The Definitive Guide to MySQL
Second Edition

MICHAEL KOFLER
Translated by DAVID KRAMER

Apress™

The Definitive Guide to MySQL, Second Edition

ISBN (pbk): 1-59059-144-5

Printed and bound in the United States of America 10987654321

Translator, Editor, and Compositor: David Kramer

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Production Manager: Kari Brooks

Proofreader: Elizabeth Berry

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

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, email 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 Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 5

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.

This chapter discusses the fundamentals of relational databases, collects the varieties of data and tables available under MySQL, offers concrete examples of MySQL database structures, shows how indexes enable more efficient table access, and, finally, offers enlightenment on concrete ways that databases and tables can be generated.

Chapter Overview

Introduction

Database design is doubtless one of the greatest challenges in the development of complex database applications. This chapter attempts to provide a first introduction to the topic. But please don't expect any foolproof recipes. Many of the skills in database design are won by experience, and in a single chapter we can transmit only some of the basic principles.

To help in your orientation in this relatively long chapter, here is an overview:

- The following two sections provide basic information. We discuss how to divide data logically among several tables, where the primary goal is to avoid redundancy. The *normalization* process helps in this.

  We also present the data types belonging to MySQL. For example, we answer the question as to what data type should be used for storing amounts of money and for long and short character strings.

- We then go into some more advanced topics: How can searching and sorting tables be speeded up with the help of indexes? What types of tables does MySQL recognize, and which should be used when?

- How are databases and their tables actually generated? A separate section introduces various *CREATE* commands, which help in this. (However, it is considerably more convenient to create new tables using instead a user interface such as phpMyAdmin.)

- Theory is all well and good, but at some point we have to roll up our sleeves, put on an apron, and start cooking up some databases of our own. Therefore, the last sections of this chapter offer some concrete examples of the design of databases: for managing a small library, for running a discussion group, and for testing special cases.

POINTER *In this chapter we cannot avoid using some SQL commands now and then, although SQL is actually introduced in detail only in the next chapter. (However, it is conversely also impossible to describe SQL without assuming a particular database layout.) You may find it necessary at times to flip back and forth between this chapter and the next.*

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:

1. 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.)

2. Judith S. Bowman et al.: *The Practical SQL Handbook*, Addison-Wesley, 2001.
3. 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.)
4. The MySQL documentation recommends an additional book as an introduction to SQL, though I am not acquainted with it: Martin Gruber's *Mastering SQL*, Sybex, 2000.
5. Another book that is frequently recommended is Peter Gulutzan and Trudy Pelzer's *SQL-99 Complete, Really*, R&D Books, 1999.

POINTER *If you are not quite ready to shell out your hard-earned (or perhaps even ill-gotten) 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`.

Database Theory

Why is that 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.

Normal Forms

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*. Addison-Wesley 2000.
>
> 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.

Our 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 5-1 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 5-1. A book database: first attempt.

title	***publisher***	***year***	***author1***	***author2***	***author3***
Linux	Addison-Wesley	2000	Kofler, M.		
Definitive . . .	Apress	2000	Kofler, M.	Kramer, D.	
Client . . .	Addison-Wesley	1997	Orfali, R.	Harkey, D.	Edwards, E.
Web . . .	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 5-2.

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*, *publisher*, and *year* are repeated for each author. There must be a better way!

Table 5-2. A book database: first normal form

id	*title*	*publisher*	*year*	*author*
1	Linux	Addison-Wesley	2000	Kofler, M.
2	Definitive Guide . . .	Apress	2000	Kofler, M.
3	Definitive Guide . . .	Apress	2000	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.

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 5-2, 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 5-3 and 5-4.

Table 5-3. title *table: second normal form*

titleID	*title*	*publisher*	*year*
1	Linux	Addison-Wesley	2000
2	Definitive Guide . . .	Apress	2000
3	Client/Server . . .	Addison-Wesley	1997
4	Web Application . . .	New Riders	2000

Table 5-4. author *table: second normal form*

authorID	*titleID*	*author*
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 *author* table, the first column, with its running *authorID* values, provides the primary key. The second column takes over the task of the foreign key. It points, or refers, to rows of the *title* table. For example, row 7 of the *author* 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 *author* 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 5-5 through 5-7.

Table 5-5. title *table: second normal form*

titleID	***title***	***publisher***	***year***
1	Linux	Addison-Wesley	2000
2	Definitive Guide . . .	Apress	2000
3	Client/Server . . .	Addison-Wesley	1997
4	Web Application . . .	New Riders	2000

Table 5-6. author *table: second normal form*

authorID	***author***
1	Kofler, M.
2	Kramer, D.
3	Orfali, R.
4	Harkey, D.
5	Edwards, E.
6	Ratschiller, T.
7	Gerken, T.

Table 5-7. rel_title_author *table: second normal form*

titleID	***authorID***
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 then, computers are another matter altogether, being totally without consciousness, regardless of what the nuttier cognitive scientists have to say.

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 *title* 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 *authorID* numbers then lead to the names of the authors.

REMARK *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* authorID *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 *title* 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 5-8 and 5-9.

Table 5-8. title *table: third normal form*

titleID	***title***	***publisherID***	***year***
1	Linux	1	2000
2	Definitive Guide . . .	2	2000
3	Client/Server . . .	1	1997
4	Web Application . . .	3	2000

Table 5-9. publisher *table: third normal form*

publisherID	***publisher***
1	Addison-Wesley
2	Apress
3	New Riders

The *author* and *rel_title_author* tables remain the same in the third normal form. The completed book database now consists of four tables, as indicated in Figure 5-1.

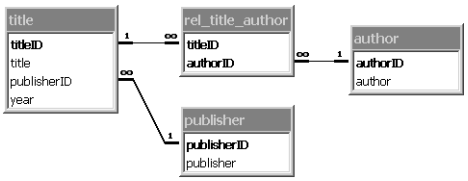

Figure 5-1. The structure of the database books

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.

