values:

```
mysql> PREPARE namepop FROM '
      '> SELECT Name, Population
      '> FROM Country
      '> WHERE Code = ?
      '> ';
Query OK, 0 rows affected (0.00 sec)
Statement prepared
```

```
mysql> SET @var1 = 'USA';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> EXECUTE namepop USING @var1;
+-----+-----+
| Name          | Population |
+-----+-----+
| United States | 278357000 |
+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> SET @var2 = 'GBR';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> EXECUTE namepop USING @var2;
+-----+-----+
| Name          | Population |
+-----+-----+
| United Kingdom | 59623400  |
+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT @var3 := 'CAN';
+-----+
| @var3 := 'CAN' |
+-----+
| CAN             |
+-----+
1 row in set (0.00 sec)
```

```
mysql> EXECUTE namepop USING @var3;
+-----+
```

```
| Name      | Population |
+-----+-----+
| Canada   | 31147000  |
+-----+-----+
1 row in set (0.00 sec)
```

As mentioned earlier, if you refer to a user variable that has not been initialized, its value is NULL:

```
mysql> EXECUTE namepop USING @var4;
Empty set (0.00 sec)
```

1.6.4. Deallocating Prepared Statements

Because prepared statements are automatically dropped both when they are redefined and when you close the connection to the server, there is rarely any reason to drop them explicitly. However, should you wish to do so (for example, to free memory on the server side), use the `DEALLOCATE PREPARE` statement:

```
mysql> DEALLOCATE PREPARE namepop;
Query OK, 0 rows affected (0.00 sec)
```

MySQL also provides `DROP PREPARE` as an alias for `DEALLOCATE PREPARE`.

1.7. Using Server-Side Help

MySQL Server 4.1 introduces *server-side help* for the `mysql` command-line client. That is, you can perform lookups in the MySQL Reference Manual for a particular topic, right from the command-line. The general syntax for accessing server-side help in the `mysql` client is `HELP keyword`. The keyword contents will show you the top-most entries of the help system:

```
mysql> HELP contents;
You asked for help about help category: "Contents"
For more information, type 'help <item>', where <item> is one of
the following categories:
  Administration
  Column Types
  Data Definition
  Data Manipulation
  Functions
  Geographic features
  Transactions
```

However, you do not need to step through the items listed in the contents list to get help on a specific subject. Suppose that you need to know how to get status information from the server, but can't remember the command. Typing in the following command yields some hints:

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

To get the more specific information offered, you can give the `SHOW` keyword to the `HELP` command:

```
mysql> HELP SHOW
```

```
Name: 'SHOW'
```

```
Description:
```

SHOW has many forms that provide information about databases, tables, columns, or status information about the server.

This section describes those following:

```
SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [LIKE 'pattern']
SHOW CREATE DATABASE db_name
SHOW CREATE TABLE tbl_name
SHOW DATABASES [LIKE 'pattern']
SHOW [STORAGE] ENGINES
SHOW ERRORS [LIMIT [offset,] row_count]
SHOW GRANTS FOR user
SHOW INDEX FROM tbl_name [FROM db_name]
SHOW INNODB STATUS
SHOW [BDB] LOGS
SHOW PRIVILEGES
SHOW [FULL] PROCESSLIST
SHOW STATUS [LIKE 'pattern']
SHOW TABLE STATUS [FROM db_name] [LIKE 'pattern']
SHOW [OPEN] TABLES [FROM db_name] [LIKE 'pattern']
SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern']
SHOW WARNINGS [LIMIT [offset,] row_count]
```

If the syntax for a given SHOW statement includes a LIKE 'pattern' part, 'pattern' is a string that can contain the SQL '%' and '_' wildcard characters. The pattern is useful for restricting statement output to matching values.

Server-side help requires the help tables in the mysql database to be loaded, but that is an administrative matter beyond the scope of this addendum.

1.8. Using INSERT ... ON DUPLICATE KEY UPDATE

MySQL Server 4.1 adds a new clause to the INSERT statement: ON DUPLICATE KEY UPDATE.

Normally, if you attempt to insert a row into a table that would result in a duplicate-key error for a unique-valued index, the insertion fails. In some cases, you can use the REPLACE statement instead, which deletes the old row and inserts the new one in its place.

However, REPLACE is not suitable if you wish to change only some columns of the old row. With the new syntax, you have the option of choosing to update one or more columns of the existing row, rather than letting the INSERT statement fail or replacing the entire row.

This INSERT syntax allows you to do in one statement what otherwise requires two (INSERT and UPDATE). Also, for non-transactional tables, it saves you from having to explicitly lock the table to prevent UPDATE errors when the referenced row may have been deleted in between the INSERT and UPDATE.

One case where this new behavior is especially useful is when you have a table with counters that are tied to key values. When it's time to increment a counter in the record for a given key, you want to create a new record if none exists for the key, but just increment the counter if the key does exist. For example, suppose that we are tracking elephants in the wild and want to count the number of times each elephant has been spotted at a given location. In this case, we can create a log table to log elephant sightings based on the unique key of elephant name and location:

```
mysql> CREATE TABLE log (
  ->   name CHAR(30) NOT NULL,
  ->   location CHAR(30) NOT NULL,
  ->   counter INT UNSIGNED NOT NULL,
  ->   PRIMARY KEY (name, location));
Query OK, 0 rows affected (0.07 sec)
```

Then, every time we wish to log a sighting, we can use `INSERT` without first checking whether the record exists. If we have just created the table, and the first two sightings that occur are for the elephant "Tantor" over by the waterhole, we would use the same `INSERT` statement each time. The first instance of the statement inserts a record and the second causes it to be updated:

```
mysql> INSERT INTO log (name, location, counter)
  -> VALUES ('Tantor', 'Waterhole', 1)
  -> ON DUPLICATE KEY UPDATE counter=counter+1;
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM log;
+-----+-----+-----+
| name  | location | counter |
+-----+-----+-----+
| Tantor | Waterhole | 1 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> INSERT INTO log (name, location, counter)
  -> VALUES ('Tantor', 'Waterhole', 1)
  -> ON DUPLICATE KEY UPDATE counter=counter+1;
Query OK, 2 rows affected (0.00 sec)
```

```
mysql> SELECT * FROM log;
+-----+-----+-----+
| name  | location | counter |
+-----+-----+-----+
| Tantor | Waterhole | 2 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Notice the difference in the "rows affected" value returned by the server for each `INSERT` statement: If a new record is inserted, the value is 1; if an already existing record was updated, the value is 2.

1.9. Using the `GROUP_CONCAT()` Function

`GROUP_CONCAT()` is a new aggregate function in MySQL 4.1. Like other aggregate functions such as `SUM()` and `MAX()`, it is used in grouping operations.

The purpose of the `GROUP_CONCAT()` function is to concatenate column values into a single string. This is useful if you would otherwise perform a lookup of many rows and then concatenate them on the client end. As our initial example, we create lists of the countries that have a particular form of government on the South American continent:

```
mysql> SELECT GovernmentForm, GROUP_CONCAT(Name) AS Countries
  -> FROM Country
  -> WHERE Continent = 'South America'
  -> GROUP BY GovernmentForm\G
***** 1. row *****
GovernmentForm: Dependent Territory of the UK
Countries: Falkland Islands
***** 2. row *****
```

```

GovernmentForm: Federal Republic
  Countries: Argentina,Venezuela,Brazil
***** 3. row *****
GovernmentForm: Overseas Department of France
  Countries: French Guiana
***** 4. row *****
GovernmentForm: Republic
  Countries: Chile,Uruguay,Suriname,Peru,Paraguay,Bolivia,
            Guyana,Ecuador,Colombia
4 rows in set (0.00 sec)

```

With the GROUP_CONCAT() function, as for all aggregate functions, we must specify the GROUP BY clause for a statement, or the "group" is taken to be all records selected by the statement.

The default string separator used by GROUP_CONCAT() is ',' (comma). Records are added to the resulting string in the order in which the database server reads them. To change the separator and the concatenation order, add SEPARATOR and ORDER BY clauses, respectively, within the parentheses. For ORDER BY, you can specify ASC or DESC, just as when you use it in other contexts:

```

mysql> SELECT GovernmentForm,
->         GROUP_CONCAT(Name ORDER BY Name ASC SEPARATOR ' - ')
->         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 - Brazil - Venezuela
***** 3. row *****
GovernmentForm: Overseas Department of France
  Countries: French Guiana
***** 4. row *****
GovernmentForm: Republic
  Countries: Bolivia - Chile - Colombia - Ecuador - Guyana -
            Paraguay - Peru - Suriname - Uruguay
4 rows in set (0.00 sec)

```

The next example for this function returns the continents that contain countries that have a name beginning with "I", as well as the form of government for those countries. The example demonstrates that GROUP_CONCAT() accepts a DISTINCT clause to remove duplicates from the concatenated list. The first query shows what the result looks like without DISTINCT, and the second uses DISTINCT to display each form of government only once:

```

mysql> SELECT Continent,
->         GROUP_CONCAT(GovernmentForm ORDER BY GovernmentForm ASC)
->         AS 'Government Form'
-> FROM Country
-> WHERE Name LIKE 'I%'
-> GROUP BY Continent;

```

Continent	Government Form
Asia	Federal Republic, Islamic Republic, Republic, Republic, Republic
Europe	Republic, Republic, Republic

```

2 rows in set (0.01 sec)

```

```

mysql> SELECT Continent,
->         GROUP_CONCAT(DISTINCT GovernmentForm

```

```

->                                ORDER BY GovernmentForm ASC)
->                                AS 'Government Form'
-> FROM Country
-> WHERE Name LIKE 'I%'
-> GROUP BY Continent;

```

Continent	Government Form
Asia	Federal Republic, Islamic Republic, Republic
Europe	Republic

2 rows in set (0.00 sec)

1.10. Using GROUP BY ... WITH ROLLUP

Another new functionality that can be used with GROUP BY is WITH ROLLUP. Unlike GROUP_CONCAT(), WITH ROLLUP is not an aggregate function, it is a modifier for the GROUP BY clause.

Suppose that you need to generate a listing of the population of each continent, as well as the total of the population on all continents. One way to do this is by running one query to get the per-continent totals and another to get the total for all continents. Another way to get the results requires some application programming: The application can retrieve the per-continent values and sum those to calculate the total population value. By using WITH ROLLUP, you get both the detailed results as well as the total sum of all rows, eliminating the need for multiple queries or extra processing on the client side:

```

mysql> SELECT Continent, SUM(Population) AS pop
-> FROM Country
-> GROUP BY Continent WITH ROLLUP;

```

Continent	pop
Asia	3705025700
Europe	730074600
North America	482993000
Africa	784475000
Oceania	30401150
Antarctica	0
South America	345780000
NULL	6078749450

8 rows in set (0.01 sec)

The difference in the output from this statement compared to one without WITH ROLLUP occurs on the last line, where the Continent value contains NULL and the pop value contains the total sum of all populations.

WITH ROLLUP performs a so-called *super-aggregate operation*: It does *not* simply generate a sum of the numbers that appear in the pop column. Instead, the final line comprises applications of the given aggregate function, as it is written in the SELECT clause, on *every single row selected*.

To illustrate this, consider the following example in which we calculate columns using the AVG() function rather than SUM(). The final rollup line contains the overall average, not the sum of averages. In other words, the rollup line contains the numbers that would appear had there been no grouping columns for the query:

```

mysql> SELECT Continent,
-> AVG(Population) AS avg_pop
-> FROM Country

```

```
-> GROUP BY Continent WITH ROLLUP;
```

Continent	avg_pop
Asia	72647562.7451
Europe	15871186.9565
North America	13053864.8649
Africa	13525431.0345
Oceania	1085755.3571
Antarctica	0.0000
South America	24698571.4286
NULL	25434098.1172

8 rows in set (0.01 sec)

```
mysql> SELECT AVG(Population) AS avg_pop
-> FROM Country;
```

avg_pop
25434098.1172

1 row in set (0.05 sec)

Before MySQL 4.1, getting all the results produced by the first statement in the preceding example would require two separate statements: one to get the per-continent data and one to get the overall totals. For large data sets, this is very inefficient compared to using WITH ROLLUP, which must scan the data only once.

The use of WITH ROLLUP gets more interesting when several columns are grouped at once. When we do this, we get summary results for each column named in the GROUP BY clause, as well as a final summary row:

```
mysql> SELECT Continent, Region,
-> SUM(Population) AS pop,
-> AVG(Population) AS avg_pop
-> FROM Country
-> GROUP BY Continent, Region WITH ROLLUP;
```

Continent	Region	pop	avg_pop
Asia	Eastern Asia	1507328000	188416000.0000
Asia	Middle East	188380700	10465594.4444
Asia	Southeast Asia	518541000	47140090.9091
Asia	Southern and Central Asia	1490776000	106484000.0000
Asia	NULL	3705025700	72647562.7451
Europe	Baltic Countries	7561900	2520633.3333
Europe	British Islands	63398500	31699250.0000
Europe	Eastern Europe	307026000	30702600.0000
Europe	Nordic Countries	24166400	3452342.8571
Europe	Southern Europe	144674200	9644946.6667
Europe	Western Europe	183247600	20360844.4444
Europe	NULL	730074600	15871186.9565
North America	Caribbean	38140000	1589166.6667
North America	Central America	135221000	16902625.0000
North America	North America	309632000	61926400.0000
North America	NULL	482993000	13053864.8649
Africa	Central Africa	95652000	10628000.0000
Africa	Eastern Africa	246999000	12349950.0000
Africa	Northern Africa	173266000	24752285.7143
Africa	Southern Africa	46886000	9377200.0000
Africa	Western Africa	221672000	13039529.4118
Africa	NULL	784475000	13525431.0345

data as the original table. Describe the steps necessary to create the copy.

Q5:

Ulf has created an empty table named `City_Copy` that has the same columns as the `City` table in the `world` database. However, the two tables have slightly different structure, as shown here:

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

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

What statement did Ulf likely use to create the `City_Copy` table?

Q6:

Ulf wants to use a single statement to create a table called `City_Copy2` that has the same columns and data as the `City` table in the `world` database. What statement should he use?

Q7:

Suppose that the following `CREATE TABLE` statement is issued at a time when a single-byte character set is the default:

```
CREATE TABLE dogs (
  dog_name CHAR(30) NOT NULL PRIMARY KEY,
  owner_name CHAR(30)
);
```

- How much space (in bytes) is set aside for character data in each record that is added to the table?
- Suppose that the table is dropped and recreated at a time when the default character set is the two-byte `ucs2` Unicode set. How much space is reserved for the character data in each record as it is added?
- For cases (a) and (b), explain how the character set used for the columns affects key values for any indexes that include those columns.

Q8:

utf8 is a character set in which individual characters take from one to three bytes storage each. If a column that uses the utf8 character set is defined as CHAR(100), how many bytes of storage does each value require?

Q9:

Will the storage requirements for the column in the previous question change if you assign only utf8 strings that consist entirely of single-byte characters to the column?

Q10:

Describe the display format used by MySQL Server 4.1 for TIMESTAMP columns, as compared to the format used for previous versions.

Q11:

In MySQL 4.1, if you issue a CREATE TABLE statement that contains a column defined as TIMESTAMP(10), how does MySQL handle the "(10)" part of the definition that specifies a display width?

Q12:

Is it possible to store NULL values in a TIMESTAMP column? If so, how do you define the column to allow this?

Q13:

What is the effect of specifying a default value of CURRENT_TIMESTAMP for a TIMESTAMP column?

Q14:

What is the effect of specifying the ON UPDATE CURRENT_TIMESTAMP attribute for a TIMESTAMP column?

Q15:

Explain how you would go about creating and maintaining two TIMESTAMP columns named created and updated that meet the following requirements:

- Both the created and updated columns are set to the current time when a record is created.
- The updated column (but not the created column) is set to the current time whenever the record is updated.

Q16:

John is sitting in London, and connects to the company's MySQL 4.1 server. The server's global time zone is the same as John's, namely '+00:00' (UTC). John executes the following statement:

```
mysql> SELECT * FROM timestamps;
+-----+
| ts                |
+-----+
| 2005-03-27 13:30:00 |
+-----+
1 row in set (0.00 sec)
```

Lydia is working in New York, where the time zone is five hours earlier than in London. She wishes to connect to the same server and see the contents of the same table — but adjusted for her own time zone. What statements does Lydia need to execute?

Q17:

John is using a `time_zone` setting of `'+00:00'`. He retrieves a `TIMESTAMP` value with the following result:

```
mysql> SELECT * FROM timestamps;
+-----+
| ts                |
+-----+
| 2005-04-02 06:45:00 |
+-----+
1 row in set (0.00 sec)
```

Lydia connects to the same server, adjusts her time zone, and retrieves the same `TIMESTAMP` value:

```
mysql> SET time_zone = '-05:00';
mysql> SELECT * FROM timestamps;
```

What value will Lydia see?

- a. '2005-04-02 01:45:00'
- b. '2005-04-02 06:45:00'
- c. '2005-04-02 11:45:00'

Q18:

The following query selects those continents that have countries in which more than 50% of the population speak English. Is this an example of using a correlated subquery? Why or why not?

```
SELECT DISTINCT Continent
FROM Country
WHERE Code IN (SELECT CountryCode
               FROM CountryLanguage
               WHERE Language='English'
               AND Percentage>50
              );
```

Q19:

The following statement uses a non-correlated subquery to find the South American country with the smallest population:

```
SELECT * FROM Country
WHERE Continent = 'South America'
AND Population = (SELECT MIN(Population) FROM Country
                  WHERE Continent = 'South America');
```

Rewrite the statement to use a correlated subquery.

Q20:

What is the effect of executing the following query?

```
SELECT Continent, Name
FROM Country c1
WHERE Population >= ALL (SELECT Population
                        FROM Country c2
                        WHERE c1.Continent=c2.Continent
                        );
```

Q21:

What is the effect of executing the following query (compare to the previous exercise)?

```
SELECT Continent, Name
FROM Country
WHERE SurfaceArea > ANY (SELECT AVG(SurfaceArea)
                       FROM Country
                       GROUP BY Continent
                       );
```

Q22:

Using the `world` database, how would you use a subquery to write a `SELECT` statement that answers the following question: What is the largest country (in terms of surface area) on each continent?

Q23:

Using the `world` database, how would you use `IN` and a subquery to write a `SELECT` statement that answers the following question: What languages are spoken in countries where the form of government is a monarchy?

Q24:

Using the `world` database, how would you use `EXISTS` and a subquery to write a `SELECT` statement that answers the following question: In what countries are people that speak German found?

Q25:

How would you use a row constructor to find the population of Houston, Texas, USA?

Q26:

Using a `SELECT` statement with a subquery in the `FROM` clause, find the number of people in the region "Western Europe" that speak German as their mother tongue. The answer must be returned as a scalar.

Q27:

Why is the following use of a subquery in the `FROM` clause not correct?

```
SELECT Name, Language
FROM Country AS c, (SELECT Language
                  FROM CountryLanguage
                  WHERE CountryCode = c.Code
                  ) AS tmp;
```

Q28:

Where in an SQL statement may a scalar subquery be placed?

Q29:

We would like to execute a query on the `world` database that returns the name of a country with the highest number of official languages, the number of official languages in that country, and what those languages are. What should be inserted for each `"..."` in the following statement to accomplish this task?

```
SELECT CONCAT(
    "The country ",
    "... ' ",
    "... '
    " official languages: ",
    ...
);
```

Hint: This exercise covers more subjects discussed in the text than you might initially think. Do test your solution on the `world` database before looking at the answer.

Q30:

Joe wants to create a table of country capitals. He creates the `Capitals` table by copying the structure and data of the `City` table, but then by mistake copies *all* of the data from the `City` table into the `Capitals` table:

```
mysql> CREATE TABLE Capitals LIKE City;
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO Capitals
-> SELECT * FROM City;
Query OK, 4079 rows affected (0.06 sec)
Records: 4079 Duplicates: 0 Warnings: 0
```

The city ID of a country's capital is stored in the `Capital` field of the `Country` table. Using a sub-query, how can Joe remove all the non-capital cities from the `Capitals` table?

Q31:

What is the main advantage of using prepared statements?

Q32:

After you execute the following statements, how many prepared statements exist?

