The Definitive Guide to MySQL5

Third Edition

Michael Kofler
Translated By David Kramer

Apress®

The Definitive Guide to MySQL 5

Copyright © 2005 by Michael Kofler

ISBN (pbk): 1-59059-535-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Translator and Editor: David Kramer
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: April Eddy
Artist: Kinetic Publishing Services, LLC
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 8

Database Design

The first stage in any database application is the design of the database. The design will have a great influence on the efficiency of the application, the degree of simplicity or difficulty in programming and maintenance, and the flexibility in making changes in the design. Errors that occur in the design phase will come home to roost in the heartache of future efforts at correction. But don't expect any easy recipe! Database design has a great deal to do with experience, and in a single chapter one can do no more than present some pointers to get you started.

This chapter discusses the fundamentals of relational databases, summarizes the different data and table types available under MySQL, and demonstrates, using the *mylibrary* database, the application of normalization rules. (The *mylibrary* database is used to manage a collection of books, and authors and publishers will be used in many of the examples in the book.) Additional topics are the correct use of indexes and integrity rules (foreign key constraints).

Tip This chapter serves as the basis for database design, but it does not explain how to create a new database and its associated tables. There are two ways of doing this:

The easiest is to use an interface such as MySQL Administrator or phpMyAdmin (see Chapters 5 and 6). This allows you to define the properties of new tables with a few clicks of the mouse.

The other possibility is to create databases and tables with SQL commands (e.g., *CREATE TABLE name…*; see Chapter 9). To be sure, the formulation of such commands is relatively tedious, but the advantage is that such commands can also be executed in a PHP script, which can be useful if you want to create temporary tables.

Further Reading

There are countless books that deal exclusively, independently of any specific database system, with database design and SQL. Needless to say, there is a variety of opinion as to which of these books are the good ones. Therefore, consider the following recommendations as my personal hit list:

Joe Celko: *SQL for Smarties*, Morgan Kaufmann Publishers, 1999. (This is not a book for SQL beginners. Many examples are currently not realizable in MySQL, because MySQL is not sufficiently compatible with ANSI-SQL/92. Nonetheless, it is a terrific example-oriented book on SQL.)

Judith S. Bowman et al.: *The Practical SQL Handbook*, Addison-Wesley, 2001.

Michael J. Hernandez: *Database Design for Mere Mortals*, Addison-Wesley, 2003. (The first half is somewhat long-winded, but the second half is excellent and very clearly written.)

Another book that is frequently recommended, but with which I am not familiar, is Peter Gulutzan and Trudy Pelzer's *SQL-99 Complete, Really*, R&D Books, 1999.

If you are not quite ready to shell out your hard-earned cash for yet another book and you are interested for now in database design only, you may find the compact introduction on the design of relational databases by Fernando Lozano adequate to your needs. See `http://www.edm2.com/0612/msql7.html`.

Table Types

A peculiarity of MySQL is that when you create a new table, you can specify its type. MySQL supports a number of table types, distinguished by a variety of properties. The three most important types are MyISAM, InnoDB, and HEAP.

If you do not specify the type when you create a table, then the MySQL server decides for you, based on its configuration, either for MyISAM or InnoDB. The default type is set with the option `default-table-type` in the MySQL configuration file.

This section provides a brief look at the different table types recognized by MySQL as well as some of their properties and when the use of one type or another is appropriate.

MyISAM Tables

The MyISAM table type is mature, stable, and simple to manage. If you have no particular reason to choose another type, then you should use this type. Internally, there are two variants of this table type, and the MySQL server chooses the appropriate type on its own:

MyISAM Static: These tables are used when all columns of the table have fixed, predetermined size. Access in such tables is particularly efficient. This is true even if the table is frequently changed (that is, when there are many *INSERT*, *UPDATE*, and *DELETE* commands). Moreover, data security is quite high, since in the case of corrupted files or other problems, it is relatively easy to extract records.

MyISAM Dynamic: If in the declaration of a table there is also only a single *VARCHAR*, *xxxTEXT*, or *xxxBLOB* field, then MySQL automatically selects this table type. The significant advantage over the static MyISAM variant is that the space requirement is usually significantly less: Character strings and binary objects require space commensurate with their actual size (plus a few bytes overhead).

However, it is a fact that data records are not all the same size. If records are later altered, then their location within the database file may have to change. In the old place there appears a hole in the database file. Moreover, it is possible that the fields of a record are not all stored within a contiguous block within the database file, but in various locations. All of this results in increasingly longer access times as the edited table becomes more and more fragmented, unless an *OPTIMIZE TABLE* or an optimization program is executed every now and then (`myisamchk`; see Chapter 14).

MyISAM Compressed: Both dynamic and static MyISAM tables can be compressed with the auxiliary program `myisamchk`. This usually results in a shrinkage of the storage requirement for the table to less than one-half the original amount (depending on the contents of the table). To be sure, thereafter, every data record must be decompressed when it is read, but it is still possible under some conditions that access to the table is nevertheless faster, particularly with a combination of a slow hard drive and fast processor.

The decisive drawback of compressed MyISAM tables is that they cannot be changed (that is, they are read-only tables).

InnoDB Tables

In addition to the MyISAM format, MySQL supports a second table format, namely InnoDB. This is a modern alternative to MyISAM, which above all offers the following additional functions:

Transactions: Database operations in InnoDB tables can be executed as transactions. This allows you to execute several logically connected SQL commands as a single entity. If an error occurs during execution, then all of the commands (not only the one during which the error occurred) are nullified. In addition, transactions offer other advantages that improve the security of database applications.

Transactions can be executed in all four isolation levels of the ANSI-SQL/92 standard (*READ UNCOMMITTED*, *READ COMMITTED*, *REPEATABLE READ*, *SERIALIZABLE*; see also Chapter 10).

Row Level Locking: In implementing transactions, the InnoDB table driver uses internal *row level locking*. This means that during a transaction, the entire table does not have to be blocked for access by other users (which is the case for a MyISAM table during a *LOCK TABLE* command), but only the data records are actually affected. If many users are simultaneously making changes on a large table, row level locking can bring about an enormous advantage in efficiency.

The InnoDB table driver automatically recognizes *deadlocks* (that is, the condition in which two processes mutually block each other) and in such a case, terminates one of the two processes automatically.

Foreign Key Constraints: When you define relations between your tables, the InnoDB table driver automatically ensures that the referential integrity of the table is preserved after *DELETE* commands. Thus it is impossible, for example, for a record in table A to refer to a no longer existing table B. (In database lingo this function is called a *foreign key constraint*.)

Crash Recovery: After a crash, InnoDB tables are automatically and very quickly returned to a consistent state (provided that the file system of the computer was not damaged). I have not tested this functionality.

The InnoDB table driver has been an integral component of MySQL since version 3.23.34. The development of the table driver and its commercial support come from the independent company Innobase (see `http://www.innodb.com`).

Limitations and Drawbacks

If InnoDB tables had only advantages and no drawbacks, one could dump MyISAM tables in the garbage and be done with it. But that is not the case, as the following list reveals:

Tablespace Administration: While with the MyISAM table driver, each table is stored in its own file, which grows or shrinks as required, the InnoDB table driver stores all data and indexes in a *tablespace*, comprising one or more files, that forms a sort of virtual file system. These files cannot later be made smaller. Nor is it possible to stop the MySQL server and then copy a table by simply copying its file. Therefore, in the administration of InnoDB tables, the command `mysqldump` must be employed more frequently than with MyISAM tables.

Record Size: A data record can occupy at most 8000 bytes. This limit does not hold for *TEXT* and *BLOB* columns, of which only the first 512 bytes are stored in the database proper. Data in such columns beyond this size are stored in separate pages of the master space.

Storage Requirement: The storage requirements for InnoDB tables are much greater than those for equivalent MyISAM tables (up to twice as big).

Full-Text Index: For InnoDB tables one cannot use a full-text index.

GIS Data: Geometric data cannot be stored in InnoDB tables.

COUNT Problem: On account of open transactions, it is relatively difficult for the InnoDB table driver to determine the number of records in a table. Therefore, executing a *SELECT COUNT(*) FROM TABLE* is much slower than with MyISAM tables. This limitation should be eliminated soon.

Table Locking: InnoDB uses its own locking algorithms in executing transactions. Therefore, you should avoid *LOCK TABLE … READ/WRITE*. Instead, you should use *SELECT … IN SHARE MODE* or *SELECT … FOR UPDATE*. These commands have the additional advantage that they block only individual records and not the entire table. For future versions of MySQL, the InnoDB-specific commands *LOCK TABLE … IN SHARE MODE* and *LOCK TABLE … IN EXCLUSIVE MODE* are planned.

mysql Tables: The *mysql* tables for managing MySQL access privileges cannot be transformed into InnoDB tables. They must remain in MyISAM format.

License Costs: Adding InnoDB support to a commercial MySQL license doubles the cost. (This is relevant only if you are developing a commercial product. With open source programs, *Indoor* projects, and normal websites, the free MySQL version suffices. See also Chapter 1.)

■ **Tip** Further details on the limitations of InnoDB tables in relation to MyISAM tables can be found at the following address: `http://dev.mysql.com/doc/mysql/en/innodb-restrictions.html`.

MyISAM or InnoDB?

You can specify individually for each table in your database which table driver is to be used. That is, it is permitted within a single database to use both MyISAM and InnoDB tables. This allows you to choose the optimal table driver for each table, depending on its content and the application that will be accessing it.

MyISAM tables are to be recommended whenever you want to manage tables in the most space- and time-efficient way possible. InnoDB tables, on the other hand, take precedence when your application makes use of transactions, requires greater security, or is to be accessed by many users simultaneously for making changes.

There is no generally valid answer to the question of which table type offers faster response. In principle, since transactions take time and InnoDB tables take up more space on the hard drive, MyISAM should have the advantage. But with InnoDB tables you can avoid the use of *LOCK TABLE* commands, which offers an advantage to InnoDB, which is better optimized for certain applications.

Furthermore, the speed of an application depends heavily on hardware (particularly the amount of RAM), the settings in the MySQL configuration file, and other factors. Therefore, I can provide here only the following advice: In speed-critical applications, perform your own tests on both types of tables.

HEAP Tables

HEAP tables exist only in RAM (not on the hard drive). They use a *hash index*, which results in particularly fast access to individual data records. HEAP tables are often used as temporary tables. See the next section for more about this topic.

In comparison to normal tables, HEAP tables present a large number of functional restrictions, of which we mention here only the most important: No *xxxTEXT* or *xxxBLOB* data types can be used. Records can be searched only with = or <=> (and not with <, >, <=, or >=). *AUTO_INCREMENT* is not supported. Indexes can be set up only for *NOT NULL* columns.

HEAP tables should be used whenever relatively small data sets are to be managed with maximal speed. Since HEAP tables are stored exclusively in RAM, they disappear as soon as MySQL is terminated. The maximum size of a HEAP table is determined in the MySQL configuration file by the parameter `max_heap_table_size`.

Temporary Tables

With all of the table types listed above there exists the possibility of creating a table on a temporary basis. Such tables are automatically deleted as soon as the link with MySQL is terminated. Furthermore, temporary tables are invisible to other MySQL links (so that it is possible for two users to employ temporary tables with the same name without running into trouble).

Temporary tables are not a separate table type unto themselves, but rather a variant of the types that we have been describing. Temporary tables are often created automatically by MySQL in order to assist in the execution of *SELECT* queries.

Temporary tables are not stored in the same directory as the other MySQL tables, but in a special temporary directory (under Windows it is usually called `C:\Windows\Temp`, while under Unix it is generally `/tmp` or `/var/tmp` or `/usr/tmp`). The directory can be set at MySQL startup.

Other Table Types

MySQL recognizes a variety of other table types, of which those listed here are the most important variants. Note that these table types are available only in the Max version or self-compiled version of MySQL. You can determine which table types your MySQL version supports with the command *SHOW ENGINES*.

BDB Tables: BDB tables were historically the first transactions-capable MySQL table type. Now that the InnoDB table driver has matured, there is not much reason to use BDB tables.

Compressed Tables (type ARCHIVE, since MySQL 4.1): This table type is designed for the archiving and logging of large data sets. The advantage of this table type is that the records are immediately compressed when they are stored.

However, *ARCHIVE* tables make sense only if the records are not to be altered. (*INSERT* is permitted, but *UPDATE* and *DELETE* are not allowed.)

ARCHIVE tables cannot be indexed. For each *SELECT* command, therefore, all records must be read! So use this table type only if you expect to access the data relatively rarely.

Tables in Text Format (type CSV, since MySQL 4.1): Records from CSV tables are saved as text files with comma-separated values. For example, *"123","I am a character string"*. CSV tables cannot be indexed.

NDB Tables (MySQL Cluster, since MySQL 4.1): The NDB table type belongs with the MySQL cluster functions, which are integrated into the MySQL Max version. (NDB stands for *network database.*) This table type is transactions-capable and is most suitable for databases that are distributed among a large number of computers. However, the use of this table type requires that first a number of MySQL Max installations be specially configured for cluster operation. Detailed information can be found at `http://dev.mysql.com/doc/mysql/en/ndbcluster.html`.

External Tables (type FEDERATED, since MySQL 5.0): This table type enables access to tables in an external database. The database system can be located, for example, on another computer in the local network. At present, the external database must be a MySQL database, though perhaps in the future, MySQL will allow connection with other database systems.

There are some restrictions in accessing *FEDERATED* tables: Neither transactions nor query optimization with Query Cache are possible. The structure of external tables cannot be changed (though the records can be). In other words, *ALTER TABLE* is not permitted, while *INSERT*, *UPDATE*, and *DELETE* are.

Further information on all the MySQL table types can be found as a separate chapter in the MySQL documentation at `http://dev.mysql.com/doc/mysql/en/storage-engines.html`.

Table Files

You can specify the location for database files at MySQL startup. (Under Unix/Linux `/var/lib/mysql` is frequently used, while under Windows it is usually `C:\Programs\MySQL\MySQL Server n.n\data`.) All further specifications are relative to this directory.

A description of each table is saved in a `*.frm` file. The `*.frm` files are located in directories whose names correspond to the name of the database: `data/dbname/tablename.frm`. This file contains the table schema (data types of the columns, etc.).

Beginning with MySQL 4.1, an additional file, `db.opt`, is stored in the database directory, which relates to the entire database: `data/dbname/db.opt`. This file contains the database settings.

For each MyISAM table, two additional files are created: `data/dbname/tablename.MYD`, with MyISAM table data, and `data/dbname/tablename.MYI`, with MyISAM indexes (all indexes of the table).

InnoDB tables are stored in individual files for each table or else collectively, in the so-called *tablespace* (depending on whether `innodb_file_per_table` is specified in the MySQL configuration file). The location and name of the tablespace are also governed by configuration settings. In current MySQL installations, the default is `data/dbname/tablename.ibd` for InnoDB table data (data and indexes), and `data/ibdata1, -2, -3` for the tablespace and undo logs, and `data/ib_logfile0, -1, -2` for InnoDB logging data.

If triggers are defined for the tables (see Chapter 13), then their code is currently stored in a file `data/dbname/tablename.TRG`, though it is possible that this will change in future versions of MySQL.