The Bestowal of Names

In our example, we have attempted to leave the names of fields unaltered from the beginning (before normalization) right through to the end, so as not to cause more confusion than absolutely necessary. The result is not quite optimal, and it would be worth going back and looking at the complete table structure and reconsidering what names should be given to the various fields. The following points provide some tips on the naming of tables and their columns (fields).

- MySQL distinguishes between uppercase and lowercase in the naming of tables, but not in the naming of columns (see also Chapter 18). Thus it is important to pay attention to this case sensitivity at least in names of tables. (In the example databases in this book, we shall use only lowercase letters for the names of tables.)

- Of course, field and table names should be as clear as possible. Field names like *author* and *publisher* are not particularly good role models. First of all, they coincide with table names, which can cause confusion, and second, *name* or *authorsName* or *companyName* would have been more precise.

- A uniform pattern for naming fields will save many errors caused by haste. Whether you prefer *authors_name* or *authorsName*, choose one format and stick with it.

- Likewise, you should consider how to deal with singular and plural forms. In our example, I could have named the tables *title*, *author*, and *publisher* just as easily *titles*, *authors*, and *publishers*. There is no rule as to what is right or wrong here, but it is certainly confusing if half of your tables use the singular, and the other half, the plural.

- Finally, it is worth thinking about whether field names should provide information about the table. In our example there are fields called *titleID*, *publisherID*, and the like. Fields containing a primary key could as well be

called *id*. This would satisfy the uniqueness criterion, since in *SELECT* queries that encompass several tables, the table name has often to be provided in any case. (SQL allows the format *tablename.fieldname*.) This could lead to confusing instructions like *WHERE publisher.publisherID = title.publisherID*.

On the other hand, many tables also contain foreign keys, and there the specification of the table name is unavoidable. (In the *title* table, for example, you could not have three columns all labeled *id*.) That would lead to *publisher.id = title.publisherID*, which is also not optimal. (You can see that there is no easy solution that works optimally for all cases.)

POINTER *From* books *another example database was created called* mylibrary. *The design of that database is described later in this chapter. It is better than the* books *database in the sense of the bestowal of names. The database* mylibrary *serves as the basis for many examples in this book, while* books *exists only for the purpose of introducing a bit of database 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 baby 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 (see the next section).
- A good database design cannot be obtained unless you have had some experience with SQL (see also next chapter). 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 the following Internet location:*

`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 hundred-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.)

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.)

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 later in 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 normalization example of the previous section (book database): Here there is a one-to-many relation between the *title* and *publisher* tables. The table *publisher* is the master table with the primary key *publisher.publisherID*. The title table is the detail table, with the foreign key *title.publisherID* (see Figure 5-1).

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

- 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 web site (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 web site 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).

REMARK *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 it is not required.*

The table diagrams in this book (such as shown in Figure 5-1) were created with Microsoft Access. For all of the technical drawbacks that Access offers on account of the file/server architecture vis-à-vis MySQL, the user interface is easy to use. There is hardly a program with which relation diagrams can be so quickly drawn. However, Access has the peculiarity of labeling $1:n$ *diagrams with* $1:\infty$*. Here* 1 *denotes the master table (primary key), while infinitely many* n *stands for the detail table (foreign key). Often, an arrow is used instead of this nomenclature.*

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:

- A table of personnel, in which each employee record (except for that of the top banana) refers to a field containing that individual's immediate supervisor.
- Discussion groups, with tables with messages, in which 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; see `http://www.mp3.com`).

REMARK *Relations within a table indeed allow for a very simple way to store a hierarchy, but the evaluation of such data is more complicated. For example, the query command* SELECT *does not allow for recursion, which is often necessary for the analysis of hierarchical relations. Consider carefully the types of queries that will be applied to your database before you heedlessly implement complex hierarchies based on internal table references.*

The database mylibrary *introduced later in this chapter provides a concrete example of the representation of hierarchies.*

$n:m$ 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:

- Normalization example (*books* 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.
- Business application, tables 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.
- 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 avoid entirely a bit of an excursus into 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 publisher
  (publisherID INT NOT NULL AUTO_INCREMENT,
  othercolumns ... ,
  PRIMARY KEY (publisherID))
```

If we translate from SQL into English, what we have is this: The field *publisherID* 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 auxiliary index file to enable rapid search. Simultaneously, it is thereby ensured that the *publisherID* value is unique when new records are input (even if a particular *publisherID* is specified in the *INSERT* command).

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).

REMARK *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.

POINTER *To generate sequential numbers for primary keys automatically, in MySQL the column is usually created with the attribute* AUTO_INCREMENT. *Additional information on* AUTO_INCREMENT *can be found in the next section of this chapter.*

Foreign Key

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 title.title, publisher.publisher FROM title, publisher
WHERE title.publisherID = publisher.publisherID
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	*publisher*
Client/Server ...	Addison-Wesley
Definitive Guide ...	Apress
Linux	Addison-Wesley
Web Application ...	New Riders

Decisive here is the clause *WHERE title.publisherID = publisher.publisherID*. It is here that the link between the tables is created.

On the other hand, 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 title
  (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

If you delete the author *Kofler* from the *author* table in the example of the normalized database (see Tables 5-6 through 5-9 and Figure 5-1), you will encounter problems in many SQL queries that access the books *Linux* and *Definitive Guide*. The *authorID* number 1 specified in the *rel_title_author* table no longer exists in the *author* 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 (and databases often must be altered by hand), many databases have rules for maintaining referential integrity. Such 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.

In MySQL 4.0, referential integrity can be automatically ensured only with InnoDB tables. Independent control of referential integrity for the more commonly deployed MyISAM tables is planned for MySQL 5.0.

With InnoDB tables, you can declare foreign key fields like this:

```
CREATE TABLE title
  (column1, column2,  ... ,
   publisherID INT,
   KEY (publisherID),
   FOREIGN KEY (publisherID) REFERENCES publisher (publisherID)
)
```

This means that *title.publisherID* is a foreign key that refers to the primary key *publisher.publisherID*. 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.

Moreover, this instruction will then be correctly executed even if the table driver does not support integrity rules. In this case, however, the *FOREIGN KEY* information is ignored, and currently, not even saved. (At least this latter should change in the future: Even if MySQL does not concern itself with maintaining referential integrity, the corresponding table declarations should be maintained. This would be particularly practical for programs for formulating SQL queries and for graphical user interfaces, which then would know how tables are linked with one another.)

> **POINTER** *In Chapter 8, SQL commands for declaring referential integrity for InnoDB tables will be discussed in detail.*

Other Indexes

> **POINTER** *The primary index, which is inseparably bound to the primary key, is a special case of an index. Indexes always have the task of speeding up searching and sorting in tables by particular columns. Other indexes supported by MySQL are introduced later in the chapter.*

MySQL Data Types

Now that we have dealt with the first issue in database design, dividing the data among several tables in the most sensible way, in this section we go a level deeper. Here you will find information about the data types that MySQL provides for every field (every column). Should a character string stored in a field be of fixed or variable length? Should currency amounts be stored as fixed-point or floating-point numbers? How are binary objects handled?

> **POINTER** *This section describes the data types, but not the* CREATE TABLE *command, with which tables are actually created. Information on how tables can be created, that is, how the database, having been designed, can actually be implemented, can be found later in this chapter.*

Default Values and NULL

Note that for each column, in addition to the data type, you can specify additional attributes. The following list names the three most important of these that are available for all data types:

- *NOT NULL:* The column may not contain the value *NULL*. This means that in saving a new data record an explicit value must be given (unless there is a default value for the column).
- *NULL:* The column is allowed to contain the value *NULL*. This is the default setting if when the table was created, neither *NOT NULL* nor *NULL* was specified.
- *DEFAULT n* or *DEFAULT 'abc'*: If no value is specified for a data record, MySQL automatically provides the value *n* or the character string *'abc'*. (Here is a tip for advanced users: If you would like MySQL to insert the default value for an *INSERT* command, you must specify for the column either no value at all, or an empty character string. It is not permitted to supply the value *NULL*. In that case, MySQL stores *NULL* or returns an error message if *NULL* is not allowed.)

Even if you do not explicitly provide a default value, in many instances, MySQL will itself provide one: *NULL* if *NULL* is allowed, and otherwise, 0 for a numerical column, an empty character string for *VARCHAR*, the date 0000-00-00 in the case of dates, the year 0000 for *YEAR*, or the first element of an *ENUM* enumeration.

REMARK *Unfortunately, MySQL does not allow a function as default value. Thus it is not possible to specify* DEFAULT RAND() *if you want a random number to be entered automatically in a column.*

Integers: xxxINT

TINYINT(M)	8-bit integer (1 byte, -128 to $+127$)
SMALLINT(M)	16- bit integer (2 bytes, $-32{,}769$ to $+32{,}767$)
MEDIUMINT	24- bit integer (3 bytes, $-8{,}388{,}608$ to $+8{,}388{,}607$)
INT, INTEGER(M)	32- bit integer (4 bytes, $-2{,}147{,}483{,}648$ to $+2{,}147{,}483{,}647$)
BIGINT(M)	64- bit integer (8 bytes, $\pm 9.22 \times 10^{18}$)

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.

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.

REMARK *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.

Floating-Point Numbers: FLOAT *and* DOUBLE

FLOAT(M, D)	floating-point number, 8-place precision (4 bytes)
DOUBLE(M, D)	floating-point number, 16-place precision (8 bytes)
REAL(M, D)	Synonym for *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 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.

REMARK *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(P,S)

DECIMAL(p, s)	fixed-point number, saved as a character string; arbitrary number of digits (one byte per digit + 2 bytes overhead)
NUMERIC, DEC	synonym for *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 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, *and* DATETIME

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 $\pm 838:59:59$ (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)

With the *DATE* and *DATETIME* data types, only a limited amount of type checking takes place. Values between 0 and 12 for months, and between 0 and 31 for days, are 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.) As for the question that may have occurred to you—Why is it that *DATETIME* requires eight bytes, while *DATE* and *TIME* each require only three bytes?—the answer is that I have no idea.

MySQL returns results of queries in the form 2003-12-31. 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
2000  September 3
2000  October 25
 ...
```

Time of the Most Recent Change: TIMESTAMP(M)

TIMESTAMP	date and time in the form 20031231235959 for times between 1970 and 2038 (4 bytes)

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.

For the automatic *TIMESTAMP* updating to function properly, either no explicit value is assigned to the field, or else the value *NULL*. In both cases MySQL itself inserts the current time.

In the declaration of a *TIMESTAMP* column, the desired column width is specified. The *TIMESTAMP* values that then result from queries will be truncated. (In the case of $M = 8$, for example, the result will be a date without the time). Internally, however, the values continue to be stored in their complete form.

REMARK *Since version 4.1, MySQL automatically returns* TIMESTAMP *columns in the form* YYYY-MM-DD HH:MM:DD *(instead of the earlier* YYYYMMDDHHMMDD*). This can result in incompatibilities in data processing. Append a zero if you wish to use the old form:* SELECT ts+0 FROM table.