```
PREPARE s1 FROM 'SELECT 1';
PREPARE s2 FROM 'SELECT 2';
PREPARE s1 FROM 'SELECT (1+2)';
```

Q33:

John connects to the server and prepares a statement:

```
mysql> PREPARE s1 FROM 'SELECT Name, Population
-> FROM Country WHERE Continent = ?';
Query OK, 0 rows affected (0.00 sec)
Statement prepared
```

Then Lydia connects to the same server and prepares a statement with the same name:

```
mysql> PREPARE s1 FROM 'SELECT NOW()';
Query OK, 0 rows affected (0.10 sec)
Statement prepared
```

What effect does this have on John's prepared statement?

Q34:

Write a prepared statement that accepts a continent name and a population value as parameters, and uses the parameter values to select, from the `Country` table, the countries located in the given continent that have a population larger than the given population.

Q35:

Use the statement prepared in the previous exercise to determine which countries in Asia have a population of more than 100 million.

Q36:

John connects to the server and prepares a statement:

```
mysql> PREPARE s1 FROM 'SELECT Name, Population
    > FROM Country WHERE Continent = ?';
Query OK, 0 rows affected (0.00 sec)
Statement prepared
```

The John disconnects, reconnects, and issues the following statements. What is the result of the `EXECUTE` statement?

```
mysql> SET @c = 'South America';
mysql> EXECUTE s1 USING @c;
```

Q37:

How do you deallocate a prepared statement? Is it necessary to do so?

Q38:

Using the `mysql` command-line client, how would you look up information on the syntax of `UPDATE` statements?

Q39:

The table `access_log` contains information on the number of times employees of a secured office open a door protected by personal ID number (PIN) codes. The structure of the table is:

Field	Type	Null	Key	Default	Extra
PIN	char(6)		PRI		
entries	int(10) unsigned			0	

The system was been put into use recently, and the table contains the following entries:

PIN	entries
-----	---------

156734	6
578924	2
479645	10
356845	5

Now, two employees enter through the secured door using their PIN codes.

- The first employee uses the PIN code 578924.
- The second employee, who has not used the system before, uses the PIN code 687456 (which is a valid PIN for the door).

How can you log both entries in the `access_log` table using a statement that is the same for each entry (except for the PIN codes)?

Q40:

Use the `GROUP_CONCAT()` function to produce one row per continent that displays a list of the countries in the continent that have a country population of more than 100 million.

Q41:

Using `WITH ROLLUP`, write a single statement that queries the `Country` table to produce counts of the number of countries in each continent, and a count of the total number of countries summed over all continents.

Answers to Exercises

A1:

The `mysql` command-line program can be started with the `--protocol=tcp` command-line option. Alternatively, the `protocol` option can be set to `tcp` in one of the option files that `mysql` reads when it starts.

A2:

If the `InnoDB` storage engine is disabled, MySQL looks up the `storage_engine` system variable, uses its value to determine which engine to use for the table, and issues a warning.

A3:

MySQL generates the warning because the statement uses the `TYPE` keyword rather than `ENGINE` to specify the table's storage engine. In MySQL 4.1, `TYPE` is deprecated in favor of `ENGINE`.

A4:

To create an exact copy of the `Country` table, Ulf first must create an exact copy of the table structure by issuing the following statement:

```
CREATE TABLE Country_Copy LIKE Country;
```

This creates the new `Country_Copy` table, which has the same structure as the `Country` table, but is empty. To copy the data in `Country` to the `Country_Copy` table, Ulf issues the following `INSERT` statement:

```
INSERT INTO Country_Copy SELECT * FROM Country;
```

The `Country_Copy` table is now an exact copy of the `Country` table.

A5:

Ulf probably used the following statement, which creates a new table that has the same columns as the original, but does not preserve indexes and omits some column attributes from the new table definition:

```
CREATE TABLE City_Copy SELECT * FROM City WHERE 0;
```

The `WHERE 0` clause causes no records to be copied to the new table, which as a result is empty.

A6:

Ulf can use the following statement to create and populate a new table that contains the same data as the `City` table:

```
CREATE TABLE City_Copy2 SELECT * FROM City;
```

A7:

- a. For single-byte character sets, one byte per character in the `CHAR` column is used whenever a record is added. Because there are two `CHAR(30)` columns in the table, 60 bytes are required for each record.
- b. For a 2-byte character set such as `ucs2`, two bytes per character in the `CHAR` column are used whenever a record is added. Because there are two `CHAR(30)` columns, 120 bytes are required for the 60 characters in each record.
- c. Each storage engine type has a maximum size for an index entry, and this maximum size is expressed in bytes, not characters. Creating an index on a `CHAR` column that uses a multi-byte character set results in key values that reach the maximum index size limit with fewer characters than would the same column using a single-byte character set.

A8:

For a fixed-length `CHAR(100)` column, MySQL must allocate 100 times the amount of storage required for the largest character in the column's character set. For `utf8`, characters can take up to three bytes, so column values require $100 \times 3 = 300$ bytes each.

A9:

No. MySQL must still allow for the possibility that the column will be used to store strings containing multi-byte characters. On the other hand, were the column to be defined as `VARCHAR(100)`, MySQL would allocate only as much storage as needed for each string, plus one byte to record the number of characters in the string. Thus, `VARCHAR(100)` storage requirements can range from one byte for an empty string to 301 bytes for a 100-character string that consists entirely of three-byte characters.

A10:

The `TIMESTAMP` display format used by MySQL Server 4.1 is the same as the display format used for `DATETIME` data.

A11:

The display width is ignored. For `TIMESTAMP` columns, display widths apply only before MySQL 4.1.

A12:

With MySQL Server 4.1, it is possible to store `NULL` values in a `TIMESTAMP` column.

However, unlike other column types, `TIMESTAMP` columns default to `NOT NULL`, and you must specifically include the `NULL` attribute in the column definition when you create or alter the `TIMESTAMP` column, if you wish to allow storage of `NULL` values.

A13:

The effect of specifying a default value of `CURRENT_TIMESTAMP` for a `TIMESTAMP` column is that whenever a new record is created, that column is set to the record creation time.

A14:

The effect of specifying the `ON UPDATE CURRENT_TIMESTAMP` attribute for a `TIMESTAMP` column is that the column is updated to the current timestamp value whenever any other column in the record is changed.

A15:

A table with two `TIMESTAMP` columns that meet the requirements can be created as follows:

```
mysql> CREATE TABLE updates (
->   created TIMESTAMP DEFAULT 0,
->   updated TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
->   data CHAR(30)
-> );
Query OK, 0 rows affected (0.01 sec)
```

For record creation, to set *both* of the `TIMESTAMP` columns to the current time, we use the `INSERT` statement to insert `NULL` into *each* column:

```
mysql> INSERT INTO updates (created, updated, data)
-> VALUES (NULL, NULL, 'original_value');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM updates;
+-----+-----+-----+
| created                | updated                | data                |
+-----+-----+-----+
| 2005-01-05 12:50:40    | 2005-01-05 12:50:40    | original_value     |
+-----+-----+-----+
1 row in set (0.00 sec)
```

For updates, we do not need to do anything special to update the `updated` column because it already has the `ON UPDATE CURRENT_TIMESTAMP` attribute:

```
mysql> UPDATE updates SET data='updated_value';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> SELECT * FROM updates;
+-----+-----+-----+
| created                | updated                | data                |
+-----+-----+-----+
| 2005-01-05 12:50:40    | 2005-01-05 12:50:46    | updated_value     |
+-----+-----+-----+
```

1 row in set (0.00 sec)

A16:

For Lydia to see the data adjusted for her own time zone after establishing a session with the server, she must execute the statement `SET time_zone = '-05:00'` before she performs the `SELECT`:

```
mysql> SET time_zone = '-05:00';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT * FROM timestamps;
```

```
+-----+
| ts                |
+-----+
| 2005-03-27 08:30:00 |
+-----+
1 row in set (0.00 sec)
```

A17:

Lydia adjusted her time zone back five hours from the time zone used by John, so the value displayed for the `TIMESTAMP` value will be five hours earlier the value John sees: `'2005-04-02 01:45:00'`.

A18:

The example shown is *not* an example of a correlated subquery because the subquery can be resolved completely without regard to the outer query. In a correlated subquery, the inner `SELECT` is dependent on the outer query.

A19:

```
SELECT * FROM Country c1
WHERE Continent = 'South America'
AND Population = (SELECT MIN(Population) FROM Country c2
                  WHERE c2.Continent = c1.Continent);
```

A20:

The query returns, for each continent, the country whose population is greater than or equal to the population of every country on the same continent. In other words, it returns the country with the greatest population on each continent.

A21:

The query returns all the countries that have a surface area larger than the average surface area of any continent. The difference between this and the earlier query is that this is not a correlated subquery, so it is not restricted to comparing only to the surface area of the continent where the country is located.

A22:

In the following query, the subquery returns the largest surface area found on each continent. This information is then used in the outer query to find the country on the same continent that has that surface area:

```
mysql> SELECT Continent C, Name, SurfaceArea
-> FROM Country
-> WHERE SurfaceArea = (
-> SELECT MAX(SurfaceArea)
```

```

-> FROM Country
-> WHERE Continent = C);
+-----+-----+
| C      | Name                | SurfaceArea |
+-----+-----+-----+
| Oceania | Australia           | 7741220.00  |
| South America | Brazil             | 8547403.00  |
| North America | Canada             | 9970610.00  |
| Asia    | China               | 9572900.00  |
| Africa  | Sudan               | 2505813.00  |
| Europe  | Russian Federation | 17075400.00 |
| Antarctica | Antarctica         | 13120000.00 |
+-----+-----+-----+
7 rows in set (0.12 sec)

```

A23:

In the following query, the subquery returns the country code of countries where the government form is monarchy. This information is used in the outer query to find the languages spoken in those countries.

```

mysql> SELECT DISTINCT Language
-> FROM CountryLanguage
-> WHERE CountryCode IN (
-> SELECT Code
-> FROM Country
-> Where GovernmentForm = "Monarchy");

```

```

+-----+
| Language |
+-----+
| Asami    |
| Dzongkha |
| Nepali   |
| Arabic   |
| Urdu     |
| Swazi    |
| Zulu     |
| English  |
| Tongan   |
+-----+
9 rows in set (0.02 sec)

```

A24:

In the following query, the subquery returns all the languages spoken in a given country. This information is then used with the EXISTS predicate to determine if German is one of the languages spoken in that country.

```

mysql> SELECT Code c, Name
-> FROM Country
-> WHERE EXISTS (SELECT *
-> FROM CountryLanguage
-> WHERE CountryCode = c
-> AND Language = 'German');

```

```

+-----+-----+
| c      | Name                |
+-----+-----+
| AUS    | Australia           |
| BEL    | Belgium             |
| BRA    | Brazil              |
| ITA    | Italy                |
| AUT    | Austria             |
| CAN    | Canada              |
+-----+-----+

```

```

+-----+-----+
| KAZ | Kazakstan |
| LIE | Liechtenstein |
| LUX | Luxembourg |
| NAM | Namibia |
| PRY | Paraguay |
| POL | Poland |
| ROM | Romania |
| DEU | Germany |
| CHE | Switzerland |
| DNK | Denmark |
| CZE | Czech Republic |
| HUN | Hungary |
| USA | United States |
+-----+-----+
19 rows in set (0.00 sec)

```

A25:

The following query uses a row constructor to look up the population of Houston, Texas, USA in the City table:

```

mysql> SELECT Population
      -> FROM City
      -> WHERE ('Houston', 'Texas', 'USA') = (Name, District, CountryCode);
+-----+
| Population |
+-----+
| 1953631 |
+-----+
1 row in set (0.03 sec)

```

A26:

The answer must be returned as a scalar value, that is, as a result set with a single row and a single column. The following query accomplishes the task:

```

mysql> SELECT SUM(Speakers)
      -> FROM (SELECT (Percentage/100) * Population AS Speakers
      -> FROM CountryLanguage cl, Country c
      -> WHERE cl.CountryCode = c.Code
      -> AND c.Region = "Western Europe"
      -> AND cl.Language = "German"
      -> ) AS tmp;
+-----+
| SUM(Speakers) |
+-----+
| 87156001.998 |
+-----+
1 row in set (0.00 sec)

```

A27:

The server returns ERROR 1109 (42S02): Unknown table 'c' in where clause if you try to execute this query. Subqueries in the FROM clause of a query cannot be correlated with the outer query.

A28:

A scalar subquery may be placed anywhere in an SQL statement that a scalar value, such as a true scalar, an argument to a function call, a term of a mathematical expression, and so forth, is expected.

A29:

For each of the missing parameters in the CONCAT function, we can insert a scalar subquery that returns the required data.

First, we consider how to find the name of the country with the highest number of official languages, and the number of official languages. The following query will give us the maximum number of official languages, and the country name that goes along with it:

```
SELECT Name, COUNT(*) AS nlanguages
FROM Country c, CountryLanguage cl
WHERE c.Code = cl.CountryCode
      AND cl.IsOfficial = 'T'
GROUP BY Name
ORDER BY nlanguages DESC
LIMIT 1;
```

However, there are a few problems with this approach that must be resolved. First of all, the preceding query does not give us a list of the official languages in the country, just the number of them. To find out what the languages are, we utilize the GROUP_CONCAT function as part of the SELECT statement:

```
SELECT Name, COUNT(*) AS nlanguages,
      GROUP_CONCAT(Language) as languages
FROM Country c, CountryLanguage cl
WHERE c.Code = cl.CountryCode
      AND cl.IsOfficial = 'T'
GROUP BY Name
ORDER BY nlanguages DESC
LIMIT 1;
```

The second problem with our approach so far is that a scalar subquery may return only a single column. The query just presented returns three. We can solve this problem by nesting the subquery once again:

```
SELECT Name
FROM (SELECT Name, COUNT(*) AS nlanguages,
      GROUP_CONCAT(Language) as languages
      FROM Country c, CountryLanguage cl
      WHERE c.Code = cl.CountryCode
            AND cl.IsOfficial = 'T'
      GROUP BY Name
      ORDER BY nlanguages DESC
      LIMIT 1
      ) AS tmp;
```

For the number of languages and the list of languages, we can utilize the same method, of course replacing Name with nlanguages or languages in the outer SELECT as required.

The last problem we need to resolve is that, in the CountryLanguage table, there are *two* countries with the maximum number of official languages: South Africa and Switzerland each have four official languages.

Choosing an arbitrary one of these as input for our surrounding CONCAT function will not invalidate the requirement in the question. The result for the nlanguages column will, in either case, always be correct. However, since there is no guarantee that either Switzerland or South Africa will be the country selected every time the subquery is evaluated, we run the risk of presenting the wrong combination of country name and list of languages. To resolve this, we place an extra condition on the ORDER BY clause, which (rather arbitrarily) does a secondary sort in ascending order by the country name.

The complete statement thus looks like this:

```

SELECT
  CONCAT(
    "The country ",
    (SELECT Name FROM (
      SELECT Name, COUNT(*) AS nlanguages,
             GROUP_CONCAT(Language) as languages
      FROM Country c, CountryLanguage cl
      WHERE c.Code = cl.CountryCode
            AND cl.IsOfficial = 'T'
      GROUP BY Name
      ORDER BY nlanguages DESC, Name
      LIMIT 1
    ) AS tmp
  ),
  " has ",
  (SELECT nlanguages FROM (
    SELECT Name, COUNT(*) AS nlanguages,
           GROUP_CONCAT(Language) as languages
    FROM Country c, CountryLanguage cl
    WHERE c.Code = cl.CountryCode
          AND cl.IsOfficial = 'T'
    GROUP BY Name
    ORDER BY nlanguages DESC, Name
    LIMIT 1
  ) AS tmp1
),
  " official languages: ",
  (SELECT languages FROM (
    SELECT Name, COUNT(*) AS nlanguages,
           GROUP_CONCAT(Language) as languages
    FROM Country c, CountryLanguage cl
    WHERE c.Code = cl.CountryCode
          AND cl.IsOfficial = 'T'
    GROUP BY Name
    ORDER BY nlanguages DESC, Name
    LIMIT 1
  ) AS tmp2
);

```

A30:

The subquery in the following statement searches the `City` table to identify all capital cities. The outer statement deletes all rows in the table `Capitals` that are not found by the subquery:

```

mysql> DELETE FROM Capitals
      -> WHERE ID NOT IN (SELECT Capital
      ->                      FROM Country
      ->                      WHERE Capital IS NOT NULL);
Query OK, 3847 rows affected (1.19 sec)

```

A31:

Prepared statements can be more efficient if the same statement is run several times because there is less network traffic and less time is spent parsing the same statement multiple times.

A32:

After the first two PREPARE statements, two prepared statements exist (s1 and s2). The third state-

ment causes the original s1 to be discarded because it uses the same statement name, but does not result in a new prepared statement because the statement contains a syntax error. Only s2 exists after all three PREPARE statements have been executed.

A33:

There is no effect. Prepared statements are specific to the session in which they are created. Statements prepared by one client do not affect those prepared by other clients.

A34:

```
mysql> PREPARE spop FROM 'SELECT Name, Population
    '> FROM Country
    '> WHERE Continent = ? AND Population > ?';
Query OK, 0 rows affected (0.00 sec)
Statement prepared
```

A35:

```
mysql> SET @c = 'Asia', @p = 100000000;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> EXECUTE spop USING @c, @p;
```

```
+-----+-----+
| Name          | Population |
+-----+-----+
| Bangladesh    | 129155000 |
| Indonesia     | 212107000 |
| India         | 1013662000|
| Japan         | 126714000 |
| China         | 1277558000|
| Pakistan      | 156483000 |
+-----+-----+
6 rows in set (0.00 sec)
```

A36:

An error occurs for the EXECUTE statement because prepared statement s1 does not exist. Statements prepared during a session are discarded when the session ends and are not available to later sessions.

A37:

To deallocate a prepared statement, use DEALLOCATE PREPARE:

```
DEALLOCATE PREPARE my_stmt;
```

DROP PREPARE can also be used.

It is not necessary to deallocate a prepared statement created within a given session because the server discards the statement automatically when the session ends. However, deallocating the statement explicitly does allow the server to release resources earlier that are associated with the statement.

A38:

The command `HELP UPDATE` will give you the information you're looking for.

A39:

For the first employee, an UPDATE statement is needed, to increase the count of the times PIN code

578924 has been used to open the door. For the second employee, a new record must be entered into the table, as PIN code 687456 is being used for the first time.

You can provide for both occurrences by utilizing the ON DUPLICATE KEY UPDATE clause of the INSERT statement:

```
INSERT INTO access_log (PIN, entries)
VALUES ("578924", 1)
ON DUPLICATE KEY UPDATE entries = entries+1;

INSERT INTO access_log (PIN, entries)
VALUES ("687456", 1)
ON DUPLICATE KEY UPDATE entries = entries+1;
```

After these two statements are executed, the access_log table has the following contents:

PIN	entries
156734	6
578924	3
479645	10
356845	5
687456	1

A40:

```
mysql> SELECT Continent, GROUP_CONCAT(Name)
-> FROM Country
-> WHERE Population > 100000000
-> GROUP BY Continent;
```

Continent	GROUP_CONCAT(Name)
Asia	Bangladesh,Pakistan,China,Japan,India,Indonesia
Europe	Russian Federation
North America	United States
Africa	Nigeria
South America	Brazil

5 rows in set (0.00 sec)

A41:

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

Continent	COUNT(*)
Asia	51
Europe	46
North America	37
Africa	58
Oceania	28
Antarctica	5
South America	14
NULL	239

8 rows in set (0.00 sec)

Chapter 2. Updates to the Professional Exam

In this chapter, we look only at the updates that are of interest to those seeking MySQL Professional certification. Thus, this chapter is not a complete overview of all features changed or updated in the MySQL server and related products.

2.1. Choosing Index Types for MEMORY tables

Starting with MySQL 4.1, it is possible to specify one of several index types (for example, during table creation). Although the syntax has been implemented for several different storage engines, the only engine to which this feature is currently applicable is the MEMORY engine. In previous versions of MySQL, the MEMORY engine was known as the HEAP storage engine.

MEMORY tables use hash indexes by default. This index type provides very fast lookups for all operations that use a unique index. For non-unique indexes, however, operations that change the indexed values (including DELETE statements) can become relatively slow when there are many duplicate index values.

If you will have only unique indexes on a MEMORY table, you should create them as HASH indexes. Because HASH indexes are the default for MEMORY tables, you can do so when defining an index either by specifying an explicit USING HASH clause or by omitting the index type specification entirely. The following two statements are equivalent:

```
CREATE TABLE lookup (  
    id INT,  
    INDEX USING HASH (id)  
) ENGINE = MEMORY;
```

```
CREATE TABLE lookup (  
    id INT,  
    INDEX (id)  
) ENGINE = MEMORY;
```

If, however, you need to create a MEMORY table with one or more non-unique indexes, and you expect that there will be many duplicate values in the index key, you should create the table with a BTREE index.

```
CREATE TABLE lookup (  
    id INT,  
    INDEX USING BTREE (id)  
) ENGINE = MEMORY;
```

If you have already created the table, you can add the new index using the CREATE INDEX statement, making use of the USING *index_type* clause. The following two statements create a table without an index, and then add a BTREE index:

```
CREATE TABLE lookup (id INT) ENGINE = MEMORY;  
CREATE INDEX id_idx USING BTREE ON lookup (id);
```

Although choosing between alternate index types is limited to MEMORY tables in MySQL 4.1, work on extending this functionality to other storage engines, such as MyISAM and InnoDB, is ongoing.

2.2. The Cluster Storage Engine

MySQL Server 4.1 sports a completely new storage engine called *NDBCluster*. Using the cluster engine is complex, and for the purposes of MySQL 4 certification you are not expected to know the details of how to set up and use *NDBCluster*. You *are*, however, expected to know the general properties of the cluster engine as compared to other storage engines.

In literature, you will see the two terms *NDB Cluster* (or just *NDB*) and *MySQL Cluster*. *NDB Cluster* refers to the cluster technology and is thus specific to the storage engine itself, whereas *MySQL Cluster* refers to a group of one or more MySQL servers that is working as a "front end" to the *NDB Cluster* engine. That is, a *MySQL Cluster* consists of a group of one or more server hosts, each of which is usually running multiple processes that include MySQL servers, *NDB* management processes, and *NDB* database storage nodes. Cluster processes are also referred to as *cluster nodes*, or just *nodes*.

The cluster engine does not run internally in MySQL Server, but is, instead, one or more separate processes running outside MySQL Server (perhaps even on different server hosts). In effect, MySQL Server provides the SQL interface to the cluster processes. From the perspective of the server however, *NDBCluster* is just another storage engine, like the *MyISAM* and the *InnoDB* engines.

NDB Cluster consists of several database processes (nodes) running on one or more physical server hosts. It manages one or more *in-memory* databases in a *shared-nothing system*. In-memory means that all the information in each database is kept in the RAM of the machines making up the cluster. Shared-nothing means that the cluster is set up in such a way that no hardware components (such as disks) are shared among two nodes.

The *NDB* cluster engine is a *transactional* storage engine, like the *InnoDB* and *BDB* storage engines.

The main reasons to consider using *MySQL Cluster* are:

- *High availability*: All records are available on several nodes. If one node fails (because, for example, the server host stops working), the same data can be gotten from another node. Spreading copies of the data across multiple nodes also makes it possible to have replicas of the data in two or more widely distributed locations.
- *Scaleability*: If the load becomes too high for the current set of nodes, extra nodes can be added and the system will reconfigure itself to make data available on more nodes, reducing the load on each individual node.
- *High performance*: All records are stored in memory, making retrieval of data extremely fast. This does not mean that information is lost if the cluster is shut down (as is the case for tables created with the *MEMORY* storage engine). All updates are written to disk, and are available when the cluster is restarted.

2.3. Using SHOW WARNINGS

Warnings are generated whenever the MySQL server is not able to fully comply with a request and when an action has possibly unintended side effects. The following example shows how warnings are generated when we attempt to insert a character string, a negative integer, and *NULL* into a column that is defined as *INT UNSIGNED NOT NULL*:

```
mysql> CREATE TABLE integers (  
->   i INT UNSIGNED NOT NULL  
-> );  
Query OK, 0 rows affected (0.01 sec)  
  
mysql> INSERT INTO integers
```

```
-> VALUES ('abc'), (-5), (NULL);
Query OK, 3 rows affected, 3 warnings (0.00 sec)
Records: 3 Duplicates: 0 Warnings: 3
```

The server warns us that there were three instances where it had to truncate or otherwise change our submitted values to accept the data that was passed during the `INSERT`. In previous versions of MySQL, you could not get specific information regarding which errors occurred, or even the rows for which they occurred. Starting with MySQL Server 4.1, you can use `SHOW WARNINGS` to ask the server to return the list of warnings that were generated. The following example shows the warnings generated by the preceding `INSERT` statement:

```
mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 1265
Message: Data truncated for column 'i' at row 1
***** 2. row *****
Level: Warning
Code: 1264
Message: Data truncated; out of range for column 'i' at row 2
***** 3. row *****
Level: Warning
Code: 1263
Message: Data truncated; NULL supplied to NOT NULL column 'i' at row 3
3 rows in set (0.00 sec)
```

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

```
mysql> SHOW WARNINGS LIMIT 1,2\G
***** 1. row *****
Level: Warning
Code: 1264
Message: Data truncated; out of range for column 'i' at row 2
***** 2. row *****
Level: Warning
Code: 1263
Message: Data truncated; NULL supplied to NOT NULL column 'i' at row 3
2 rows in set (0.00 sec)
```

Warnings generated by one statement are available from the server only for a limited time (until you issue another statement that can generate warnings). You should always fetch warning messages as soon as you detect that warnings were generated.

2.4. Exercises

Many of the exercises in this book are based on the world database. You can download this database from the MySQL documentation main page (<http://dev.mysql.com/doc>), install it on your system, and use it to solve the exercises.

Q1:

You want to create a copy of the table `Country` using the `MEMORY` storage engine. There is only one index on `Country`, namely `Code`, which is the primary key.

You will be doing several `INSERT` and `DELETE` operations on the new table. Should you change the default index type for the primary key?

Q2:

You want to create a table with the same structure as `CountryLanguage`, but using the `MEMORY` storage engine. This is done using the following two statements:

```
CREATE TABLE memCountryLanguage LIKE CountryLanguage;
ALTER TABLE memCountryLanguage ENGINE=MEMORY;
```

There is currently only one index on `CountryLanguage`, namely the primary key which is indexed across (`CountryCode`, `Language`). Once the table is created, you will be doing several `INSERT` and `DELETE` operations on the new table.

The `memCountryLanguage` table is needed to do fast lookups in the `Language` column, so you want to add a separate, non-unique index on that column. Is it better to choose the `HASH` or the `BTREE` index type for this index? What statement would you use to create the index?

Q3:

Why is MySQL Cluster considered to be a high-availability solution?

Q4:

When records are retrieved from the MySQL Cluster, are they read from disk on the node that provides the data? Are updates to data on a node saved to disk?

Q5:

The following `CREATE TABLE` statement produces a warning. What statement would you use to see what caused the warning?

```
mysql> CREATE TABLE dogs (
->   dog_id INT UNSIGNED NOT NULL PRIMARY KEY,
->   dog_name CHAR(30),
->   owner_name CHAR(30)
-> ) TYPE=InnoDB;
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

Q6:

The following series of statements causes the MySQL server to emit a total of four warnings:

```
mysql> CREATE TABLE cats (
->   cat_id TINYINT,
->   cat_name CHAR(7)
-> ) TYPE=MyISAM;
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> INSERT INTO cats (cat_id, cat_name)
-> VALUES ('1', 'Garfield');
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> INSERT INTO cats (cat_id, cat_name)
-> VALUES ('200', 'Scratchy');
Query OK, 1 row affected, 2 warnings (0.00 sec)
```

When `SHOW WARNINGS` is used to see the warnings generated by those statements, only two warnings are shown:

```
mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 1264
Message: Data truncated; out of range for column 'cat_id' at row 1
***** 2. row *****
Level: Warning
Code: 1265
Message: Data truncated for column 'cat_name' at row 1
2 rows in set (0.00 sec)
```

Why do we see only two of the four warnings? Is there any way to see what the other two warnings were?

Answers to Exercises

A1:

The default index type for MEMORY tables is HASH, which is well suited for unique indexes but not as well suited for non-unique indexes. Because the primary key, by definition, is unique, there is no reason to change the index to another index type.

A2:

Because the Language column is non-unique and we expect many INSERT and DELETE operations on the table, the BTREE index type should be preferred. The statement to add the index could be the following:

```
CREATE INDEX Language USING BTREE ON memCountryLanguage (Language);
```

A3:

In a MySQL Cluster, all records are available on several nodes. If one node fails, availability is not compromised because another node can take over and provide the needed data.

A4:

A MySQL Cluster node holds all the available data in RAM, so no disk reads are performed when retrieving data. All data updates *are* written to disk, so, in the event that the node shuts down, data can be re-read from disk when the node restarts.

A5:

The SHOW WARNINGS statement returns details about the warning generated by the previous statement:

```
mysql> SHOW WARNINGS\G
***** 1. row *****
Level: Warning
Code: 1287
Message: 'TYPE=storage_engine' is deprecated;
        use 'ENGINE=storage_engine' instead
1 row in set (0.01 sec)
```

A6:

Warnings generated by an SQL statement are available only until you issue another statement that can generate warnings. You must check for warnings and fetch any messages from the server immediately

after each statement that generates a warning or the information is lost. Short of re-running the statements, there is no way to retrieve previously generated warnings.

Chapter 3. Errata for the MySQL Certification Study Guide

The following are the known errata for the printed *MySQL Certification Study Guide* at the time of publishing of this addendum. You may wish to check whether more have appeared on <http://www.mysql.com/certification/studyguides>.

- Although reading the exam content layout soon makes it apparent, the layout of the book does not make a clear distinction between the chapters referring to the Core exam and the Professional exam. Chapters 1-9 contain the material related to the Core exam; chapters 10-16 contain the material related to the Professional exam.
- Page 12: The `world` database that is used for examples in several places in the book may be downloaded from <http://dev.mysql.com/doc>.
- Page 37, 2nd line from the bottom: `Country` should read `CountryCode`.
- Page 45: The first two `shell>` examples: replace `--fields-terminated-by` with `-fields-enclosed-by`.
- Page 69: In the `CREATE TABLE` statement: Delete the comma in `CountryCode CHAR(3) NOT NULL,`
- Page 99, 4th bullet point: "Handing" should be "Handling"
- Page 100: In the table of conversions, 5th row, the `INT` and `DECIMAL` conversions should both read 1978 rather than 1970.
- Page 128, answer 57 should read: "The value inserted is '0012-00-00'. MySQL interprets the inserted value as a date, '12-00-00', which is interpreted as the year '12'."
- Page 131, answer 69: The parenthesized note should read: "(Although `creation` is declared as `NULL`, MySQL will never assign a `NULL` value to a `TIMESTAMP` column, but use the "zero" value instead.)"
- Page 139, answer 110: The first statement should read: `SHOW TABLES LIKE '%test%';`
- Page 152, 2nd line: `(1+2+8)` should read `(1+2+4)`.
- Page 178, answer 12c: Remove "`AS Species`"; in the rest of the text, "`species`" should appear with lowercase 's'.
- Page 180: In answer 16, The first `SELECT` statement shown should include the keyword `ALL` for both `UNION` statements.
- Page 200: After "The other aggregate functions always ignore `NULL` values", add: ", except where the values being aggregated are all `NULL`". In the example immediately following, the output should be `(9, 7, NULL, NULL)` rather than `(9, 7, 0, 0)`.
- Page 201: Delete the paragraph that begins "Note that `SUM() ...`", just before the heading of section 6.7.
- Page 218: The result set shown on lines 3-7 should include the `name` column:

```
+-----+-----+
```

name	COUNT(*)
Lennart	4

- Page 265, 1st bullet point: "CountryLanguage and CountryLanguage tables." should read "CountryLanguage and Country tables."
- Page 282, question 10: This question/answer pair contains several errors. Both should be struck from the text.
- Page 283-285, questions 12-16: In the `project` table in the row where `pid` is 10020, the value of `id` should be 110 (rather than 100).
- Page 291, answer 6: "WHERE START BETWEEN..." should read "WHERE p.start BETWEEN..."
- Page 292, answer 10: This question/answer pair contains several errors. Both should be struck from the text.
- Page 307, 3rd bullet point, line 1: ('/') should be ('\').
- Appendix B: Several minor changes have been made in the on-line version of the *MySQL Certification Candidate Guide*, to reflect the fact that the Study Guide is now available.