MySQL Data Types

Every table is composed of a number of columns. For each column, the desired data type may be specified. This section provides an overview of the data types available in MySQL.

Integers (xxxINT)

With the *INT* data type, both positive and negative numbers are generally allowed. With the attribute *UNSIGNED*, the range can be restricted to the positive integers. But note that then subtraction returns *UNSIGNED* integers, which can lead to deceptive and confusing results.

With *TINYINT*, numbers between -128 and +127 are allowed. With the attribute *UNSIGNED*, the range is 0 to 255. If one attempts to store a value above or below the given range, MySQL simply replaces the input with the largest or, respectively, smallest permissible value. See Table 8-1.

Table 8-1. *Data Types for Integers*

MySQL Data Type	Meaning
TINYINT(m)	8-bit integer (1 byte, -128 to +127); the optional value *m* gives the desired column width in *SELECT* results (*Maximum Display Width*), but has no influence on the permitted range of numbers.
SMALLINT(m)	16-bit integer (2 bytes, -32,768 to + 32,767)
MEDIUMINT(m)	24-bit integer (3 bytes, -8,388,608 to +8,388,607)
INT(m), INTEGER(m)	32-bit integer (4 bytes, -2.147,483,648 to +2,147,483,647)
BIGINT(m)	64-bit integer (8 bytes, $\pm 9.22*10^{18}$)
SERIAL	Synonym for *BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY*

Optionally, in the definition of an integer field, the desired column width (number of digits) can be specified, such as, for example, *INT(4)*. This parameter is called *M* (for *Maximum Display Size*) in the literature. It assists MySQL as well as various user interfaces in presenting query results in a readable format.

Note Note that with the *INT* data types, the *M* restricts neither the allowable range of numbers nor the possible number of digits. In spite of setting *INT(4)*, for example, you can still store numbers greater than 9999. However, in certain rare cases (such as in complex queries for the evaluation of which MySQL constructs a temporary table), the numerical values in the temporary tables can be truncated, with incorrect results as a consequence.

AUTO_INCREMENT Integers

With the optional attribute *AUTO_INCREMENT* you can achieve for integers that MySQL automatically inserts a number that is 1 larger than the currently largest value in the column when a new record is created for the field in question. *AUTO_INCREMENT* is generally used in the definition of fields that are to serve as the primary key for a table.

The following rules hold for *AUTO_INCREMENT*:

- This attribute is permitted only when one of the attributes *NOT NULL*, *PRIMARY KEY*, or *UNIQUE* is used as well.
- It is not permitted for a table to possess more than one *AUTO_INCREMENT* column.
- The automatic generation of an ID value functions only when in inserting a new data record with *INSERT*, a specific value or *NULL* is not specified. However, it is possible to generate a new data record with a specific ID value, provided that the value in question is not already in use.
- If you want to find out the *AUTO_INCREMENT* value that a newly inserted data record has received, after executing the *INSERT* command (but within the same connection or transaction), execute the command *SELECT LAST_INSERT_ID()*.
- If the *AUTO_INCREMENT* counter reaches its maximal value, based on the selected integer format, it will not be increased further. No more insert operations are possible. With tables that experience many *INSERT* and *DELETE* commands, it can happen that the 32-bit *INT* range will become used up, even though there are many fewer than two billion records in the table. In such a case, use a *BIGINT* column.

Binary Data (BIT and BOOL)

The keyword *BOOL* in MySQL is a synonym for *TINYINT*. This was also true prior to version 5.0.2 for *BIT*. Beginning with version 5.0.3, *BIT* is an independent data type for storing binary values with up to 64 bits.

Floating-Point Numbers (FLOAT and DOUBLE)

Since version 3.23 of MySQL, the types *FLOAT* and *DOUBLE* correspond to the IEEE numerical types for single and double precision that are available in many programming languages.

Optionally, the number of digits in *FLOAT* and *DOUBLE* values can be set with the two parameters *m* and *d*. In that case, *m* specifies the number of digits before the decimal point, while *d* gives the number of places after the decimal point. The floating-point data types are summarized in Table 8-2.

Table 8-2. *Data Types for Floating-Point Numbers*

Data Type	Meaning
FLOAT(m, d)	Floating-point number, 8-place accuracy (4 byte); the optional values *m* and *d* give the desired number of places before and after the decimal point in *SELECT* results; these values have no influence on the actual way the numbers are stored.
DOUBLE(m, d)	Floating-point number, 16-place accuracy (8 byte).
REAL(m, d)	Synonym for *DOUBLE*.

The parameter *m* does no more than assist in the formatting of numbers; it does not limit the permissible range of numbers. On the other hand, *d* has the effect of rounding numbers when they are stored. For example, if you attempt to save the number 123456.789877 in a field with the attribute *DOUBLE(6,3)*, the number stored will, in fact, be 123456.790.

■ **Note** MySQL expects floating-point numbers in international notation, that is, with a decimal point, and not a comma (which is used in a number of European countries). Results of queries are always returned in this notation, and very large or very small values are expressed in scientific notation (e.g., 1.2345678901279e+017).

If you have your heart set on formatting floating-point numbers differently, you will have either to employ the function *FORMAT* in your SQL queries (though this function is of use only in the thousands groupings) or to carry out your formatting in the client programming language (that is, in PHP, Perl, etc.).

Fixed-Point Numbers (DECIMAL)

The integer type *DECIMAL* is recommended when rounding errors caused by the internal representation of numbers as *FLOAT* or *DOUBLE* are unacceptable, perhaps with currency values. Since the numbers are stored as character strings, the storage requirement is much greater. At the same time, the possible range of values is smaller, since exponential notation is ruled out. The fixed-point data types are summarized in Table 8-3.

Table 8-3. *Data Types for Fixed-Point Numbers*

Data Type	Meaning
DECIMAL(p, s)	Fixed-point number, saved as a character string; arbitrary number of digits (1 byte per digit + 2 bytes overhead)
NUMERIC, DEC	Synonym for *DECIMAL*

The two parameters *p* and *s* specify the total number of digits (*precision*) and, respectively, the number of digits after the decimal point (*scale*). The range in the case of *DECIMAL(6,3)* is from 9999.999 to -999.999. This bizarre range results from the apparent fact that six places are reserved for the number plus an additional place for the minus sign. When the number is positive, the place for the minus sign can be commandeered to store another digit. If *p* and *s* are not specified, then MySQL automatically uses (10, 0), with the result that positive integers with eleven digits and negative integers with ten digits can be stored.

Date and Time (DATE, TIME, DATETIME, TIMESTAMP)

Table 8-4 summarizes the data types for storing time values.

Table 8-4. *Data Types for Date and Time*

MySQL Keyword	Meaning
DATE	Date in the form '2003-12-31', range 1000-01-01 to 9999-12-31 (3 bytes)
TIME	Time in the form '23:59:59', range ±838:59:59 (3 bytes)
DATETIME	Combination of *DATE* and *TIME* in the form '2003-12-31 23:59:59' (8 bytes)
YEAR	Year 1900–2155 (1 byte)

Data Validation

In older versions of MySQL, the *DATE* and *DATETIME* data types did only a limited amount of type checking. Values between 0 and 12 for months, and between 0 and 31 for days, were generally allowed. However, it is the responsibility of the client program to provide correct data. (For example, 0 is a permissible value for a month or day, in order to provide the possibility of storing incomplete or unknown data.)

Beginning with MySQL 5.0.2 there is a more thorough validation, so that only valid data can be stored. Still allowed are the month and day values 0, as well as the date 0000-00-00.

Validation can be controlled through the MySQL system variable *sql_mode* (see also Chapter 14). Table 8-5 summarizes the *sql_mode* values that are relevant for time validation.

Table 8-5. *sql_mode Settings*

Setting	Meaning
ALLOW_INVALID_DATES	Obviously incorrect dates (e.g., 2005-02-31) are accepted.
NO_ZERO_DATE	0000-00-00 will no longer be accepted as a valid date.
NO_ZERO_IN_DATE	0 will not be accepted as a valid month or day.

Peculiarities of TIMESTAMP

Among the data types for date and time, *TIMESTAMP* plays a particular role. Fields of this type are automatically updated whenever the record is altered, thereby reflecting the time of the last change. Fields of type *TIMESTAMP* are therefore usually employed only for internal management, not for the storage of "real" data, though such is possible.

Many database operations with particular client libraries (such as with Connector/ODBC) work correctly only if every table in the database has a *TIMESTAMP* column. The time of the last change is often needed for internal administration of the data.

For the automatic *TIMESTAMP* updating to function properly, the column involved must have either no explicit value assigned or else *NULL*. In either case, MySQL itself inserts the current time.

If there is more than one *TIMESTAMP* column in a table, then the first column is updated for which a constant time value is not explicitly defined (*DEFAULT 0*).

Since MySQL 4.1.3 it has been possible to control the behavior of *TIMESTAMP* columns more precisely through two attributes. Possible combinations of attributes are shown in Table 8-6.

Table 8-6. *TIMESTAMP Variants*

Setting	Meaning
TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	The column is automatically updated both when a new record is created and for each change in a record.
TIMESTAMP	As above, but with less typing.
TIMESTAMP DEFAULT CURRENT_TIMESTAMP	The column is initialized when a new record is created, but is left unchanged thereafter.
TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	The column is initialized to zero when it is created. The current time is stored only on subsequent changes.
TIMESTAMP DEFAULT 'yyyy-mm-dd hh:mm:ss' ON UPDATE CURRENT_TIMESTAMP	The column is initialized on creation to the given time; the current time is stored only when changes are subsequently made.

■ **Caution** Do not use a *TIMESTAMP* column if you wish to store the date and time yourself. For that, one has the data type *DATETIME*.

If you wish to deviate from the default behavior and not have the *TIMESTAMP* column change when a record is changed, then you must specify the date and time explicitly:

```
UPDATE tablename SET col='new value, ts=ts;
```

MySQL versions through 4.0 formatted *TIMESTAMP* values in the form *YYYYMMDDHHMMSS* (instead of *YYYY-MM-DD HH:MM:SS*). This can lead to eventual incompatibilities when data are later processed. Add a zero if you prefer the old format (that is, *SELECT ts+0 FROM table*).

Microseconds: In future versions of MySQL it is possible that TIMESTAMP columns will also store microseconds. The syntax for such times has already been set (*2005-31-12 23:59:59.nnnnnn*), and there exist various *MICROSECOND* functions for processing such data. However, actually storing the data is not yet possible (this includes MySQL 5.0.2).

Functions for Processing and Formatting Dates and Times

MySQL returns dates in the form 2005-12-31. However, with *INSERT* and *UPDATE* it manages to deal with other formats, provided that the order year/month/day is adhered to and the values are numeric. If the year is given as a two-digit number, then the following interpretation is made: 70–99 becomes 1970–1999, while 00–69 becomes 2000–2069.

If query results are to be specially formatted, there are several MySQL functions available for processing date and time values. The most flexible of these is *DATE_FORMAT*, whose application is demonstrated in the following example:

```
SELECT DATE_FORMAT(birthdate, '%Y %M %e') FROM students
1977 September 3
1981 October 25
...
```

Tip *DATE_FORMAT*, as well as many other MySQL functions for processing dates and times, are introduced in Chapter 10. There you will also learn how to convert times among different time zones. A reference for all time and date functions is in Chapter 21.

Character Strings (CHAR, VARCHAR, xxxTEXT)

Table 8-7 summarizes the data types for storing character strings.

Table 8-7. *Data Types for Character Strings*

MySQL Keyword	Meaning
CHAR(n)	character string with specified length, maximum 255 characters
VARCHAR(n)	character string with variable length, maximum 255 characters (MySQL through 4.1: *n*<256; MySQL from 5.0.3: *n*<65,535)
TINYTEXT	character string with variable length, maximum 255 bytes *TEXT* character string with variable length, maximum 2^{16}-1 characters
MEDIUMTEXT	character string with variable length, maximum 2^{24}-1 characters
LONGTEXT	character string with variable length, maximum 2^{32}-1 characters

With *CHAR*, the length of a character string is strictly specified. For example, *CHAR(20)* demands 20 bytes in each record, regardless of the length of the character string actually stored. (Blank characters at the beginning of a character string are eliminated before storage. Short character strings are extended with blanks. These blank characters are automatically deleted when the data are read out, with the result that it is impossible to store a character string that actually has blank characters at the end.)

In contrast, the length of a character string of type *VARCHAR* or one of the four *TEXT* types is variable. The storage requirement depends on the actual length of the character string.

Although *VARCHAR* and *TINYTEXT*, both of which can accept a character string up to a length of 65,535 characters, at first glance seem equivalent, there are, in fact, several features that distinguish one from the other: The maximum number of characters in *VARCHAR* columns must be specified (in the range 0 to 65,535) when the table is declared. Character strings that are too long will be unceremoniously, without warning, truncated when they are stored. In contrast, with *xxxTEXT* columns one cannot specify a maximal length. (The only limit is the maximal length of the particular text type.)

Caution When it creates a new table, MySQL frequently changes the column definition into a form that is more efficient for MySQL (see also Chapter 9). These automatic changes are described in the MySQL documentation as *silent column changes*. They affect both *CHAR* and *VARCHAR* columns:

VARCHAR(n) with *n*<4 is always changed into *CHAR(n)*.

CHAR(n) with *n*>3 is changed into *VARCHAR(n)* if there are additional *VARCHAR*, *TEXT*, or *BLOB* columns in the table. If the table contains only columns with constant length, then *CHAR(n)* remains unchanged.

New Aspects of VARCHAR: In MySQL 5.0 there are two significant innovations for the data type *VARCHAR*. (In the tested version 5.0.3, the new *VARCHAR* implementation was available only for MyISAM tables, and not for InnoDB tables.)

- The maximal column size for MyISAM tables is now 65,535 bytes (it was previously 255 bytes). The maximum number of characters depends on the character set, since many character sets require more than 1 byte per character.
- Spaces at the beginning and end of *VARCHAR* values are now stored in the table. Thus *INSERT INTO table (varcharcolumn) VALUES (' abc ')* actually stores *' abc '* in the column, that is, a space, the characters *a*, *b*, and *c*, and finally another space character. (Previously, MySQL deleted spaces at the end of *VARCHAR* values, which was in violation of the ANSI standard.)

Binary Attribute: Columns of type *CHAR* and *VARCHAR* can optionally be given the attribute *BINARY*. They then behave essentially like *BLOB* columns (see below). The attribute *BINARY* can also be useful when you store text: What you achieve is that in sorting, it is exclusively the binary code of the characters that is considered (and not a particular sorting table). Thus case distinction is made, which otherwise would not be the case. This makes the internal management of binary character strings simpler and therefore faster than is the case with garden-variety character strings.

Character Set Fundamentals

With all text columns you can use the additional attributes *CHARACTER SET charactersetname COLLATE sortorder* to specify the desired character set and sort order. Character sets determine what code is used to represent the various characters. Most character sets agree on the 128 English ASCII characters (e.g., code 65 for the letter A). More problematic is the representation of international characters.

Latin Character Sets: In the past, each linguistic region developed various one-byte character sets, of which the *Latin* character sets have achieved the most widespread use: *Latin1*, alias ISO-8859-1, contains all characters usual in Western Europe (äöüßáàå etc.). *Latin2*, alias ISO-8859-2, contains characters from Central and East European languages. *Latin0*, alias Latin9, alias ISO-8859-15, is the same as Latin 1, but with the euro symbol included.