WARNING *Do not use a* TIMESTAMP *column if you wish to store the date and time yourself. For that, one has the data type* DATETIME. *With* TIMESTAMP *columns the MySQL server automatically changes the content of the column every time a change is made. You can prevent this behavior only by explicitly giving the desired date and time with each* UPDATE *or* INSERT *command. (The following example assumes that the* TIMESTAMP *column is called* ts.*)*

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

Sooner or later, you will forget ts=ts *and will then have the incorrect date/time in your table.*

REMARK *Many database operations with particular client libraries (for example, with Connector/ODBC) function only when each table of the database displays a* TIMESTAMP *column. The time of the last update is often needed in the internal administration of data.*

Character Strings

CHAR(n)	character string with specified length, maximum 255 bytes
VARCHAR(n)	character string with variable length, maximum 255 bytes
TINYTEXT	character string with variable length, maximum 255 bytes
TEXT	character string with variable length, maximum $2^{16} - 1 = 65{,}535$ bytes
MEDIUMTEXT	character string with variable length, maximum $2^{24} - 1 = 16{,}777{,}215$ bytes
LONGTEXT	character string with variable length, maximum $2^{32} - 1 = 4{,}294{,}967{,}295$ bytes

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.

REMARK *MySQL 4.0 uses the character set* latin1 *by default for all databases, where there is the simple rule of one byte per character for storage.*

Since MySQL 4.1, on the other hand, each column of a table can have a particular character set assigned to it. This includes character sets for which the storage requirement varies according to the character, for example, Unicode UTF8. This has the consequence that in a VARCHAR(50) *column with the UTF8 character set, you can store fifty ASCII characters (one byte per character), but only twenty-five special German characters (two bytes per character). Many Asian characters require even more bytes. Thus it is no longer possible to specify exactly how many characters can be stored in a column. It depends on the character set used.*

Although *VARCHAR* and *TINYTEXT*, both of which can accept a character string up to a length of 255 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 255) 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.)

- In *VARCHAR* columns, as with *CHAR* columns, blank characters are deleted from the beginning of a character string. (This behavior is ANSI compliant only for *CHAR* columns, not for *VARCHAR* columns. This behavior may change in future versions of MySQL.)

  With *xxxTEXT* columns, character strings are stored exactly as they are input.

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 be useful when you store text (and not binary objects): 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). The internal management of binary character strings is simpler and therefore faster than is the case with garden-variety character strings.

REMARK *As we have already mentioned, MySQL supports Unicode character strings, beginning with version 4.1. Of course, in version 4.0 as well you can store Unicode character strings in* TEXT *as well as* BLOB *fields. MySQL, however, is incapable of sorting such tables correctly, and there can be problems in searching and comparing as well.*

Binary Data (xxxBLOB)

TINYBLOB	binary data, variable length, max 255 bytes
BLOB	binary data, variable length, max $2^{16} - 1$ bytes (64 kilobytes)
MEDIUMBLOB	binary data, variable length, max $2^{24} - 1$ bytes (16 megabytes)
LONGBLOB	binary data, variable length, max $2^{32} - 1$ bytes (4 gigabytes)

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.

REMARK *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). 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.

Enumerations: ENUM, SET

ENUM	selects one from at most 65,535 character strings (1 or 2 bytes)
SET	combines at most 64 character strings (1–8 bytes)

MySQL offers the two special enumeration types *ENUM* and *SET*. With *ENUM*, you can manage a list of up to 65,535 character strings, ordered by a running index. Then, in a field, one of these character strings can be selected.

In queries involving comparison of character strings there is no case distinction. In addition to the predefined character strings, an empty character string can also be stored in a field (as well as *NULL*, unless this has been excluded via *NOT NULL*).

Such a field is then handled like any other character string field. The following commands show how a table with an *ENUM* enumeration is generated and used. In the field *color* of the table *testenum*, one of five predefined colors can be stored:

```
CREATE TABLE testenum
  (color ENUM ('red', 'green', 'blue', 'black', 'white'))
INSERT testenum VALUES ('red')
SELECT * FROM testenum WHERE color='red'
```

SET uses a similar idea, though here arbitrary combinations are possible. Internally, the character strings are ordered by powers of 2 (1, 2, 4, 8, etc.), so that a bitwise combination is possible. The storage requirement is correspondingly larger (one bit per character string). At most 64 character strings can be combined (in which case the storage requirement is 8 bytes).

For a combination of several character strings to be stored in one field, these must be given separated by commas (and with no blank characters between strings). The order of the strings is irrelevant and is not considered. In query results, combinations are always specified in the order in which the set was defined:

```
CREATE TABLE testset
  (fontattr SET ('bold', 'italic', 'underlined'))
INSERT testset VALUES ('bold,italic')
```

In queries with the operator "=" an exact comparison is made of the entire combination. The result is that only those records are returned for which the combination corresponds exactly. Thus if in *testset* only the above-inserted record is stored with *'bold,italic'*, then the following query returns no result:

```
SELECT * FROM testset WHERE fontattr='italic'
```

In order to locate records in which an attribute has been set (regardless of its combination with other attributes), the MySQL function *FIND_IN_SET* can be used. This function returns the position of the sought character string within the set (in our example, 1 if *'bold'* is found, 2 for *'italic'*, etc.):

```
SELECT * FROM testset WHERE FIND_IN_SET('italic', fontattr)>0
```

TIP ENUM *and* SET *values are represented internally as integers, not as character strings. If you wish to determine the internally stored value via a query, simply use* SELECT x+0 FROM table, *where* x *is the column name of the* ENUM *or* SET *column. It is also permitted to store numeric values with* INSERT *and* UPDATE *commands.*

REMARK *The contents of* ENUM *and* SET *fields are not alphabetically sorted, but are maintained in the order in which the character strings for selection were defined. The reason for this is that MySQL works internally with numeric values associated with the character strings. If you would like an alphabetic sorting, you must transform the string explicitly into a character string, for example, via* SELECT CONCAT(x) AS xstr . . . ORDER BY xstr.