The problem with these character sets is obvious: None of these character sets contains all the characters of all the European languages, so there is no Latin character set for all of Europe.

Unicode Variants: To solve this problem, the 2-byte Unicode character set was developed. With 65,535 possible characters, it covers not only all of Europe, but most of the Asian languages as well.

However, Unicode determines only which code is associated with which character, not how the codes are internally stored. For this there are several variants, of which UCS-2 (universal character set) and UTF-8 (Unicode transfer format) are the most important.

- UCS-2, alias UTF-16, represents what is apparently the simplest solution, to represent each character by 2 bytes (thus 16 bits). This format is called UTF-16 or UCS-2. Almost all operating system functions of Microsoft Windows use this representation.

  However, this representation has two drawbacks: First, the storage requirement for characters is automatically doubled, even in those cases in which only European or even only the English ASCII characters are to be stored. Second, the byte code 0 appears in many places in Unicode character strings. Thus in texts with English ASCII characters, every second byte is zero. Many C programs, email servers, and the like assume that a zero byte signals the end of a character string.

- UFT-8 is the most popular alternative to UTF-16. In this case, all the ASCII characters (7 bit) are represented as before by a single byte whose first bit is 0. All other Unicode characters are represented by strings of 2 to 4 bytes.

  The clear disadvantage of this format is that there is no obvious relationship between the number of bytes and the number of characters in a document. Because of its greater compatibility to existing programs and a number of other advantages, UTF-8 has established itself as the standard under Unix/Linux and most other components important for Web development. When Unicode is mentioned in this book, it will always mean the UTF-8 format.

Despite the clear advantages of Unicode, in any of its formats, there are two reasons not to jump on the bandwagon immediately: First, the Unicode character set is incompatible with the well-known 1-byte character sets. Second, Unicode support for components used in Web development are anything but perfect. (The weakest link in the chain is PHP, and the situation will likely not improve before version 5.2.)

MySQL Character Set Support

Through version 4.0 of MySQL, the character set and sort order for all text fields were specified by the MySQL server. The *Latin1* character set was the default, together with the Swedish sort order. Another character set and sort order could be specified in the MySQL configuration file, but this setting then held for all databases. Furthermore, any change required restarting the database server and re-creation of all indexes. Unicode was not supported at all.

Beginning with MySQL 4.1, the situation has greatly improved: Now the character set and sort order can be specified individually for every column. At the same time, the selection of character sets and sort orders is larger than ever, including Unicode UTF-8 and UCS-2.

A long list of all the character sets as well as the associated possible sort orders can be displayed with the SQL command *SHOW COLLATION*, and the result is displayed here in abbreviated form. (Note that character sets and sort orders cannot be randomly mixed. Each character set offers a particular selection of sort orders. The default sort order for each character set is indicated by *Yes* in the column *Default*.)

```
SHOW COLLATION
Collation             Charset   Id  Default  Compiled  Sortlen
ascii_bin             ascii     65                           0
ascii_general_ci      ascii     11  Yes                      0
binary                binary    63  Yes      Yes             1
latin1_bin            latin1    47           Yes             1
latin1_danish_ci      latin1    15                           0
latin1_general_ci     latin1    48                           0
latin1_general_cs     latin1    49                           0
latin1_german1_ci     latin1     5                           0
latin1_german2_ci     latin1    31           Yes             2
latin1_spanish_ci     latin1    94                           0
latin1_swedish_ci     latin1     8  Yes      Yes             1
latin2_bin            latin2    77                           0
latin2_croatian_ci    latin2    27                           0
latin2_general_ci     latin2     9  Yes                      0
latin2_hungarian_ci   latin2    21                           0
latin5_bin            latin5    78                           0
latin5_turkish_ci     latin5    30  Yes                      0
latin7_bin            latin7    79                           0
latin7_estonian_cs    latin7    20                           0
latin7_general_ci     latin7    41  Yes                      0
```

```
latin7_general_cs     latin7     42                     0
utf8_bin              utf8       83           Yes       1
utf8_czech_ci         utf8      202           Yes       8
utf8_danish_ci        utf8      203           Yes       8
utf8_estonian_ci      utf8      198           Yes       8
utf8_general_ci       utf8       33  Yes      Yes       1
utf8_icelandic_ci     utf8      193           Yes       8
utf8_latvian_ci       utf8      194           Yes       8
utf8_lithuanian_ci    utf8      204           Yes       8
utf8_persian_ci       utf8      208           Yes       8
utf8_polish_ci        utf8      197           Yes       8
utf8_roman_ci         utf8      207           Yes       8
utf8_romanian_ci      utf8      195           Yes       8
utf8_slovak_ci        utf8      205           Yes       8
utf8_slovenian_ci     utf8      196           Yes       8
utf8_spanish2_ci      utf8      206           Yes       8
utf8_spanish_ci       utf8      199           Yes       8
utf8_swedish_ci       utf8      200           Yes       8
utf8_turkish_ci       utf8      201           Yes       8
utf8_unicode_ci       utf8      192           Yes       8
...
```

Table 8-8 gives additional information about the most important combinations of character set and sort order.

Table 8-8. *The Most Important Character Sets and Sort Orders*

Character Set	Sort Order	Meaning
latin1	*latin1_swedish_ci*	Swedish sort order, valid in MySQL by default for *latin1* columns; *ci* stands for case-insensitive.
latin1	*latin1_general_ci*	General sort order, suitable for many Western European languages without taking country-specific issues into account.
latin1	*latin1_general_cs*	As above, but case-sensitive: uppercase letters are sorted before lowercase letters.
latin1	*latin1_german1_ci*	German sort order according to the DIN-1 standard (ä=a, ö=o, ü=u, ß=s).
latin1	*latin1_german2_ci*	German sort order according to the DIN-2 standard (telephone directory rules, thus ä = ae, ö = oe, ü = ue, ß = ss).
utf8	*utf8_general_ci*	General sort order, suitable for many Western European languages without taking country-specific issues into account; is the default in MySQL for *utf8* columns.

If you do not specify a character set and sort order for a column, then the default character set for the table, the database, or the MySQL server is used, depending on the level at which the default settings are defined.

Which character set is optimal for your database depends, of course, on the application. If there will be no need to use characters outside of the Western European languages, then you can stick with *latin1*. This character set generally poses no problems for further data processing.

Tip There is much more in this book on the subject of character sets and sort orders. See the topics SQL syntax, SQL commands, MySQL variables, and MySQL configuration. Furthermore, all the chapters on programming discuss issues surrounding the processing of Unicode character strings. Look in the index under *Unicode*.

Binary Data (xxxBLOB and BIT)

For the storage of binary data there are four *BLOB* data types at your service, all of which display almost the same properties as the *TEXT* data types. (Recall that "BLOB" is an acronym for "binary large object.") The only difference is that text data are usually compared and sorted in text mode (case-insensitive), while binary data are sorted and compared according to their binary codes.

There is considerable disagreement as to whether large binary objects should even be stored in a database. The alternative would be to store the data (images, for example) in external files and provide links to these files in the database.

The advantage to using BLOBs is the resulting integration into the database (more security, simpler backups, unified access to all data). The drawback is the usually significant slowdown. It is particularly disadvantageous that large and small data elements—strings, integers, etc.—on the one hand and BLOBs and long texts on the other must be stored all mixed together in a table file. The result is a slowdown in access to all of the data records.

Note as well that BLOBs in general can be read only as a whole. That is, it is impossible to read, say, the last 100 kilobytes of an 800-kilobyte BLOB. The entire BLOB must be transmitted.

Table 8-9 summarizes the data types for binary data.

Table 8-9. *Data Types for Binary Data*

MySQL Keyword	Meaning
BIT(n)	Bit data, where *n* is the number of bits (up to 64)
TINYBLOB	Binary data with variable length, maximum 255 bytes
BLOB	Binary data with variable length, maximum 2^{16}-1 bytes
MEDIUMBLOB	Binary data with variable length, maximum 2^{24}-1 bytes
LONGBLOB	Binary data with variable length, maximum 2^{32}-1 bytes

BIT Data

New since MySQL 5.0.3 is the possibility of defining columns of width up to 64 bits. To be sure, the data type *BIT* existed previously, but in earlier versions of MySQL, *BIT* was a synonym for *TINYINT(1)*.

There is a new syntax *b'0101'* for writing bit values. *SELECT* queries return binary values for *BIT* columns. However, the tested client programs were incapable of displaying these values correctly. If necessary, use *SELECT bitcolumn+0* to convert binary values to integers, and use *SELECT BIN(bitcolumn+0)* to display these integers in binary notation.

If you insert numbers into *BIT* columns causing underflow or overflow, all the bits will be set to 1. If, for example, you were to insert the numbers -1, 0, 1, 7, 8 into a *BIT(3)* column, the following binary values will be stored: *b'111', b'000', b'001', b'111', b'111'.*

Other Data Types

The two data types *ENUM* and *SET* are a peculiarity of MySQL. They enable a particularly efficient management of character set enumerations and combinations for the MySQL server.

With *ENUM* you can manage a list of up to 65,535 character strings assigned consecutive numbers. Then one of these strings can be selected in a field.

SET follows a similar idea, but here arbitrary combinations are possible. Internally, the strings are associated with powers of 2 (1, 2, 4, 8, etc.), so that a bitwise combination is possible. Accordingly, the memory requirement is greater than with *ENUM*s (1 bit per string). At most 64 strings can be thus combined. (The memory requirement is then 8 bytes.)

These data types have some drawbacks, however. First, managing them with PHP is relatively complex (for example, if you wish to determine the strings available in an *ENUM* field). Second, most other database systems know nothing about *ENUM* and *SET*, which could complicate a future transition to another database system. Therefore, it is often better to use additional linked tables in place of *ENUM*s and *SET*s.

Table 8-10 summarizes the additional data types.

Table 8-10. *Additional Data Types*

MySQL Keyword	Meaning
ENUM	Enumeration of up to 65,535 strings (1 or 2 bytes; see Chapter 10)
SET	Combination of up to 255 strings (1 to 8 bytes; see Chapter 10)
GEOMETRY, *POINT*, etc.	Geometric object (since MySQL 4.1; see Chapter 12)

Options and Attributes

A variety of options and additional attributes can be specified when a column is created. Table 8-11 lists the most important options. Note that many attributes are suitable only for particular data types.

Table 8-11. *Important Column Attributes and Options*

MySQL Keyword	Meaning
NULL	The column may contain *NULL* values. (This setting holds by default.)
NOT NULL	The value *NULL* is not permitted.
DEFAULT xxx	The default value *xxx* will be used if no other value is specified on input.
DEFAULT CURRENT_TIMESTAMP	For *TIMESTAMP* columns, the current time is stored when new records are input.
ON UPDATE CURRENT_TIMESTAMP	For *TIMESTAMP* columns, the current time is stored when changes are made (*UPDATE*).
PRIMARY KEY	Defines the column as a primary key.
AUTO_INCREMENT	A sequential number is automatically input. *AUTO_INCREMENT* can be used only for a column with integer values. Moreover, the options *NOT NULL* and *PRIMARY KEY* must be given. (Instead of *PRIMARY KEY*, the column can be given a *UNIQUE* index.)
UNSIGNED	Integers are stored without a sign. Warning: calculations are then also made without a sign.
CHARACTER SET name [COLLATE sort]	For strings, specifies the character set and optionally the desired sort order.

Unfortunately, MySQL does not allow a function to be given as default value. It is also impossible, for example, to specify *DEFAULT RAND()* if you wish to have a random number automatically stored in a column. It is also impossible to define validation rules for columns (so that, for example, only values between 0 and 100 can be stored).

Tips and Tricks on Database Design

Rules for Good Database Design

- Tables should not contain redundant (repetitive) data. (If you are repeatedly entering the same numbers or strings in a table, something is wrong.)
- Tables should not have columns like *order1*, *order2*, *order3*. Even if you allow for 10 such columns, the day will come when a customer orders 11 articles.
- The storage requirements for all your tables should be as small as possible.
- Frequently required database queries should be able to be executed simply and efficiently. (One usually notices a violation of this rule only when the database contains not a couple of test records, but thousands or even millions. At that point, a change in the design may no longer be possible.)

These rules have the same import as the normalization rules presented in the next section, but these here are often easier to follow.

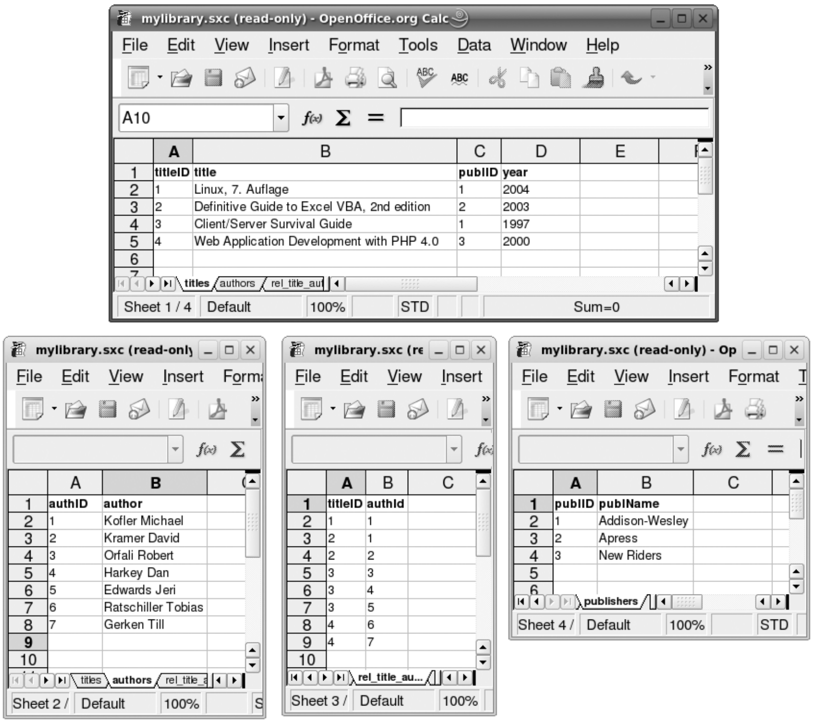
Tips for Naming

- MySQL is case-sensitive in regard to database and table names, but not so with column names. It is thus important, at least with your databases and tables, to arrive at a uniform system for using upper- and lowercase letters. (In the example databases in this book, only lowercase letters are generally used for naming databases and tables.)
- Names of databases, tables, and columns can be at most 64 characters long.
- Avoid special characters (e.g., üàû) in names. MySQL allows all alphanumeric characters, but different operating systems and Linux distributions use different default character sets, and changing a system could lead to problems.
- Choose clear field and table names. Take care in naming fields that they describe the content accurately. Thus *authName* is better than *name*.
- A uniform naming system for fields can save many careless errors. Whether you prefer *author_name* or *authName* is irrelevant, as long as you are consistent.
- Similarly, you should consider how you want to deal with singular and plural. In my tables, I have tried to use plural exclusively. There is no rule as to what is correct, but it is confusing if half your tables use singular, the other half plural.

Tips on the Design Process

It is no easy matter to distribute a collection of data efficiently and intelligently among several tables. Novices in the area of database design should take the following suggestions to heart:

- Begin with a relatively small number of test data, and attempt to enter them in one or more tables. (The scope of the test data should not, however, be so small that obvious design problems go undetected. But it should not be so large that the time taken up in design is too great.)
- Perform your first experiments not with real MySQL tables, but instead, with some worksheets in a table calculation program such as Excel or OpenOffice Calc. (Figure 8-1 gives a preview of the next section.) This allows you to work in a far less complex environment. At this point, you should be focusing on the distribution of data among the tables and their columns, not on database-specific details such as column format and indexes.


mylibrary.sxc (read-only) - OpenOffice.org Calc
File Edit View Insert Format Tools Data Window Help
A10
titles authors rel_title_aut
Sheet 1 / 4 Default 100% STD Sum=0
Sheet 2 / Default 100%
Sheet 3 / Default 100%
publishers
Sheet 4 / Default 100% STD

titleID	title	publID	year
1	Linux, 7. Auflage	1	2004
2	Definitive Guide to Excel VBA, 2nd edition	2	2003
3	Client/Server Survival Guide	1	1997
4	Web Application Development with PHP 4.0	3	2000

authID	author
1	Kofler Michael
2	Kramer David
3	Orfali Robert
4	Harkey Dan
5	Edwards Jeri
6	Ratschiller Tobias
7	Gerken Till

titleID	authId
1	1
2	1
2	2
3	3
3	4
3	5
4	6
4	7

publID	publName
1	Addison-Wesley
2	Apress
3	New Riders

Figure 8-1. *Database design with a table calculation program*

Normalization Rules

Why is it that authors of books think about nothing (well, almost nothing) but books? The author begs the indulgence of his readers in that the example in this section deals with books. The goal of the section is to create a small database in which data about books can be stored: book title, publisher, author(s), publication date, and so on. These data can of course, be stored without a database, in a simple list in text format, for example, as appears in the Bibliography at the end of this book:

Michael Kofler: *Linux*, 7th edition. Addison-Wesley 2004.

Michael Kofler, David Kramer: *Definitive Guide to Excel VBA*, Second Edition. Apress 2003.

Robert Orfali, Dan Harkey, Jeri Edwards: *Client/Server Survival Guide*. Addison-Wesley 1997.

Tobias Ratschiller, Till Gerken: *Web Application Development with PHP 4.0*. New Riders 2000.

This is a nice and convenient list, containing all necessary information. Why bother with all the effort to transform this text (which perhaps exists as a document composed with some word-processing program) into a database?

Needless to say, the reasons are legion. Our list can be easily searched, but it is impossible to organize it in a different way, for example, to create a list of all books by author *x* or to create a new list ordered not by author, but by title.

The resulting database *mylibrary* will be improved step by step in the course of this chapter. The finished database is available for download at the Apress website. In addition, the *mylibrary* database is the basis for countless examples in this book.

A First Attempt

Just think: Nothing is easier than to turn our list into a database table. (To save space we are going to abbreviate the book titles and names of authors.)

Table 8-12 immediately shows itself to be riddled with problems. A first glance reveals that limiting the number of authors to three was an arbitrary decision. What do you do with a book that has four or five authors? Do we just keep adding columns, up to *authorN*, painfully aware that those columns will be empty for most entries?

Table 8-12. *Library Database: First Attempt*

title	publName	year	authName1	authName2	authName3
Linux	Addison-Wesley	2004	Kofler M		
Definitive Guide …	Apress	2003	Kofler M	Kramer D	
Client/Server …	Addison-Wesley	1997	Orfali R	Harkey D	Edwards E.
Web Application …	New Riders	2000	Ratschiller T	Gerken T	

The First Normal Form

Database theorists have found, I am happy to report, a solution to such problems. Simply apply to your database, one after the other, the rules for the three *normal forms*. The rules for the first normal form are as follows (though for the benefit of the reader, they have been translated from the language of database theorists into what we might frivolously call "linguistic normal form," or, more simply, plain English):

- Columns with similar content must be eliminated.
- A table must be created for each group of associated data.
- Each data record must be identifiable by means of a *primary key*.

In our example, the first rule is clearly applicable to the *authorN* columns.

The second rule seems not to be applicable here, since in our example we are dealing exclusively with data that pertain specifically to the books in the database. Thus a single table would seem to suffice. (We will see, however, that this is not, in fact, the case.)

The third rule signifies in practice that a running index must be used that uniquely identifies each row of the table. (It is not strictly necessary that an integer be used as primary key. Formally, only the uniqueness is required. For reasons of efficiency the primary key should be as small as possible, and thus an integer is generally more suitable than a character string of variable length.)

A reconfiguration of our table after application of the first and third rules might look like that depicted in Table 8-13.

Table 8-13. *Library Database: First Normal Form*

titleID	title	publName	year	authName
1	Linux	Addison-Wesley	2004	Kofler M.
2	Definitive Guide …	Apress	2003	Kofler M.
3	Definitive Guide …	Apress	2003	Kramer D.
4	Client/Server …	Addison-Wesley	1997	Orfali R.
5	Client/Server …	Addison-Wesley	1997	Harkey D.
6	Client/Server …	Addison-Wesley	1997	Edwards E.
7	Web Application …	New Riders	2000	Ratschiller T.
8	Web Application …	New Riders	2000	Gerken T.

Clearly, the problem of multiple columns for multiple authors has been eliminated. Regardless of the number of authors, they can all be stored in our table. Of course, there is no free luncheon, and the price of a meal here is rather high: The contents of the columns *title*, *publName*, and *year* are repeated for each author. There must be a better way!

Second Normal Form

Here are the rules for the second normal form:

- Whenever the contents of columns repeat themselves, this means that the table must be divided into several subtables.
- These tables must be linked by *foreign keys*.

If you are new to the lingo of the database world, then the term *foreign key* probably seems a bit, well, foreign. A better word in everyday English would probably be *cross reference*, since a foreign key refers to a line in a different (hence foreign) table. For programmers, the word *pointer* would perhaps be more to the point, while in Internet jargon the term *link* would be appropriate.

In Table 8-13, we see that data are repeated in practically every column. The culprit of this redundancy is clearly the author column. Our first attempt to give the authors their very own table can be seen in Tables 8-14 and 8-15.

Table 8-14. *titles Table: Second Normal Form*

titleID	title	publName	year
1	Linux	Addison-Wesley	2004
2	Definitive Guide	Apress	2003
3	Client/Server	Addison-Wesley	1997
4	Web Application	New Riders	2000

Table 8-15. *authors Table: Second Normal Form*

authID	titleID	authName
1	1	Kofler M.
2	2	Kofler M.
3	2	Kramer D.
4	3	Orfali R.
5	3	Harkey D.
6	3	Edwards E.
7	4	Ratschiller T.
8	4	Gerken T.

In the *authors* table, the first column, with its running *authID* values, provides the primary key. The second column takes over the task of the foreign key. It points, or refers, to rows of the *titles* table. For example, row 7 of the *authors* table indicates that *Ratschiller, T.* is an author of the book with ID *titleID=4*, that is, the book *Web Application ….*

Second Normal Form, Second Attempt

Our result could hardly be called optimal. In the *authors* table, the name *Kofler, M.* appears twice. As the number of books in this database increases, the amount of such redundancy will increase as well, whenever an author has worked on more than one book.

The only solution is to split the *authors* table again and live without the *titleID* column. The information as to which book belongs to which author must be specified in yet a third table. These three tables are shown in Tables 8-16 through 8-18.

Table 8-16. *titles Table: Second Normal Form*

titleID	title	publName	year
1	Linux	Addison-Wesley	2004
2	Definitive Guide	Apress	2003
3	Client/Server	Addison-Wesley	1997
4	Web Application	New Riders	2000

Table 8-17. *authors Table: Second Normal Form*

authID	authName
1	Kofler M.
2	Kramer D.
3	Orfali R.
4	Harkey D.
5	Edwards E.
6	Ratschiller T.
7	Gerken T.

Table 8-18. *rel_title_author Table: Second Normal Form*

titleID	authID
1	1
2	1
2	2
3	3
3	4
3	5
4	6
4	7

This is certainly the most difficult and abstract step, probably because a table of the form *rel_title_author* has no real-world content. Such a table would be completely unsuited for unautomated management. But once a computer has been provided with a suitable program, such as MySQL, it has no trouble at all processing such data. Suppose you would like to obtain a list of all authors of the book *Client/Server* .... MySQL would first look in the *titles* table to find out what *titleID* number is associated with this book. Then it would search in the *rel_title_author* table for data records containing this number. The associated *authID* numbers then lead to the names of the authors.

■Note It may have occurred to you to ask why in the *rel_title_author* table there is no *ID* column, say, *rel_title_author_ID*. Usually, such a column is omitted, since the combination of *titleID* and *authID* is already an optimal primary key. (Relational database systems permit such primary keys, those made up of several columns.)

Third Normal Form

The third normal form has a single rule, and here it is:

- Columns that are not directly related to the primary key must be eliminated (that is, transplanted into a table of their own).

In the example under consideration, the column *publisher* appears in the *titles* table. The set of publishers and the set of book titles are independent of one another and therefore should be separated. Of course, it should be noted that each title must be related to the information as to the publisher of that title, but it is not necessary that the entire name of the publisher be given. A foreign key (that is, a reference, a pointer, a link) suffices. See Tables 8-19 and 8-20.

Table 8-19. *titles Table: Third Normal Form*

titleID	title	publID	year
1	Linux	1	2001
2	Definitive Guide	2	2003
3	Client/Server	1	1997
4	Web Application	3	2000

Table 8-20. *publishers Table: Third Normal Form*

publID	publName
1	Addison-Wesley
2	Apress
3	New Riders

The *authors* and *rel_title_author* tables remain the same in the third normal form. The completed book database now consists of four tables.

If we had paid closer attention to the rules for the first normal form (associated data belong together in a table), we could, of course, have saved some of our intermediate attempts. But that would have diminished the pedagogical value of our example. In fact, in practice, it often occurs that only when test data are inserted and redundancies are noticed does it become clear how the tables need to be subdivided.

■Tip The *mylibrary* database is somewhat more complex than the previous sections would indicate. The *titles* table contains some additional fields, for example, for storing a subtitle or a comment. An additional *languages* table contains a list of all languages in which the books are written. The *langID* field of the *titles* table contains the language for each title. Finally, the *categories* table contains a hierarchical list of all book categories (e.g., *Computer books*). The *catID* field of the *titles* table supplies the category for each title.

The structure of the *categories* table is handled in the next section. A summary of all the database properties can be found at the end of the chapter.

Normalization Theory

More Theory …

The three normal forms for relational databases were first formulated by the researcher E. F. Codd. They continue to form the basis for a branch of research that is concerned with the formal description of mathematical sets in general and of relational databases in particular.

Depending on what you read, you may come across three additional normal forms, which, however, are of much less significance in practice. The normal forms and their rules are described much more precisely than we have done. However, such descriptions are so teeming with such exotica as *entities*, *attributes*, and their ilk that the connection with relational databases can easily be lost.

If you are interested in further details on this topic, then you are encouraged to look into a good book on the subject of database design (see also the suggestions at the beginning of this chapter).

Less Theory …

I have attempted to present the first three normal forms in as simple and example-oriented a way as possible, but perhaps even that was too theoretical. Actually, the normal forms are not necessarily helpful to database beginners, since the correct interpretation of the rules is often difficult. Here are some rules that will perhaps make those first steps less shaky:

- Give yourself sufficient time to develop the database. (If at a later date you have to alter the database design, when the database is already stuffed with real data and there is already client code in existence, changes will take a great deal of time and effort.)
- Avoid giving columns names with numbers, such as *name1*, *name2*, or *object1*, *object2*. There is almost certainly a better solution involving an additional table.
- Immediately supply your database with some test data. Attempt to include as many special cases as possible. If you encounter redundancies, that is, columns in which the same content appears several times, this is usually a hint that you should break your table into two (or more) new tables.
- Try to understand the ideas of relations (more on this later in the chapter).
- A good database design cannot be obtained unless you have had some experience with SQL (see also the following two chapters). Only when you know and understand the range of SQL queries can you judge the consequences of the various ways of organizing your data.
- Orient yourself using an example database (from this book or from another book on databases).

Tip A good example of normalizing a database can be found at `http://www.phpbuilder.com/columns/barry20000731.php3`.

Normal Forms: Pro and Con

Normal forms are a means to an end, nothing more and nothing less. Normal forms should be a help in the design of databases, but they cannot replace human reasoning. Furthermore, it is not always a good idea to follow the normal form thoughtlessly, that is, to eliminate every redundancy.

Con: The input of new data, say, in a form on a web page, becomes more and more complex as the number of tables among which the data are distributed increases. This is true as much for the end user (who is led from one page to another) as for the programmer.

Furthermore, for efficiency in queries, it is sometimes advantageous to allow for a bit of redundancy. Bringing together data from several tables is usually slower than reading data from a single table. This is true especially for databases that do not change much but that are frequently given complex queries to respond to. (In a special area of databases, the *data warehouses*, redundancy is often deliberately planned for in order to obtain better response times. The purpose of data warehouses is the analysis of complex data according to various criteria. However, MySQL is not a reasonable choice of database system for such tasks anyhow, and therefore, we shall not go more deeply into the particular features of this special application area.)

Pro: Redundancy is generally a waste of storage space. You may hold the opinion that in the era of 400-gigabyte hard drives this is not an issue, but a large database will inevitably become a slow database (at the latest, when the database size exceeds the amount of RAM).

As a rule, databases in normal form offer more flexible query options. (Unfortunately, one usually notices this only when a new form of data query or grouping is required, which often occurs months after the database has become operational.)

Managing Hierarchies

In the *mylibrary* database, the table *categories* helps to order the books into various categories (textbooks, children's books, etc.). What is important here is that the categories can be ordered hierarchically. From the point of view of database design, this is simple: The field *parentID* simply refers, for each category, to the parent category. (For the root category *All books*, *parentID* contains the value *NULL*. This special case must always be taken into account in management and evaluation of the *categories* table.) Table 8-21 shows the hierarchy, and Table 8-22 shows how this hierarchy can be represented in a database table.

Table 8-21. *Example Data for the categories Table*

All books
Children's books
Computer books
Databases
Object-oriented databases
Relational databases
SQL
Programming
Perl
PHP
Literature and fiction

Table 8-22. *Database Representation of the Hierarchy of Table 8-20*

catID	CatName	parentCatID
1	Computer books	11
2	Databases	1
3	Programming	1
4	Relational databases	2
5	Object-oriented databases	2

catID	CatName	parentCatID
6	PHP	3
7	Perl	3
8	SQL	2
9	Children's books	11
10	Literature and fiction	11
11	All books	*NULL*

Hierarchy Problems

Although the representation of such hierarchies looks simple and elegant at first glance, they cause many problems as well. For example, it is impossible with simple *SELECT* queries to determine all subcategories or supercategories. Therefore, generally a number of queries have to be executed in the client program to construct the hierarchy. The necessary programming techniques are described in Chapter 15 for the programming language PHP. You can also use stored procedures for evaluating hierarchies (see Chapter 13).