TIP *If you would like to determine the list of all admissible character strings for an* ENUM *or* SET *field (in a client program, for example), you must summon* DESCRIBE tablename columnname *to your aid. This SQL command returns a table in which the field* columnname *is described. The column* Type *of this table contains the* ENUM *or* SET *definition. In Chapter 11 you will find an example for the evaluation of this information (in the programming language PHP).*

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.)

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'*).

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. (Internally, in addition, B trees are used for managing the index entries.)

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.)

Types of Index

All indexes are not alike, and in this section we discuss the various types of indexes.

Ordinary Index

The only task of an ordinary index (defined via the key word *INDEX*) is to speed up access to data.

> **TIP** *Index only those columns that you require in conditions (*WHERE column= . . . *) or for sorting (*ORDER BY column*). Index, if possible, columns with compact data (for example, integers). Do not index columns that contain many identical values. (For example, it makes little sense to index a column with 0/1 or Y/N values.)*

Restrictions

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.

Finally, 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.)

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

The primary index mentioned again and again in this chapter is nothing more than an ordinary *UNIQUE* index. The only peculiarity is that the index has the name *PRIMARY*.

(You can tell that a primary index is an ordinary index because MySQL uses the first *UNIQUE* index of a table as a substitute for the missing primary index.)

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*.

POINTER *An extensive description of the SQL syntax for full-text search, together with a host of application examples, can be found in Chapter 7.*

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 three queries, 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 the *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. Further information on all *JOIN* types can by found in the MySQL documentation.)

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.

POINTER *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://www.mysql.com/doc/en/Query_Speed.html
http://www.mysql.com/doc/en/EXPLAIN.html
```

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	possible_key	key_len	ref	rows
titles	ALL	PRIMARY			23
rel_title_author	ref	PRIMARY PRIMARY	4	titles.titleID	1
authors	eq_ref	PRIMARY PRIMARY	4	rel_title_author.authID	1

This means that first all records from the *titles* table are read, without using an index. For *titles.titleID* there is, in fact, an index, but since the query considers all records in any case, the index is not used. Then with the help of the primary indexes of *rel_title_author* and *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.

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=1
  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	2	Using where; Using filesort
rel_title_author	ref	PRIMARY	4	titles.titleID	1	Using index
authors	eq_ref	PRIMARY	4	rel_title_author.authID	1	

To save space in this example, we have not shown the column *possible_keys*. 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 36 records of the *titles* table must be read, and no index can be used:

table	type	key	key_len	ref	rows	Extra
titles	ALL				36	Using where; Using filesort
rel_title_author	ref	PRIMARY	4	titles.titleID	1	Using index
authors	eq_ref	PRIMARY	4	rel_title_author.authID	1	

MySQL Table Types

Up till now, we have tacitly assumed that all tables have been set up as MyISAM tables. That has been the default behavior since MySQL 3.23, and it holds automatically if you do not explicitly demand a different type of table when the table is generated. This section provides a brief overview of the different types of table recognized by MySQL, their properties, and when they should be used. (The MySQL server can, however, be configured using the option `default-table-type` to make another table type valid.)

- **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 10).

- **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).

- **ISAM tables:** The ISAM table type is the precursor of MyISAM, and is supported only for reasons of compatibility. ISAM tables exhibit a host of disadvantages with respect to MyISAM tables. Perhaps the most significant of these is that the resulting files are not independent of the operating system. Thus in general, an ISAM table that was generated by MySQL under operating system x cannot be used under operating system y. (Instead, you must use `mysqldump` and `mysql` to transfer it to the new operating system.) In general, there is no longer any good reason to use ISAM tables (unless, perhaps, you are using an antiquated version of MySQL).

- **InnoDB tables:** The InnoDB table type is the most important alternative to the MyISAM tables. The use of such tables involves a number of different advantages and disadvantages.

  - Advantages vis-à-vis MyISAM: InnoDB tables support transactions, which ensures the integrity of multistep database operations. InnoDB tables also support functions for ensuring referential integrity.
  - Disadvantages vis-à-vis MyISAM: All InnoDB tables are stored in a single binary file, which is thus to some extent a black box. To move a single table or database from one server to another requires the use of `mysqldump`. (With MyISAM tables you can simply copy and transfer the file.) Moreover, InnoDB tables do not yet support a full-text index. The storage requirements on the hard disk are greater than those of comparable MyISAM tables.

    Elementary database operations are generally slower with InnoDB tables than with MyISAM tables. InnoDB tables can, however, greatly improve their performance speed if through the use of transactions one can do without *LOCK* commands.

  In general, many InnoDB functions are similar to those in Oracle tables. The InnoDB table type has been developed over a number of years and is used in a large number of database systems. You will find extensive information on InnoDB in Chapter 8.

- **BDB (Berkeley_DB):** BDB tables also support transactions. (It was the BDB table type that first made transactions possible under MySQL.) In comparison to InnoDB, the integration into MySQL is not as good, and the documentation in a number of details is rather vague. For these reasons, the BDB table driver is not integrated into the standard version of the MySQL server. As a rule, it is advisable to use InnoDB tables in preference to BDB tables.

- **Gemini:** In 2001, the Nusphere company made available its own version of MySQL, distinguished by the addition of the Gemini table type, which supports transactions. Unfortunately, a legal battle developed between Nusphere and the MySQL company, a battle over whether Nusphere had adhered to the rules of the MySQL license (GPL). The disagreement was settled in November 2002, with the result that the code for the Gemini table type was not added to the official MySQL code.
- **MERGE:** MERGE tables are essentially a virtual union of several existing MyISAM tables all of which exhibit identical column definitions. A MERGE table composed of several tables can have some advantages over a single, large, MyISAM table, such as a higher read speed (if the tables are distributed over several hard drives) or a circumvention of the maximum file size in a number of older operating systems (for example, 2 gigabytes for Linux; 2.2 for 32-bit processors). Among the disadvantages are that it is impossible to insert data records into MERGE tables (that is, *INSERT* does not function). Instead, *INSERT* must be applied to one of the subtables.

  In the meanwhile, since most modern operating systems support files of arbitrary size, as well as RAID (that is, the division of a file system on several hard disks), MERGE tables play a subordinate role in practice.
- **HEAP:** 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.

  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 <, >, <=, >=). *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 by the parameter `max_heap_table_size`, which can be set when MySQL is launched.

REMARK *From among the table types listed above, only the most important will be discussed in this book: MyISAM, InnoDB, and HEAP. Further information on the other table types can be found in the MySQL documentation:*

`http://www.mysql.com/doc/en/Table_types.html`

Temporary Tables

With most 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 launch.

> **REMARK** *Often, temporary tables are generated that are of type* HEAP. *However, this is not a necessary combination. Note, however, that nontemporary* HEAP *tables are visible to all MySQL users until they are deleted.*

Table Files

The location of database files can be set when MySQL is launched. (Under Unix/Linux, `/var/lib/mysql` is frequently used, while under Windows it is usually `C:\mysql\data`.) All further input is given relative to this directory.

Where MySQL tables are actually stored depends very much on the table type. Basically, the description of each table is stored in a `*.frm` file, and these `*.frm` files are organized in directories that correspond to the names of the databases:

```
data/dbname/tablename.frm
```

for the table structure (data type of the columns, indexes, etc.).

For each MyISAM table, two additional files are created:

```
data/dbname/table name.MYD
```

for the MyISAM table data, and

```
data/dbname/tablename.MYI
```

for the MyISAM table indexes (all the indexes of the table).

InnoDB tables, on the other hand, are stored together in a single file or in several associated files. These files form the so-called *tablespace*, a sort of virtual storage for all tables. The name and location of these files can be prescribed by configuration settings. By default,

```
data/ibdata1, 2, 3
```

hold the InnoDB tables (data and indexes), while

```
data/ib_logfile0, 1, 2
```

hold the InnodDB logging files.

Starting with MySQL 4.1, an additional file, `db.opt`, is stored in the database directory. It contains settings that hold for the entire database:

```
data/dbname/db.opt
```

Creating Databases, Tables, and Indexes

In the course of this chapter you have learned a great deal about database design. But building castles in the air is one thing, and assembling the plans and materials for real-world construction quite another. We now confront the question of how a database and its tables are actually built. As always, there are several ways of going about this task:

- Genuine SQL pros will not shrink before the prospect of doing things the old-fashioned way and typing in a host of *CREATE TABLE* commands in `mysql`. However, the complicated syntax of this command is not compatible with the creative character of the design process. Nonetheless, SQL commands are often the only way to achieve certain features of tables.
- For beginners in the wonderful world of database construction (and for all who do not enjoy making their lives more complicated than necessary), the graphical user interfaces offer more convenience in the development of a database. Some of these programs (for example, the MySQL Control Center and phpMyAdmin) have been introduced in the previous chapter.

In this chapter, only the *CREATE TABLE* variant is considered. All of our examples refer to the little book database *books*, which in the portion of this chapter dealing with theory was introduced for the purpose of demonstrating the normalization process.

REMARK *Regardless of the tool that you choose to work with, before you begin, the question of access rights must be clarified. If MySQL is securely configured and you do not happen to be the MySQL administrator, then you are not permitted to create a new database at all.*

The topics of access privileges and security are not dealt with until Chapter 9. If you are responsible for the security of MySQL, then you should probably concern yourself with this issue before you create your new database. If there is an administrator, then please ask him or her to set up an empty database for you and provide you with a user name and password.

POINTER *This section assumes that the new database will be created on the local computer, which is plausible if one is creating a new application. What happens, then, when a web application will run on the Internet service provider's computer? How do you transport the database from the local computer to that of the ISP? How can you, should the occasion arise, create a new database directly on the ISP's computer? Such questions will be discussed in their own section of Chapter 10.*

POINTER *There is yet another path to a MySQL database: the conversion of an existing database from another system. Fortunately, there are quite a few converters available, and they can be found at the MySQL web site, which at last sighting was at the following addresses:*

```
http://www.mysql.com/doc/en/Contrib.html
http://www.mysql.com/portal/software/index.html
```

In this book only two variants will be considered, namely, conversion of databases in the format of Microsoft Access or of Microsoft SQL Server. This topic is dealt with in Chapter 17.

*Creating a Database (*CREATE DATABASE*)*

Before you can create tables for your database, you must first create the database itself. The creation of a database results in an empty directory being created on the hard drive. The following *USE* command makes the new database the default database for all further SQL commands:

```
CREATE DATABASE books
USE books
```

REMARK *The commands given here can be executed, for example, in the MySQL monitor* `mysql`*. (Please recall that in* `mysql` *you must follow each command with a semicolon.)*

A complete syntax reference for the SQL commands introduced here can be found in Chapter 18.

*Creating Tables (*CREATE TABLE*)*

For the four tables of our example database, we require four *CREATE TABLE* commands. Each of these commands contains a list of all fields (columns) with the associated data type and attributes. The syntax should be immediately clear. The definition of the primary index is effected with *PRIMARY KEY*, where all columns that are to be included in the index are specified in parentheses. Of interest here is the table *rel_title_author*, in which the primary index spans both columns:

```
CREATE TABLE author (
   authorID      INT             NOT NULL    AUTO_INCREMENT,
   author        VARCHAR(60)     NOT NULL,
   PRIMARY KEY   (authorID))
CREATE TABLE publisher (
   publisherID   INT             NOT NULL    AUTO_INCREMENT,
   publisher     VARCHAR(60)     NOT NULL,
   PRIMARY KEY   (publisherID))
```