Tip In point of fact, this chapter is supposed to be dealing with database design and not with SQL, but these two topics cannot be cleanly separated. There is no point in creating a super database design if the capabilities of SQL do not suffice to extract the desired data from the database's tables.

If you have no experience with SQL, you should dip into the following chapter a bit. Consider the following instructions as a sort of "advanced database design."

Almost all problems with hierarchies have to do with the fact that SQL does not permit recursive queries:

- With individual queries it is impossible to find all categories lying above a given category in the hierarchy.

  Example: The root category is called *Relational databases* (*parentCatID=2*). You would like to create a list that contains *Computer books* ➤ *Databases* ➤ *Relational databases*.

  With *SELECT * FROM categories WHERE catID=2*, you indeed find *Databases*, but not *Computer books*, which lies two places up the hierarchy. For that, you must execute an additional query *SELECT * FROM categories WHERE catID=1*. Of course, this can be accomplished in a loop in the programming language of your choice (Perl, PHP, etc.), but not with a single SQL instruction.

- It is just as difficult to represent the entire table in hierarchical form (as a tree). Again, you must execute a number of queries.

- It is not possible without extra effort to search for all books in a higher category.

  Example: You would like to find all books in the category *Computer books*.

  With *SELECT * FROM titles WHERE catID=1* you find only those titles directly linked to the category *Computer books*, but not the titles in the categories *Databases*, *Relational databases*, *Object-oriented databases*, etc. The query must be the following: *SELECT * FROM titles WHERE catID IN (1, 2 …)*, where *1, 2 …* are the ID numbers of the subordinate categories. The actual problem is to determine these numbers.

- In the relatively simple representation that we have chosen, it is not possible to associate the same subcategory with two or more higher-ranking categories.

  Example: The programming language *SQL* is linked in the above hierarchy to the higher-ranking category *Databases*. It would be just as logical to have a link to *Programming*. Therefore, it would be optimal to have *SQL* appear as a subcategory of both *Databases* and *Programming*.

- There is the danger of circular references. Such references can, of course, appear only as a result of input error, but where there are human beings who input data (or who write programs), there are certain to be errors. If a circular reference is created, most database programs will find themselves in an infinite loop. The resolution of such problems can be difficult.

None of these problems is insuperable. However, hierarchies often lead to situations in which answering a relatively simple question involves executing a whole series of SQL queries, and that is a slow process. Many problems can be avoided by doing without genuine hierarchies (for example, by allowing at most a two-stage hierarchy) or if supplementary information for a simpler resolution of hierarchies is provided in additional columns or tables (see the following section).

Building the Hierarchy Tree

If you have taken to heart the section on normalization of databases, then you know that redundancy is bad. It leads to unnecessary usage of storage space, management issues when changes are made, etc.

Yet there are cases in which redundancy is quite consciously sought in order to increase the efficiency of an application.

The following lines should make clear that database design is a multifaceted subject. There are usually several ways that lead to the same goal, and each of these paths is, in fact, a compromise of one sort or another. Which compromise is best depends largely on the uses to which the database will be put: What types of queries will occur most frequently? Will data be frequently changed?

The necessity will continually arise to display the *categories* table in hierarchical representation similar to that of Table 8-21. As we have already mentioned, such processing of the data is connected either with countless SQL queries or complex client-side code.

A possible solution is provided by the two additional columns *hierNr* and *hierIndent*. The first of these gives the row number in which the record would be located in a hierarchical representation. (The assumption is that data records are sorted alphabetically by *catName* within a level of the hierarchy.) The second of these two columns determines the level of indentation. In Table 8-23 are displayed for both of these columns the values corresponding to the representation in Table 8-21.

Table 8-23. *categories Table with the hierNr Column*

catID	CatName	parentCatID	HierNr	hierIndent
1	Computer books	11	2	1
2	Databases	1	3	2
3	Programming	1	7	2
4	Relational databases	2	5	3
5	Object-oriented databases	2	4	3
6	PHP	3	9	3
7	Perl	3	8	3
8	SQL	2	6	3

catID	CatName	parentCatID	HierNr	hierIndent
9	Children's books	11	1	1
10	Literature and fiction	11	10	1
11	All books	NULL	0	0

A simple query in `mysql` proves that this arrangement makes sense. Here are a few remarks on the SQL functions used: *CONCAT* joins two character strings. *SPACE* generates the specified number of blank characters. *AS* gives the entire expression the new Alias name *category*:

```
SELECT CONCAT(SPACE(hierIndent*2), catName) AS category,
  hierNr, hierIndent
FROM categories ORDER BY hierNr
category                         hierNr   hierIndent
All books                          0        0
  Children's books                 1        1
  Computer books                   2        1
    Databases                      3        2
      Object-oriented databases    4        3
      Relational databases         5        3
      SQL                          6        3
    Programming                    7        2
      Perl                         8        3
      PHP                          9        3
  Literature and fiction          10        1
```

You may now ask how the numerical values *hierIndent* and *hierNr* actually come into existence. The following example-oriented instructions show how a new data record (the computer book category *Operating systems*) is inserted into the table:

1. The data of the higher-ranking initial record (that is, *Computer books*) are known: *catID=1, parentCatID=11, hierNr=1, hierIndent=1.*

2a. Now we search within the *Computer books* group for the first record that lies in the hierarchy immediately after the record to be newly inserted (here, this is *Programming*). All that is of interest in this record is *hierNr.*

    Here is a brief explanation of the SQL command:

    WHERE parentCat_ID=1 finds all records that are immediately below *Computer books* in the hierarchy (that is, *Databases* and *Programming*).

    catName>'Operating Systems' restricts the list to those records that occur after the new record *Operating Systems.*

    ORDER BY catname sorts the records that are found.

    LIMIT 1 reduces the result to the first record.

```
SELECT hierNr FROM categories
WHERE parentCatID=1 AND catName>'Operating Systems'
ORDER BY catName
LIMIT 1
```

    The query just given returns the result *hierNr=7.* It is thereby clear that the new data record should receive this hierarchy number. First, however, all existing records with *hierNr>=7* should have their values of *hierNr* increased by 1.

2b. It can also happen that the query returns no result, namely, when there are no entries in the higher-ranking category or when all entries come before the new one in alphabetic order. (This would be the case if you wished to insert the new computer book category *Software engineering*.)

In that case, you must search for the next record whose *hierNr* is larger than *hierNr* for the initial record and whose *hierIndent* is less than or equal to *hierIndent* of the initial record. (In this way, the beginning of the next equal- or higher-ranking group in the hierarchy is sought.)

```
SELECT hierNr FROM categories
WHERE hierNr>1 AND hierIndent<=1
ORDER BY hierNr LIMIT 1
```

This query returns the result 10 (that is, *hierNr* for the record *Literature and fiction*). The new record will get this hierarchy number. All existing records with *hierNr>=10* must have their *hierNr* increased by 1.

2c. If this query also returns no result, then the new record must be inserted at the end of the hierarchy list. The current largest *hierNr* value can easily be determined:

```
SELECT MAX(hiernr) FROM categories
```

3. To increase the *hierNr* of the existing records, the following command is executed (for case 2a):

```
UPDATE categories SET hierNr=hierNr+1 WHERE hiernr>=7
```

4. Now the new record can be inserted. For *parentCatID*, the initial record *catID* will be used. Above, *hierNr=7* was determined. Here *hierIndent* must be larger by 1 than was the case with the initial record:

```
INSERT INTO categories (catName, parentCatID, hierNr, hierIndent)
VALUES ('Operating systems', 1, 7, 2)
```

The new columns in *categories* simplify many read operations. But the insert operations are still complex. It is more complicated to alter the hierarchy after the fact. Imagine that you wish to change the name of one of the categories in such a way as to change its place in the alphabetical order. This would affect not only the record itself, but many other records as well. For large sections of the table it will be necessary to determine *hierNr*. You see, therefore, that redundancy is bad.

In summary, you will have to decide whether it is more important to optimize read operations or write operations. For this book I have left *categories* without the two additional columns *hierNr* and *hierIndent*. First of all, this has pedagogical justification: In Chapters 13 (on stored procedures) and 15 (PHP) I can demonstrate various programming techniques for processing hierarchical data with maximum efficiency. Furthermore, I have the impression that the advantages of the simpler database design would win out in the case of a real application. Even managing the huge book database at `amazon.com`, a few thousand categories would suffice. One can expect serious problems of efficiency as a rule only with much larger tables.

Searching for Lower-Ranked Categories in the categories Table

Suppose you want to search for all *Databases* titles in the *titles* table. Then it would not suffice to search for all titles with *catId=2*, since you also want to see all the titles relating to *Relational databases*, *Object-oriented databases*, and *SQL* (that is, the titles with *catID* equal to 4, 5, and 8). The totality of all these categories will be described in the following search category group.

There are two problems to be solved: First, you must determine the list of the *catID* values for the search category group. For this a series of *SELECT* queries is needed, and we shall not go into that further here. Then you must determine from the *titles* table those records whose *catID* numbers agree with the values just found.

Thus in principle, the title search can be carried out, but the path is thorny, with the necessity of several SQL queries and client-side code.

The other solution consists in introducing a new (redundant) table, in which are stored all records lying above each of the *categories* records. This table could be called *rel_cat_parent*, and it would consist of two columns: *catID* and *parentID* (see Table 8-24). We see, then, for example, that the category *Relational databases* (*catID=4*) lies under the categories *All books*, *Computer books*, and *Databases* (*parentID=11, 1, 2*).

Table 8-24. *Some Entries in the Table rel_cat_parent*

catID	parentID
1	11
2	1
2	11
3	1
3	11
4	1
4	2
4	11
	..

The significant drawback of the *rel_cat_parent* table is that it must be synchronized with every change in the *categories* table. But that is relatively easy to take care of.

In exchange for that effort, now the question of all categories ranked below *Databases* is easily answered:

```
SELECT catID FROM rel_cat_parent
WHERE parentID=2
```

If you would like to determine all book titles that belong to the category *Databases* or its subcategories, the requisite query looks like the following. The key word *DISTINCT* is necessary here, since otherwise, the query would return many titles with multiplicity:

```
SELECT DISTINCT titles.title FROM titles, rel_cat_parent
WHERE (rel_cat_parent.parentID = 2 OR titles.catID = 2)
      AND titles.catID = rel_cat_parent.catID
```

We are once more caught on the horns of the efficiency versus normalization dilemma: The entire table *rel_cat_parent* contains nothing but data that can be determined directly from *categories*. In the concrete realization of the *mylibrary* database, I decided to do without *rel_cat_parent*.

Searching for Higher-Ranked Categories in the categories Table

Here we confront the converse question to that posed in the last section: What are the higher-ranking categories above an initial, given category? If the initial category is *Perl* (*catID=7*), then the higher-ranking categories are first *Programming*, then *Computer books*, and finally, *All books*.

When there is a table *rel_cat_parent* like that described above, then our question can be answered by a simple query:

```
SELECT CONCAT(SPACE(hierIndent*2), catName) AS category
FROM categories, rel_cat_parent
WHERE rel_cat_parent.catID = 7
  AND categories.catID = rel_cat_parent.parentID
ORDER BY hierNr

category
All books
  Computer books
    Programming
```

On the other hand, if *rel_cat_parent* is not available, then a series of *SELECT* instructions must be executed in a loop *categories.parentCatID* until this contains the value *NULL*.

Relations

If you want to transform a database into normal form, you have to link a number of tables. These links are called *relations* in database-speak. At bottom, there are three possible relations between two tables:

1:1. In a one-to-one relation between two tables, each data record of the first table corresponds to precisely one data record of the second table and vice versa. Such relations are rare, since in such a case the information in both tables could as easily be stored in a single table.

1:*n*. In a one-to-many relation, a single record in the first table can correspond to several records in the second table (for example, a vendor can be associated with many orders). The converse may be impossible: A single order cannot, say, be filled by many vendors. Occasionally, one hears of an *n*-to-1 relation, but this is merely a 1-to-*n* relation from the opposite point of view.

***n*:*m*.** Here a data record in the first table can be linked to several records in the second table, and vice versa. (For example, several articles can be included in a single order, while the same article may be included in several different orders. Another example is books and their authors. Several authors may have written a single book, while one author may have written several books.)

1:1 Relations

A one-to-one relation typically comes into being when a table is divided into two tables that use the same primary key. This situation is most easily grasped with the aid of an example. A table containing a corporation's personnel records contains a great deal of information: name, department, date of birth, date of employment, and so on. This table could be split into two tables, called, say, *personnel* and *personnel_extra*, where *personnel* contains the frequently queried and generally accessible data, while *personnel_extra* contains additional, less used, and more private data.

There are two possible reasons for such a division. One is the security aspect: It is simple to protect the table *personnel_extra* from general access. (For MySQL, this argument is less important, since access privileges can in any case be set separately for each column of a table. Since MySQL 5.0, the security problem can also be solved with *Views*.)

The other reason is that of speed. If a table contains many columns, of which only a few are required by most queries, it is more efficient to keep the frequently used columns in a single table. (In the ideal situation, the first table would contain exclusively columns of a given size. Such tables are more efficient to manage than tables whose columns are of variable size. An overview of the types of tables supported by MySQL can be found in the first section of this chapter.)

The significant disadvantage of such a separation of tables is the added overhead of ensuring that the tables remain synchronized.

1:*n* Relations

One-to-many relations come into play whenever a particular field of a data record in a detail table can refer to various columns of another table (the master table).

The linkage takes place via key fields. The columns of the master table are identified by a primary key. The detail table contains a foreign key field, whose contents refer to the master table. Here are a few examples:

The *mylibrary* database: Here there is a one-to-many relation between the *titles* and *publishers* tables. The table *publishers* is the master table with the primary key *publishers.publID*.

Each publisher (1) can publish several books (*n*).

The *mylibrary* database contains two additional 1-to-*n* relations: between *titles* and *languages* (field *langID*) and between *titles* and *categories* (field *catID*).

A business application containing tables with orders: A detail table contains data on all processed orders. In this table, a foreign key field refers to the master table, with its list of all customers.

Each customer (1) can execute many orders (*n*).

Discussion groups containing tables with messages: A detail table contains data on every contribution to a discussion group in place on the website (title, text, date, author, group, etc.). Two possible master tables are a group table with a list of all discussion groups, and an author table with a list of all members of the website who are allowed to make contributions to a discussion.

Each author (1) can contribute to arbitrarily many discussions (*n*). Each discussion group (1) can contain arbitrarily many contributions (*n*).

A database containing tables of music CDs: A detail table contains data on every CD in the collection (title, performer, number of disks, etc.). Two possible master tables are a table containing a list of performers occurring in the database, and a recording label table with a list of recording companies.

Each performer (1) can appear on arbitrarily many CDs (*n*). Each label (1) can market arbitrarily many CDs (*n*).

■ **Note** Often during the creation of a database, one attempts to give the same name to fields of two tables that will later be linked by a relation. This contributes to clarity, but is not required.

It is also possible for the primary and foreign keys to be located in the same table. Then a data record in such a table can refer to another record in the same table. This is useful if a hierarchy is to be represented. Here are a few examples:

The *mylibrary* database: Each category in the *categories* table refers via the field *parentID* to a subcategory (or to *NULL*).

A table of personnel: In this table, each employee record (except for that of the top banana) refers to a field containing that individual's immediate supervisor.

Discussion groups, a table with messages: In these tables, each message refers to a field containing the next-higher message in the hierarchy (that is, the one to which the current message is responding).

A music database, containing tables with different types of music: Each style field refers to a field with the name of the genre of which the current style is a subset (for example, bebop within the category jazz, or string quartet within the category of chamber music).

n:m Relations

For *n:m* relations, it is necessary to add an auxiliary table to the two original tables so that the *n:m* relation can be reduced to two 1:*n* relations. Here are some examples:

The *mylibrary database:* Here we have an *n:m* relation between book titles and authors. The relation is established by means of the *rel_title_author* table.

A possible extension of this table could be an additional field that determines the order of authors (if it is not to be simply in alphabetical order).

Business application, a table with orders: To establish a relation between an order and the articles included in the order, the auxiliary table specifies how many of article *x* are included in order *y*.

This table could then consist of the columns *articleID*, *orderID*, *quantity*, and perhaps also *price*. To be sure, the price is contained in the articles table, but it could change over time, with the price at the time of purchase different from the current price. In this case, the orders table would contain the price that was valid at the time of the order.

College administration, list of exams: To keep track of which student has passed which exam and when and with what grade, it is necessary to have a table that stands between the table of students and the table of exams.

Primary and Foreign Keys

Relations depend intimately on primary and foreign keys. This section provides a comprehensive explanation of these two topics and their application. Alas, we cannot entirely avoid a bit of a detour into referring to various SQL commands which are not formally introduced until the end of this chapter and the chapter following.

Primary Key

The job of the primary key is to locate, as fast as possible, a particular data record in a table (for example, to locate the record with *id=314159* from a table of a million records). This operation must be carried out whenever data from several tables are assembled—in short, very often indeed.

With most database systems, including MySQL, it is also permitted to have primary keys that are formed from several fields of a table. Whether it is a single field or several that serve as primary key, the following properties should be satisfied:

- The primary key must be unique. It is not permitted that two records have the same content in their primary key field.
- The primary key should be compact, and there are two reasons for this:

  First, for the primary key it is necessary to maintain an index (the primary index) to maximize the speed of search (e.g., for *id=314159*). The more compact the primary field key, the more efficient the management of this index. Therefore, an integer is more suitable than a character string of variable length for use as a primary key field.

  Second, the content of the primary key field is used as a foreign key in other tables, and there, as well, it is efficient to have the foreign key as compact as possible. (Relations between tables are established not least to avoid wasted space on account of redundancies. This makes sense only if the use of key fields doesn't take up even more space.)

With most database systems it has become standard practice to use a 32- or 64-bit integer as primary key field, generated automatically in sequence (1, 2, 3, ...) by the database system. Thus neither the programmer nor the user need be concerned how a new and unique primary key value is to be found for each new record.

In MySQL such fields are declared as follows:

```
CREATE TABLE publishers
  (publID INT NOT NULL AUTO_INCREMENT,
   othercolumns ...,
   PRIMARY KEY (publID))
```

If we translate from SQL into English, what we have is this: The field *publID* is not permitted to contain *NULL*. Its contents are generated by the database (unless another value is explicitly inserted there). The field functions as a primary key; that is, MySQL creates an index to enable rapid search. It is thereby ensured that the *publID* value is unique when new records are input.

For tables in which one expects to make many new entries or changes, one should usually use *BIGINT* (64-bit integer) instead of *INT* (32 bits).

■**Note** The name of the primary key field plays no role. In this book we usually use *id* or *tablenameID*. Often, you will see combinations with *no* or *nr* (for "number") as, for example, in *customerNr*.

Foreign Keys

The task of the foreign key field is to refer to a record in the detail table. However, this reference comes into being only when a database query is formulated, for example, in the following form:

```
SELECT titles.title, publishers.publName FROM titles, publishers
WHERE titles.publID = publishers.publID
ORDER BY title
```

With this, an alphabetical list of all book titles is generated, in which the second column gives the publisher of the book. The result would look something like this:

```
title                 publName
Client/Server ...     Addison-Wesley
Definitive Guide ...  Apress
Linux                 Addison-Wesley
Web Application ...   New Riders
```

Decisive here is the clause *WHERE titles.publID = publishers.publID*. It is here that the link between the tables is created. Chapter 9 discusses some other ways of linking two tables with queries.

In the declaration of a table the foreign key plays no particular role. For MySQL, a foreign key field is just another ordinary table field. There are no particular key words that must be employed. In particular, no index is necessary (there is practically never a search to find the contents of the foreign key). Of course, you would not be permitted to supply the attribute *AUTO_INCREMENT*. After all, you want to specify yourself the record to which the field refers. You need to take care, though, that the foreign key field is of the same data type as the type of the primary key field. Otherwise, the evaluation of the *WHERE* condition can be very slow.

```
CREATE TABLE titles
  (othercolumns ...,
   publisherID INT NOT NULL)
```

Whether you specify the attribute *NOT NULL* depends on the context. In most cases, *NOT NULL* is to be recommended in order to avoid at the outset the occurrence of incomplete data. However, if you wish to allow, for example, that in the book database a book could be entered that had no publisher, then you should do without *NOT NULL*.

Referential Integrity (Foreign Key Constraints)

If you delete the author *Kofler* from the *authors* table in the *mylibrary* database, you will encounter problems in many SQL queries that access the books *Linux* and *Definitive Guide*. The *authID* number 1 specified in the *rel_title_author* table no longer exists in the *authors* table. In database language, one would put it like this: The referential integrity of the database has been damaged.

As a database developer, it is your responsibility to see that such events cannot happen. Therefore, before deleting a data record you must always check whether there exists a reference to the record in question in another table.

Since one cannot always rely on programmers, many databases have rules for maintaining referential integrity. So-called *foreign key constraints* (integrity rules) test at every change in the database whether any cross references between tables are affected. Depending on the declaration of the foreign key, there are then two possible consequences: Either the operation will simply not be executed (error message), or all affected records in dependent tables are deleted as well. Which modus operandi is to be preferred depends on the data themselves.

MySQL also offers such a control mechanism, though at the present only for InnoDB tables. The following lines show a fragment of SQL code for declaring a foreign key with integrity rules. (SQL commands for creating and editing tables are discussed in Chapter 9.)

```
CREATE TABLE titles
  (column1, column2,  ...,
   publID INT,
   FOREIGN KEY (publID) REFERENCES publishers (publisherID)
```

This means that *titles.publID* is a foreign key that refers to the primary key *publishers.publID*. With options such as *ON DELETE*, *RESTRICT*, and *ON DELETE CASCADE*, one may further specify how the database system is to respond to potential damage to its referential integrity.

```
CREATE TABLE titles
  (column1, column2,  ...,
   publID INT,
   FOREIGN KEY (publisherID) REFERENCES publishers (publID)
  )
```

This means that *titles.publID* is a foreign key that refers to the primary key *publishers.publID*. The integrity rule has the following effects on the two tables:

- In *titles*, you cannot insert a title with a *publID* number that does not exist in the *publishers* table. (Nor can you change the value of *publID* in *titles* for an existing title if there is no corresponding *publishers* data record.)
- You cannot delete a publisher from *publishers* that is referred to by the *titles* table. (Here as well the restriction on *UPDATE* commands is in effect.)

This has consequences for the order of operations: If you wish to store a new title for a new publisher, you must first enter the publisher and then the title. If you wish to delete a publisher and a title, you must first delete the title and then the publisher. (The latter operation is possible only if there are no additional titles from the publisher in question in the *titles* table.)

Syntax

The general syntax for defining a foreign key constraint for a foreign key field *table1.column1* looks as follows:

```
FOREIGN KEY [name] (column1) REFERENCES table2 (column2)
  [ON DELETE {CASCADE | SET NULL | NO ACTION | RESTRICT}]
  [ON UPDATE {CASCADE | SET NULL | NO ACTION | RESTRICT}]
```

The foreign key constraint can be named with *name*. The foreign key is *table1.column1* for which the constraint is defined. The field of the second table to which the constraint refers is *table2.column2*. (In many cases, *column2* is the primary key of *table2*. This is not requisite, but *column2* must be equipped with an index.)

One may also have foreign key constraints in which *table1* and *table2* are the same table. This makes sense when a table possesses references to itself. This is the case, for example, with the *categories* table in *mylibrary*, where *parentCatID* refers to *CatID*, thus creating a hierarchical relationship among the categories.

Actions When Integrity Is Damaged

The optional clause *ON DELETE* determines how the table driver should behave if a record that is referred to by *table1* is deleted from *table2*. There are four possibilities:

RESTRICT is the default behavior. The *DELETE* command causes an error, and the record is not deleted. (An error does not necessarily mean the termination of a running transaction. The command is simply not executed. The transaction must, as usual, be terminated with a *COMMIT* or *ROLLBACK*.)

SET NULL has the effect that the record from *table2* is allowed to be deleted. In *table1*, all records that refer to the deleted record will have *column1* set to *NULL*. This rule assumes that *NULL* is a permitted value for *table1.column1*.

For the *titles/publisher* example, this means that if you remove publisher *x* from *publishers*, then all records in *titles* that were published by this publisher will have their *publID* set to *NULL*.

CASCADE has the effect that the record from *table2* is allowed to be deleted. At the same time, however, all records from *table1* that refer to that record will be deleted.

For the *titles/publishers* example, this means that if you delete publisher *x* from the *publishers* table, then all records in *titles* that were published by publisher *x* will be deleted from *titles*.

NO ACTION has the effect that the loss of referential integrity is tolerated. This action seldom makes sense, since it is easier simply not to use a foreign key constraint.

These four actions can also be specified analogously for *ON UPDATE* (where by default, *RESTRICT* again holds). The *UPDATE* rules come into effect if in *table2*, the key field of an existing record is altered. In such a case, the effect of *RESTRICT*, *SET NULL*, and *NO ACTION* is the same as with *ON DELETE*.

On the other hand, the effect of *CASCADE* is a bit different: A change in the key field in *table2* is now also carried out in the foreign key field in *table1*. For the *titles/publishers* example, this means that if you change the *publID* field for publisher *x* in *publishers*, then the *publID* field will be updated for all affected records in *titles*.

■ **Note** Integrity rules do not prevent you from deleting an entire table. For example, you can execute *DROP TABLE publishers*, even if this means damaging the referential integrity.

Conditions for Setting Up Integrity Rules

Foreign key constraints can be used only if a set of preconditions is satisfied. If such conditions have not been satisfied, then the result is an error message, generally *Error 1005: Can't create table xxx (errno: 150)*, and the constraint is not stored. (The cause of the error can be quite trivial, such as a typo in the name of a column.)

- Each of *table1.column1* and *table2.column2* must be equipped with at least an ordinary index. This index is not created by *FOREIGN KEY*, and must therefore be explicitly provided for in *CREATE TABLE* or after the fact with *ALTER TABLE*.

  table2.column2 is often the primary key field of *table2*, but that is not necessary.

  If you use keys across several fields (*INDEX(columnA, columnB)*), then the key field from the foreign key constraint must appear first. Otherwise, an additional separate index for the field must be created.
- The data types of *table1.column1* and *table2.column2* must agree to the extent that a direct comparison is possible without transformation of data types. Most efficient is if both fields are declared with *INT* or *BIGINT*. Both columns must be of the same sign type (*SIGNED* or *UNSIGNED*).
- If the optional rule *ON DELETE/UPDATE SET NULL* is defined, then the value *NULL* must be permitted in *table1.column1*.
- The foreign key constraint must be satisfied from the start: If the tables are already filled with data, it can happen that individual records do not conform to the integrity rules. In this case, an *ALTER TABLE* command results in error 1216 (*A foreign key constraint fails*). The records must be corrected before the constraint can be set up.

■ **Tip** If an error occurs in setting up a foreign key constraint, you can obtain more precise information about the cause of the error with *SHOW INNODB STATUS*.

Finding Unsatisfied Integrity Rules

The obvious question upon receipt of error 1216 is, "How can the error-ridden records be found?" A simple sub*SELECT* command returns all records in the *titles* table for which *titles.publID* contains a value for which there is no corresponding value in *publishers.publID*:

```
SELECT titleID, publID FROM titles
WHERE publID NOT IN (SELECT publID FROM publishers)

titleID  publID  publishers.publID
     66      99    NULL
```

The title with *titleID=66* thus refers to a publisher with *publID=99* in the *publishers* table, but there is no such ID number in that table. Now you must either insert the missing publisher or delete the title with ID equal to 66. (In the *titles* table you could also change erroneous *publID*s to *NULL*, though in practice, that is often not allowed in linked tables.)

Deleting Foreign Key Constraints

To delete a foreign key constraint, execute *ALTER TABLE*:

```
ALTER TABLE tablename DROP FOREIGN KEY foreign_key_id
```

You can determine the *foreign_key_id* of the index to be deleted with *SHOW CREATE TABLE*. Note that deleting foreign key constraints can cause problems if you are using replication at the same time. The reason is that the foreign key index can have a different name in the duplicated database from that in the original database.

Temporarily Deactivating Integrity Checks

With *SET foreign_key_checks=0* you can turn off the automatic checking of integrity rules. This can make sense, for example, to speed up the reading in of large backup tables. *SET foreign_key_checks=1* reactivates the rules. While the rules are deactivated, changes in the database will not be checked. If integrity rules have been violated, the errors incurred will not be automatically recognized.

Foreign Key Constraints for the mylibrary Database

The *mylibrary* database consists exclusively of InnoDB tables. All the relations among the tables are secured with integrity rules, as shown in Table 8-25.

Table 8-25. *Foreign Keys and Referenced Keys in the mylibrary Database*

Foreign Key	Referenced Key
titles.publID	*publishers.publID*
titles.langID	*languages.langID*
titles.catID	*categories.catID*
categories.parentCatID	*categories.catID*
rel_title_author.titleID	*titles.titleID*
rel_title_author.authID	*authors.authID*

Indexes

If you are searching for a particular record in a table or would like to create a series of data records for an ordered table, MySQL must load *all* the records of the table. The following lines show some of the relevant *SELECT* commands (details to follow in the next chapter):

```
SELECT column1, column2 ... FROM table WHERE column3=12345
SELECT column1, column2 ... FROM table ORDER BY column3
SELECT column1, column2 ... FROM table WHERE column3 LIKE 'Smith%'
SELECT column1, column2 ... FROM table WHERE column3 > 2000
```

With large tables, performance will suffer under such everyday queries. Fortunately, there is a simple solution to cure our table's performance anxiety: Simply use an index for the affected column (in the example above, for *column3*).

An index is a special file or, in the case of InnoDB, a part of the tablespace, containing references to all the records of a table. (Thus a database index functions like the index in this book. The index saves you the trouble of reading the entire book from one end to the other if you simply want to find out where a particular topic is covered.)

Caution Indexes are not a panacea! They speed up access to data, but they slow down each alteration in the database. Every time a data record in changed, the index must be updated. This drawback can be ameliorated to some extent with various SQL commands by means of the option *DELAY_KEY_WRITE*. The effect of this option is that the index is not updated with each new or changed record, but only now and then. *DELAY_KEY_WRITE* is useful, for example, when many new records are to be inserted in a table as quickly as possible.

A further apparent disadvantage of indexes is that they take up additional space on the hard drive. Therefore, use indexes only for those columns that will often be searched and sorted. Indexes remain largely useless when the column contains many identical entries. (In such cases, you might ask yourself whether the normalization of the database has been optimally carried out.)

In principle, an index can be created for each field of a table, up to a maximum of sixteen indexes per table. (MySQL also permits indexes for several fields simultaneously. That makes sense if sorting is frequently carried out according to a combination of fields, as in *WHERE country='Austria' AND city='Graz'*).

Indexes for InnoDB Tables

Indexes are more important for InnoDB tables than for MyISAM tables. In the former, indexes are used not only for searching for records, but also for *row level locking*. This means that during a transaction, individual records are barred from access by other users. This affects, among others, the commands *SELECT … LOCK IN SHARE MODE*, *SELECT … FOR UPDATE*, and *INSERT*, *UPDATE*, and *DELETE*. (There is more on transactions in Chapter 10.)

The internal labeling of locked records takes place, for the sake of efficiency, not in the actual tables, but in the index. This works, of course, only if a suitable index is available.

Limitations

- MySQL cannot use indexes where inequality operators are used (*WHERE column != …*).
- Likewise, indexes cannot be used for comparisons where the contents of the column are processed by a function (*WHERE DAY(column)= …*).
- With *JOIN* operations (that is, in uniting data from various tables), indexes are of use only when primary and foreign keys refer to the same data type.
- If the comparison operators *LIKE* and *REGEXP* are used, an index is of use only when there is no wild card at the beginning of the search pattern. With *LIKE 'abc%'* an index is of use, but with *LIKE '%abc'*, it is not.
- Indexes are used with *ORDER BY* operations only if the records do not have to be previously selected by other criteria. (Unfortunately, an index rarely helps to speed up *ORDER BY* with queries in which the records are taken from several tables.)
- Indexes are ineffectual if a column contains the same value over and over. It is therefore not advisable to index a column with 0/1 or Y/N values.

Ordinary Indexes, Unique Indexes, Primary Indexes

Ordinary Index

The only task of an ordinary index (definition via the keyword *KEY* or *INDEX*) is to speed up access to data. You should therefore index columns that you frequently use in conditions (*WHERE column=…*) or for sorting (*ORDER BY column*). If possible, index columns with compact data (e.g., integers).

Unique Index

With an ordinary index it is allowed for several data records in the indexed field to refer to the same value. (In a table of personnel, for example, the same name can appear twice, even though it refers to two distinct individuals.)

When it is clear from context that a column contains unique values, you should then define an index with the key word *UNIQUE*. This has two consequences. One is that MySQL has an easier time managing the index; that is, the index is more efficient. The other is that MySQL ensures that no new record is added if there is already another record that refers to the same value in the indexed field. (Often, a *UNIQUE* index is defined for this reason alone, that is, not for access optimization, but to avoid duplication.)

Primary Index

For primary key fields, mentioned repeatedly in the previous section, a primary index must be defined. This involves a *UNIQUE* index that is distinguished only in that it has the name *PRIMARY*.

Foreign Key Index

Even if you define an integrity rule for a foreign key field (see the previous section), MySQL defines an internal index. This index serves to maintain the foreign key constraint as efficiently as possible.

Combined Indexes

An index can cover several columns, as in *INDEX(columnA, columnB)*. A peculiarity of such indexes is that MySQL can selectively use such an index. Thus when a query requires an index for *columnA* only, the combined index for *INDEX(columnA, columnB)* can be used. This holds, however, only for partial indexes at the beginning of the series. For instance, *INDEX(A, B, C)* can be used as index for *A* or *(A, B)*, but not as index for *B* or *C* or *(B, C)*.

Limits on the Index Length

In the definition of an index for *CHAR* and *VARCHAR* columns you can limit an index to a particular number of characters (which must be smaller than the maximum number of characters allowed in this field). The consequence is that the resulting index file is smaller and its evaluation quicker than otherwise. In most applications, that is, with character strings representing names, perhaps ten to fifteen characters altogether suffice to reduce the search set to a few data records.

With *BLOB* and *TEXT* columns you must institute this restriction, where MySQL permits a maximal index length of 255 characters.

Full-Text Index

An ordinary index for text fields helps only in the search for character strings that stand at the beginning of the field (that is, whose initial letters are known). On the other hand, if you store texts in fields that consist of several, or possibly very many, words, an ordinary index is useless. The search must be formulated in the form *LIKE '%word%'*, which for MySQL is rather complex and with large data sets leads to long response times.

In such cases it helps to use a full-text index. With this type of index, MySQL creates a list of all words that appear in the text. A full-text index can be created during the database design or afterwards:

```
ALTER TABLE tablename ADD FULLTEXT(column1, column2)
```

In *SELECT* queries, one can now search for records that contain one or more words. This is the query syntax:

```
SELECT * FROM tablename
WHERE MATCH(column1, column2) AGAINST('word1', 'word2', 'word3')
```

Then all records will be found for which the words *word1*, *word2*, and *word3* appear in the columns *column1* and *column2*.

■ **Note** The InnoDB table driver does not support full-text indexes.

An extensive description of the SQL syntax for full-text search, together with a host of application examples, can be found in Chapter 10.

Query and Index Optimization

Realistic performance estimates can be made only when the database has been filled with a sufficient quantity of test data. A test database with several hundred data records will usually be located entirely in RAM after the first query, and all queries will be answered quickly with or without an index. Things become interesting when tables contain well over 1000 records and when the entire size of the database is larger than the total RAM of the MySQL server.

In making the decision as to which columns should be provided with indexes, one may sometimes obtain some assistance from the command *EXPLAIN SELECT*. This is simply an ordinary *SELECT* command prefixed with the key word *EXPLAIN*. Instead of *SELECT* being simply executed, MySQL places information in a table as to how the query was executed and which indexes (to the extent that they exist) came into play.

Here are some pointers for interpreting the table created by *EXPLAIN*. In the first column appear the names of the tables in the order in which they were read from the database. The column *type* specifies how the table is linked to the other tables (*JOIN*). This functions most efficiently (i.e., quickly) with the type *system*, while more costly are the types *const*, *eq_ref*, *ref*, *range*, *index*, and *ALL*. (*ALL* means that for each record in the next-higher table in the hierarchy, all records of this table must be read. That can usually be prevented with an index.

The column *possible_keys* specifies which indexes MySQL can access in the search for data records. The column *key* specifies which index MySQL has actually chosen. The length of the index in bytes is given by *key_len*. For example, with an index for an *INTEGER* column, the number of bytes is 4. Information on how many parts of a multipart index are used is also given by *key_len*. As a rule, the smaller *key_len* is the better (that is, the faster).

The column *ref* specifies the column of a second table with which the linkage was made, while *rows* contains an estimate of how many records MySQL expects to read in order to execute the entire query. The product of all the numbers in the *rows* column allows one to draw a conclusion as to how many combinations arise from the query.

Finally, the column *extra* provides additional information on the *JOIN* operation, for example, *using temporary* when MySQL must create a temporary table in executing a query.

■ **Tip** Though the information proffered by *EXPLAIN* is often useful, the interpretation requires a certain amount of MySQL and database experience. You will find further information in the MySQL documentation: `http://dev.mysql.com/doc/mysql/en/query-speed.html` and `http://dev.mysql.com/doc/mysql/en/explain.html`.

A quite readable presentation on MySQL speed optimization can be found in the OpenOffice format: `http://dev.mysql.com/tech-resources/presentations/` and `http://dev.mysql.com/Downloads/Presentations/OSCON-2004.sxi`.

Example 1

This query produces an unordered list of all books with all their authors. All *ID* columns are equipped with primary indexes.

```
USE mylibrary
EXPLAIN SELECT * FROM titles, rel_title_author, authors
  WHERE rel_title_author.authID =  authors.authID
  AND   rel_title_author.titleID = titles.titleID
```

table	type	key	key_len	ref	rows	Extra
titles	ALL	authName	60	NULL	53	Using index
rel_title_author	ref	authID	4	authors.authID	1	Using index
authors	eq_ref	PRIMARY	4	rel_title_author.titleID	1	

This *EXPLAIN* result means that first all records from the *titles* table are read, with the index *authName* being used. (This was not actually necessary, since the *SELECT* command does not require the sorting of results.) Then with the help of the *authID* index of *rel_title_author* and the primary index of *authors*, the links to the two other tables are made. The tables are thus optimally indexed, and for each part of the query there are indexes available.

To save space, some of the columns of the *EXPLAIN* result were removed, including *possible_keys*. This column contains a list of all indexes that were used for the corresponding part of the query. The column *key* tells which of these indexes MySQL chose.

Example 2

Here the query produces a list of all books (together with their authors) that have been published by a particular publisher. The list is ordered by book title. Again, all *ID* columns are equipped with indexes. Furthermore, in the *titles* table, *title* and *publID* are indexed:

```
EXPLAIN SELECT title, authName
FROM titles, rel_title_author, authors
WHERE titles.publID=2
  AND titles.titleID = rel_title_author.titleID
  AND authors.authID = rel_title_author.authID
ORDER BY title
```

table	type	key	key_len	ref	rows	Extra
titles	ref	publIDIndex	5	const	4	Using where; Using filesort
rel_title_author	ref	PRIMARY	4	titles.titleID	2	Using index
authors	eq_ref	PRIMARY	4	rel_title_author.authID	1	

The interpretation is this: The tables are optimally indexed; that is, for each part of the query there are indexes available. It is interesting that the title list (*ORDER BY title*) is apparently sorted externally, although there is an index for the *title* column as well. The reason for this is perhaps that the *title* records are first selected in accordance with the condition *publID=2*, and the *title* index can then no longer be applied.

Example 3

This example uses the same *SELECT* query as does Example 2, but it assumes that *titles.publID* does not have an index. The result is that now all 53 records of the *titles* table must be read. The index *authName* is indeed used, but it is of no help in speeding up the query. *Using temporary* means that a temporary table holding intermediate results was created.

table	type	key	key_len	ref	rows	Extra
titles	index	authName	60	NULL	53	Using index; Using temporary; Using filesort
rel_title_author	ref	PRIMARY	4	titles.titleID	4	Using index
authors	eq_ref	PRIMARY	4	rel_title_author.authID	4	Using where

Views

Views make it possible to define a special representation of one or more tables. A view behaves much like a table. That is, you can query data with *SELECT* queries and (depending on the definition of the view) alter data with *INSERT*, *UPDATE*, and *DELETE*.

Views have been available since MySQL 5.0. This section assumes that you are familiar with *SELECT* commands (see the following chapter) and that you understand MySQL access privileges (see Chapter 11). Note that phpMyAdmin 2.6, the version current at the time of writing, still cannot deal with views.

There are two basic reasons for using views:

Security: It can happen that you wish to prevent certain users of a database from having full access to a table. A typical example is a personnel table in a business setting, in which data on all employees are stored. You would like all users to have access to certain data (names and telephone numbers) but not to others (birth date and salary).

The solution is a view that contains the columns accessible by everyone. You have to set the MySQL access privileges in such a way that the user is allowed to access the view, but not the underlying table.

Convenience: In many applications, the same queries must be executed repeatedly to collect data from one or more tables according to certain requirements. Instead of forcing every user or programmer to formulate repeatedly the same complex *SELECT* commands, you, as the database administrator, can define a view.

The Definition of a View

Views act like virtual tables containing the result of a *SELECT* query. So it is no surprise that the definition is based on a *SELECT* command. The following two examples define two views for the *mylibrary* database and show five records for each of these views.

```
CREATE VIEW v1 AS
  SELECT titleID, title, subtitle FROM titles
  ORDER BY title, subtitle
SELECT * FROM v1 LIMIT 5
```

```
titleID  title                          subtitle
     11  A Guide to the SQL Standard     NULL
     52  A Programmer's Introduction ... NULL
     19  Alltid den där Annette          NULL
     51  Anklage Vatermord               Der Fall Philipp Halsmann
     78  Apache Webserver 2.0            Installation, ...
CREATE VIEW v2 AS
  SELECT title, publname, catname FROM titles, publishers, categories
  WHERE titles.publid=publishers.publid
  AND titles.catID = categories.catID
  AND langID=2
SELECT * FROM v2 ORDER BY title LIMIT 5

title                     publname         catname
Anklage Vatermord         Zsolnay          Literature and fiction
Apache Webserver 2.0      Addison-Wesley   Computer books
CSS-Praxis                Galileo          Computer books
Ein perfekter Freund      Diogenes Verlag  Literature and fiction
Excel 2000 programmieren  Addison-Wesley   Programming
```

To execute *CREATE VIEW* you must have the *Create View* privilege. What privileges are and how they are managed will be discussed in Chapter 11.

Changing View Records

Whether the commands *INSERT*, *UPDATE*, and *DELETE* can be used with a view (that is, whether the view is *updatable*) depends on the underlying *SELECT* command. For changeable views, the following rules hold:

- The *SELECT* command may not contain *GROUP BY*, *DISTINCT*, *LIMIT*, *UNION*, or *HAVING*.
- Views that process data from more than one table are almost always unchangeable.
- The view should contain all columns for which primary or unique indexes or foreign key constraints have been defined. If such columns are missing from the view, the MySQL option `updatable_views_with_limit` decides whether changes should be made with a warning (current default setting 1) or should trigger an error (setting 0).

View Options

The complete syntax of the *CREATE VIEW* command looks like this:

```
CREATE [OR REPLACE] [ALGORITHM = UNDEFINED | MERGE | TEMPTABLE]
VIEW name [(columnlist)] AS select command
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

The following points provide a brief explanation of the various options:

- *OR REPLACE* means that an existing view should be replaced by the new view without an error message.
- *ALGORITHM* tells how the view is represented internally. This option was, alas, undocumented as this section was written. By default, MySQL always uses *UNDEFINED* (can be determined via *SHOW CREATE TABLE viewname*).
- *WITH CHECK OPTION* means that changes to view records are allowed only if the *WHERE* conditions of the *SELECT* command are satisfied. *WITH CHECK OPTION* is relevant, of course, only if the view is changeable.

The variant *WITH LOCAL CHECK OPTION* affects views that themselves are derived from other views (which is allowed!). *LOCAL* means that only the *WHERE* conditions of the *CREATE VIEW* command are considered, and not the *WHERE* conditions of the subordinate views.

The exact opposite effect is achieved by *CASCADED CHECK OPTION*: The *WHERE* conditions of all subordinate views are considered. If you specify neither *CASCADED* nor *LOCAL*, then the default *CASCADED* holds.

Viewing View Definitions

Just as you can determine with *SHOW CREATE TABLE name* the SQL code of the command for creating a table, *SHOW CREATE VIEW name* is possible for views. You must have the *Create View* privilege in order to be able to execute *CREATE VIEW*.

```
SHOW CREATE VIEW v1

CREATE ALGORITHM=UNDEFINED VIEW `mylibrary`.`v1` AS
  SELECT `mylibrary`.`titles`.`titleID`,
         `mylibrary`.`titles`.`title`,
         `mylibrary`.`titles`.`subtitle`
  FROM `mylibrary`.`titles`
  ORDER BY `mylibrary`.`titles`.`title`,
           `mylibrary`.`titles`.`subtitle`
```

Deleting Views

SHOW TABLES returns a list of all tables and views. To delete views, though, you cannot use *DROP TABLE*. Instead, you must use the command *DROP VIEW viewname*.

Example Database mylibrary (Library Management)

In the course of this chapter I have presented several aspects of database design in reference to the *mylibrary* database. This has caused the description of the *mylibrary* database to have been so extended that you may have lost an overview of the tables, fields, and indexes that this database comprises, the data types used, and so on. Therefore, this section offers a summary of all the properties of the *mylibrary* database. The totality of these properties is usually called a *database schema*.

Figure 8-2 presents a graphical summary of the schema. The figure shows neither the data types of the columns nor all the indexes, but nevertheless, one sees clearly which columns are used as primary indexes and how the tables are related one to the other. (Like all the schema figures in this book, this one was created with the query designer in OpenOffice.)

■ **Note** A `*.sql` file with the complete definition of the *mylibary* database containing test data records can be found in the companion files to this book. To read in this file, use phpMyAdmin to create an empty database with *Latin1* as the default character set, then change to the page *SQL*, and load the `*.sql` file. Alternatively, you can create the database with the following two commands:

```
> mysqladmin -u root -p create mylibrary
> mysql -u root -p mylibrary < mylibrary.sql
```

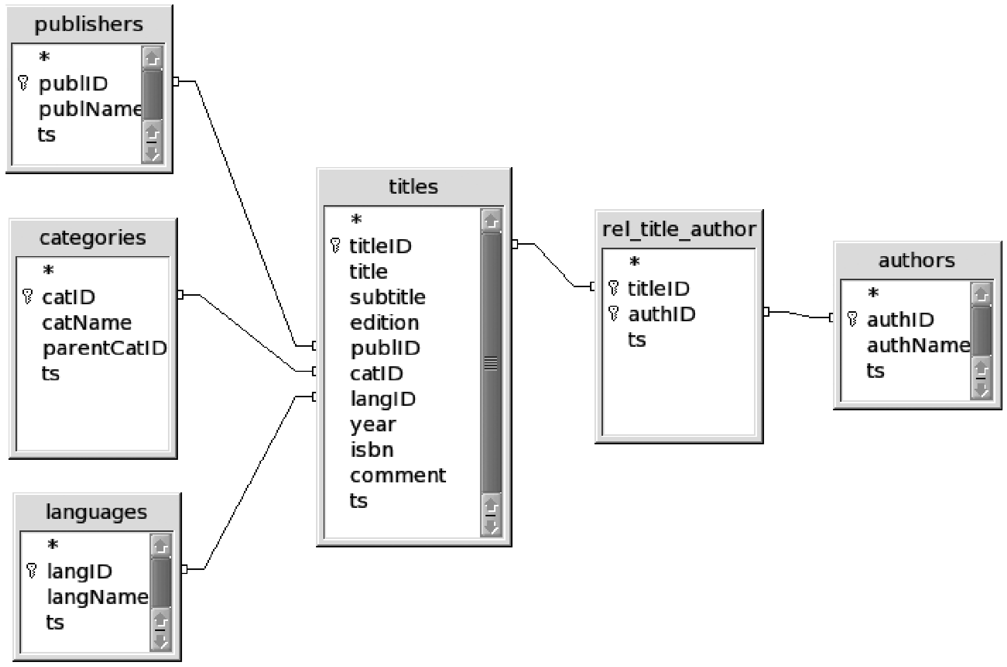


Figure 8-2. *The schema of the mylibrary database*

Properties of the Database

For the database *mylibrary* and all the text fields contained therein, the default character set is *latin1* and the sort order is *latin1_german1_ci*.

Properties of the Tables

All the tables are InnoDB tables. Tables 8-26 through 8-31 specify for each *mylibrary* table all fields, data types, attributes, and indexes. The meaning of most of the columns does not require additional explication. Worthy of note and as yet unexplained is the *ts* column that appears in each table. In this column the time of the last change is automatically logged. This column is necessary so that the database can be used by ODBC/ADO programs.

Table 8-26. *Properties of the authors Table*

Field	Data Type	Attribute	Index, Foreign Key Constraints
authID	*INT*	*NOT NULL AUTO_INCREMENT*	*PRIMARY KEY*
authName	*VARCHAR(60)*		*KEY*
ts	*TIMESTAMP*		

Table 8-27. *Properties of the categories Table*

Field	Data Type	Attribute	Index, Foreign Key Constraints
catID	*INT*	*NOT NULL AUTO_INCREMENT*	*PRIMARY KEY*
catName	*VARCHAR(60)*	*NOT NULL*	*KEY*
parentCatID	*INT*		*KEY, FOREIGN KEY categories.catID*
ts	*TIMESTAMP*		

Table 8-28. *Properties of the languages Table*

Field	Data Type	Attribute	Index, Foreign Key Constraints
langID	*INT*	*NOT NULL AUTO_INCREMENT*	*PRIMARY KEY*
langName	*VARCHAR(60)*	*NOT NULL*	*KEY*
ts	*TIMESTAMP*		

Table 8-29. *Properties of the publishers Table*

Field	Data Type	Attribute	Index, Foreign Key Constraints
publID	*INT*	*NOT NULL AUTO_INCREMENT*	*PRIMARY KEY*
publName	*VARCHAR(60)*	*NOT NULL*	*KEY*
ts	*TIMESTAMP*		

Table 8-30. *Properties of the rel_title_author Table*

Field	Data Type	Attribute	Index, Foreign Key Constraints
authID	*INT*	*NOT NULL*	*PRIMARY KEY, FOREIGN KEY authors.authID*
titleID	*INT*	*NOT NULL*	*PRIMARY KEY, KEY, FOREIGN KEY titles.titleID*
ts	*TIMESTAMP*		

Table 8-31. *Properties of the titles Table*

Field	Data Type	Attribute	Index, Foreign Key Constraints
titleID	*INT*	*NOT NULL AUTO_INCREMENT*	*PRIMARY KEY*
title	*VARCHAR(100)*	*NOT NULL*	*KEY*
subtitle	*VARCHAR(100)*		
edition	*TINYINT*		
publID	*INT*		*KEY, FOREIGN KEY publishers.publID*

Field	Data Type	Attribute	Index, Foreign Key Constraints
catID	*INT*		*KEY,* *FOREIGN KEY categories.catID*
langID	*INT*		*KEY,* *FOREIGN KEY languages.langID*
year	*INT*		
isbn	*VARCHAR(20)*		
comment	*VARCHAR(255)*		
ts	*TIMESTAMP*		

Example Database myforum (Discussion Group)

Most of the examples in this book use the *mylibrary* database, which we have been talking about throughout this chapter. Since all SQL concepts cannot be revealed by a single database, and particularly such a small one as *mylibrary*, two additional databases will be presented in this and the next sections: *myforum* and *exceptions*.

The first of these is a database for a real PHP application, whose code is not discussed in the book due to lack of space. What is at issue here is rather the design (schema) of the database. The database contains more than one thousand records and thus forms a good basis for trying out full-text search with MySQL (see Chapter 10).

There is no application behind the database *exceptions*. This database serves exclusively for experimenting with unusual MySQL data types and testing various sort orders.

The Discussion Group Database myforum

Among the best-loved MySQL applications are guest books, discussion groups, and other websites that offer users the possibility of creating a text and thereby adding their own voices to the website. The database *myforum* shows how a database can serve as the basis for a discussion group. The database consists of three tables:

- *forums* contains a list of the names of all discussion groups. Furthermore, each group can be assigned a particular language (English or German).
- *users* contains a list of all users registered in the database who are permitted to contribute to discussions. For each registered user, a login name, password, and email address are stored. The column *authent* contains a random character string that is sent to a new user by email at the time of registration. The account is activated only after the user clicks on the link contained in the email (column *active*) and supplies the random number.
- *messages* contains all the stored contributions. These consist of a *Subject* row, the actual text, *forumID*, *userID*, and additional management information. This table is the most interesting of the three from the standpoint of database design, and it will be considered further in this section.

MyISAM tables are used for this database, since at present only this format supports full-text search. All columns with strings use the *Latin1* character set and the sort order *latin1_german1_ci*. The schema for the database is summarized in Figure 8-3.

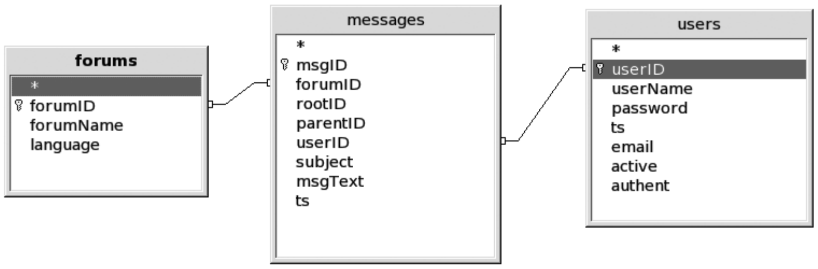


Figure 8-3. *The myforum database schema*

The *myforum* database, which is included in the files accompanying this book, contains more than 3,000 forum contributions from the website `www.kofler.cc`. (For this book, the authors' names have been concealed.) This database is well suited for trying out various aspects of full-text search (see also Chapter 10).

■ **Note** The *messages* table is a good demonstration that a full-text index can occupy a considerable amount of space. In the test database, *messages* contains about 3000 messages with about 1.5 million characters altogether. For storing the *messages* data about 1.7 megabytes are required. The storage requirement for the full-text index is another 1.4 megabytes, which practically doubles the requirement for the entire table!

Hierarchies Among Messages

As we have already seen in the case of the *mylibrary* database, there is a battle in *myforum* in dealing with hierarchies. A significant feature of discussion groups is that the discussion thread is represented in a hierarchical list. The hierarchy is the result of the fact that each contribution can elicit a response. A discussion among the five participants Antony, Banquo, Coriolanus, Duncan, and Edmund (*A*, *B*, *C*, *D*, and *E* for short) might look like that depicted in Table 8-32.

Table 8-32. *Discussion Threads*

Sample Messages
A: How do I sort with MySQL Tables? (17.1.2005 12:00)
B: first answer (17.1.2005 18:30)
A: thanks! (17.1.2005 19:45)
C: better suggestion (19.1.2005 10:30)
D: second answer (18.1.2005 3:45)
A: I don't understand that (18.1.2005 9:45)
D: the same explanation, but with more detail (18.1.2005 22:05)
E: third answer (18.1.2005 19:00)

Here as illustration a description of the content is added to the title line (*subject*) of each contribution. In reality, the title line usually contains just the thread title.

A representation in database format might look like that shown in Table 8-33. It is assumed here that the five participants *A* through *E* have *userID* numbers 201 through 205. The messages of the thread have *msgID* numbers that begin with 301. (In practice, it is natural to expect that a thread will not exhibit sequential *msgID* numbers. More likely, other threads will break into the sequence.) The table is sorted chronologically (that is, in the order in which the messages were posted).

Table 8-33. *The messages Table with the Data Records of a Discussion Thread*

msgID	forumID	rootID	parentID	userID	subject	ts
301	3	301	NULL	201	How do I…	2005-01-17 12:00
302	3	301	301	202	first answer	2005-01-17 18:30
303	3	301	302	201	thanks!	2005-01-17 19:45
304	3	301	301	204	second answer	2005-01-18 3:45
305	3	301	304	201	I don't understand…	2005-01-18 9:45
306	3	301	301	205	third answer	2005-01-18 19:00
307	3	301	304	204	the same…	2005-01-18 22:05
308	3	301	302	203	better suggestion	2005-01-19 10:30

The hierarchy is expressed via the column *parentID*, which refers to the message up one level in the hierarchy. If *parentID* has the value *NULL*, then the contribution is one that began a new thread.

The column *rootID* refers to the first message of a thread. This column may not contain *NULL*. For the first message of a thread, one has *msgID=rootID*. The latter column actually contains redundant information, since one can determine via *parentID* the start message for any message. However, *rootID* simplifies this task and greatly increases the efficiency of the entire forum (for example, in determining the number of messages in a thread or assembling all the messages of a thread with a simple *SELECT* query).

Example Database Exceptions (Special Cases)

When you begin to develop an application with a new database, programming language, or API that is unfamiliar to you it is often practical to implement a simple test database for quickly testing various special cases. For the work on this book, as well as for testing various APIs and various import and export tests, the database *exceptions* was used. Among the tables of this database are the following:

- Columns with most data types supported by MySQL, including *xxxTEXT*, *xxxBLOB*, *SET*, and *ENUM*
- *NULL* values
- Texts and BLOBs with all possible special characters
- All 255 text characters (code 1 to 255)

The following paragraphs provide an overview of the tables and their contents. The column names indicate the data type (thus the column *a_blob* has data type *BLOB*). The *id* column is an *AUTO_INCREMENT* column (type *INT*). In all the columns except the *id* column the value *NULL* is allowed.

The Table testall

This table contains columns with the most important MySQL data types (though not all types).

Columns: *id, a_char, a_text, a_blob, a_date, a_time, a_timestamp, a_float, a_decimal, a_enum, a_set*

The Table text_text

With this table you can test the use of text.

Columns: *id, a_varchar (maximum 100 characters), a_text, a_tinytext, a_longtext*

The Table test_blob

With the *test_blob* table you can test the use of binary data (import, export, reading and storing a client program, etc.).

Columns: *id, a_blob*

Content: A record (*id=1*) with a 512-byte binary block. The binary data represent byte for byte the codes 0, 1, 2, …, 255, 0, 1, …, 255

The Table test_date

With this table you can test the use of dates and times.

Columns: id, a_date, a_time, a_datetime, a_timestamp

Content: A data record (*id=1*) with the values *2005-12-31, 23:59:59* and again *2005-12-31, 23:59:59*

The Table test_enum

With this table you can test the use of *SET* and *ENUM*.

Columns: *id, a_enum,* and *a_set* (with the character strings 'a', 'b', 'c', 'd', 'e')

Content:

```
id  a_enum   a_set
1   a        a
2   e        b,c,d
3
4   NULL     NULL
```

The Table test_null

With this table you can test whether *NULL* (first record) can be distinguished from an empty character string (second record).

Columns: *id, a_text*

Content:

```
id  a_text
1   NULL
2
3   'a text'
```

The Table test_sort1

With this table you can test various sort orders (*ORDER BY*) for the *Latin1* and *UTF-8* (Unicode) character sets. The table consists of three columns: *id* contains sequential numbers from 33 to 126 and from 161 to 255; *latin1char* contains the *Latin1* characters for the code in *id*; *utf8char* contains the associated Unicode characters. The two *CHAR* columns were deliberately not given an index.

Content:

```
id    latin1char   ut8char
...
65    'A'          'A'
66    'B'          'B'
...
```

The Table test_sort2

This table also serves for testing sort orders. It contains again three columns, *id*, *latin1text*, and *utf8text* (data type *VARCHAR(100)* for both character sets *Latin1* and *UTF8*). This time, the text columns contain entire words, namely *abc*, *Abc*, *ABC*, *Bar*, *Bär*, *Bären*, *Barenboim*, *bärtig*, and *Ärger*. In the next chapter you will find examples using the tables *test_sort1* and *test_sort2*.

The Tables importtable1, importtable2, exporttable

These three tables contain test data for importation and exportation of text files. A description of the tables as well as numerous possibilities for import and export by MySQL can be found in Chapter 14.