```
CREATE TABLE rel_title_author (
   titleID       INT                    NOT NULL,
   authorID      INT                    NOT NULL,
   PRIMARY KEY   (titleID, authorID))
CREATE TABLE title (
   titleID       INT                    NOT NULL    AUTO_INCREMENT,
   title         VARCHAR(120)           NOT NULL,
   publisherID   INT                    NOT NULL,
   year          INT                    NOT NULL,
   PRIMARY KEY   (titleID))
```

*Editing a Table Design (*ALTER TABLE*)*

It is fortunate indeed that you do not have to recreate your database from scratch every time you wish to modify its design, say to add a new column or change the data type or default value of a column. The command *ALTER TABLE* offers sufficient flexibility in most cases. It is also good to know that MySQL is capable in most cases of preserving existing data (even if the data type of a column is changed). Nevertheless, it is a good idea to maintain a backup copy of the table in question.

The following example shows how the maximal number of characters in the *title* column of the *title* table is reduced to 100 characters. (Titles longer than 100 characters will be truncated.) The first line of code shows how to make a backup of the original table (which is a good idea, since we might lose some data if titles are truncated):

```
CREATE TABLE titlebackup SELECT * FROM title
ALTER TABLE title CHANGE title title VARCHAR(100)
NOT NULL
```

Something that might be a bit confusing with our use of the *ALTER TABLE* command is the threefold appearance of the name *title*. The first usage refers to the table, while the second is the name of the column before the change, and the third the new, in this case unchanged, name of the column.

*Inserting Data (*INSERT*)*

Normally, data are inserted into the database via a client program (which, it is to be hoped, is convenient and easy to use). Since you, as developer, are responsible in most cases for the code of these programs, it would not hurt for us to take a peek at the *INSERT* command. (Additional commands for inserting, changing, and deleting data can be found in the next chapter.)

```
INSERT INTO author VALUES ( '1', 'Kofler M.')
INSERT INTO author VALUES ('2', 'Kramer D.')
 ...
INSERT INTO publisher VALUES ('1', 'Addison-Wesley')
INSERT INTO publisher VALUES ('2', 'Apress')
 ...
INSERT INTO title VALUES ('1', 'Linux', '1', '2000')
INSERT INTO title VALUES ('2', 'Definitive Guide to Excel VBA', '2', '2000')
```

```
 ...
INSERT INTO rel_title_author VALUES ('1', '1')
INSERT INTO rel_title_author VALUES ('2', '1')
INSERT INTO rel_title_author VALUES ('2', '2')
 ...
```

You can also edit several records at once:

```
INSERT INTO rel_title_author VALUES ('1', '1'), ('2', '1'), ('2', '2'), ...
```

*Creating Indexes (*CREATE INDEX*)*

Indexes can be created via the *CREATE TABLE* command or else later with an *ALTER TABLE* or *CREATE INDEX* command. The syntax for describing the index is the same in all three cases. The following three commands show three variants for providing the *title* column of the *title* table with an index. The index received the name *idxtitle*. (Of course, only one of the three commands can be executed. If you try a second command, MySQL will inform you that an index already exists.)

```
CREATE TABLE title (
  titleID ... , title ... , publisherID ... , year ... , PRIMARY KEY ... ,
  INDEX idxtitle (title))
CREATE INDEX idxtitle ON title (title)
ALTER TABLE title ADD INDEX idxtitle (title)
```

SHOW INDEX FROM tablename produces a list of all defined indexes. Existing indexes can be eliminated with *DROP INDEX indexname ON tablename*.

If you wish to reduce the number of significant characters per index in the index to the first 16 characters, the syntax looks like this:

```
ALTER TABLE title ADD INDEX idxtitle (title(16))
```

Example *mylibrary* (Library)

Writers of books frequently not only write books, but read them as well (as do many other individuals who do not write books), and over time, many of these readers acquire a significant collection of books, sometimes so many that they lose track of what's what and what's where. After a hopeless effort at bringing order to the author's domestic library, he created the database *mylibrary*. See Figure 5-2 for an overview of the structure of the *mylibrary* database. The database *mylibrary* is an extended version of the database *books* that was introduced at the beginning of this chapter. Note that the names of individual tables and columns have been changed in the migration from *books* to *library*.

The main purpose of this database is to store the titles of books together with their authors and publishers in a convenient format. The database will additionally offer the convenient feature of allowing books to be ordered in various categories.

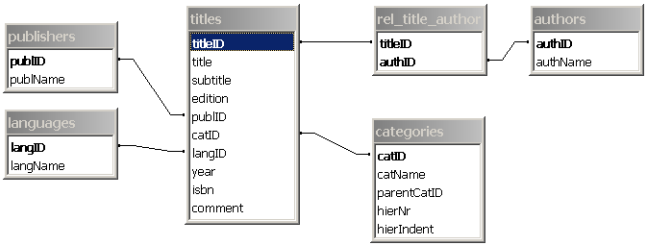

Figure 5-2. Structure of the mylibrary *database*

TIP *If you happen to be a victim of the compulsion to collect something, you can use* mylibrary *as a basis for your own database. You should have little trouble in converting this database into one for CDs, MP3 files, stamps, scientific articles, postal and e-mail addresses, Barbie dolls, birthdays and other noteworthy dates, arachnids, and so on.*

REMARK *The database* mylibrary *as well as its variants* mylibraryodbc *and* mylibraryinno, *are available at* `www.apress.com`.

For mylibararyodbc, *all tables are equipped with an additional* timestamp *column. This column is necessary if the database is to be altered with ODBC programs (see Chapter 17).*

For mylibraryinno, *all tables have been transformed into the InnoDB format. Furthermore, integrity rules (*foreign key constraints *have been defined.*

Basic Design

Titles, Authors, and Publishers

Just as in the *books* database, here there are three tables in which book titles, names of authors, and names of publishers are stored: *titles*, *authors*, and *publishers*.

The $n : m$ relation between *titles* and *authors* is created via the table *rel_title_author*. Here the sequence of authors for a given book can be specified in the field *authNr* (that is, the first, second, third, etc., author). This is practical above all if the database is to store the names of scientific publications, where it is not always a good idea to list the authors in alphabetical order. However, *authNr* can be left empty (that is, *NULL*) if you are not interested in such supplemental information.

In general, in the *library* database it is permitted to store the value *NULL* in many of the columns. Thus it is possible to store a book title quickly without bothering to specify the publisher or year of publication. Perfectionists in database design would, of course, frown on such practice, but on the other hand, too many restrictions often lead to the result that data simply do not get catalogued. From this point of view, it is practical to allow the publisher, say, to be indicated for some books without it being mandatory that it be given for all.

Categories and Languages

The table *categories* enables the construction of a hierarchical category list in which the books can be ordered. (Details on this table appear in the following section.) The table *languages* contains a list with the languages in which the books have appeared. Thus for each title, the language can be stored as well.

Dealing with (Author) Names

In *mylibrary*, names are generally stored in a field where the surname must be given first (to take a random example, *Kofler Michael*, and not *Michael Kofler*). This is the usual ordering for sorting by name.

A quite different approach would be to save surnames, given names, and even initials and titles, in separate fields (for example, *firstname*, *lastname*, and perhaps even *middlename*, etc.). Several fields are, of course, more trouble to manage than a single field, but in return, a greater flexibility is achieved as to how the name can be displayed. This is important, for example, if the data are to be used for personalizing letters or e-mail messages.

Independent of the internal storage, there is clearly a problem in that many people (myself included, alas) now and again input the surname and given name in the wrong order. One falsely input name will be practically untraceable in the database.

Therefore, it is important that the user interface always show and receive names in a uniform order and that the desired order—best indicated by an example—be shown in the input form. Be sure to label the input fields in a clear and unambiguous way (for example, use “family name” or “surname” and not “last name”).

Data Types

All tables are provided with an *AUTO_INCREMENT* column, which also serves as the primary index. Most fields possess the attribute *NOT NULL*. Exceptions are those fields that describe the properties of the title. Thus *NULL* is allowed for *subtitle*, *edition*, *publID*, *catID*, *langID*, *year*, *isbn*, and *comment*.

Column name	Data type
All ID fields	*INT*
authors.authName	*VARCHAR(60)*
categories.catName	*VARCHAR(60)*
categories.hierNr	*INT*
categories.hierIndent	*TINYINT*
languages.langName	*VARCHAR(40)*
publishers.publName	*VARCHAR(60)*
titles.title and *.subtitle*	*VARCHAR(100)*
titles.year	*INT*
titles.edition	*TINYINT*
titles.isbn	*VARCHAR(20)*
titles.comment	*VARCHAR(255)*

Hierarchical Categorization of the Books

The table *categories* helps in ordering books into various categories (technical books, children's books, etc.). To make the *library* more interesting from the point of view of database design, the field *parentCatID* will be represented hierarchically. Here is an example: Figure 5-3 shows how the hierarchy represented in Table 5-10 can be depicted with respect to the database.

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

Figure 5-3. Example data for the categories *table*

Table 5-10. Database representation of the hierarchy in Figure 5-3

catID	***catName***	***parentCatID***
1	Computer books	11
2	Databases	1
3	Programming	1
4	Relational databases	2
5	Object-oriented databases	2
6	PHP	3
7	Perl	3
8	SQL	2
9	Children's books	11
10	Literature and fiction	11
11	All books	NULL

REMARK *In the development of applications,* parentCatID=NULL *must be treated as a special case.*

Hierarchy Problems

Let me state at once that although the representation of such hierarchies looks simple and elegant at first glance, they cause many problems as well. (That this example nonetheless uses hierarchies has, of course, pedagogical purposes: It allows us to demonstrate interesting SQL programming techniques. Even though the representation of hierarchies in this example is somewhat artificial, there are many database applications where the use of hierarchies is unavoidable.)

> **POINTER** *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 often 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).

Optimization (Efficiency Versus Normalization)

If you have taken to heart the subject of normalization of databases as presented in this chapter, 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 *mylibrary* database contains many examples of this, which we shall discuss here rather fully.

This section 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?

Authors' Names in the titles *Table*

The authors' names are stored in the *authors* table in normalized form. However, it will often happen that you wish to obtain a list of book titles, and with each book title you would like to list all of that book's authors, such as in the following form:

Client/Server Survival Guide by Robert Orfali, Dan Harkey, Jeri Edwards
Definitive Guide to Excel VBA by Michael Kofler, David Kramer
Linux by Michael Kofler
Web Application Development with PHP 4.0 by Tobias Ratschiller, Till Gerken

There are now two possible ways of proceeding:

- **Variant 1:** The simplest and thus most often chosen route is first to execute an SQL query to determine the desired list of titles, without the corresponding authors. Then for each title found, a further SQL query is formulated to determine all the authors of that title. Now, however, twenty-one SQL queries were necessary to obtain a list with twenty book titles. The following lines show the requisite pseudocode. The bits in pointy brackets cannot be executed in

SQL; they must be formulated in a client programming language (Perl, PHP, etc.).

```
SELECT titleID, title FROM titles WHERE ... ⟨for each titleID⟩
  SELECT authName FROM authors, rel_title_author
  WHERE rel_title_author.authID = authors.authID
  AND rel_title_author.titleID = ⟨titleID⟩
  ORDER BY authNR
```

- **Variant 2:** Another possibility is to include all the data in a single query:

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

You thereby obtain, however, a list in which each book title appears as many times as it has authors (see Table 5-11).

Table 5-11. Query result in the mylibrary *database*

titleID	***title***	***authName***
3	Client/Server Survival Guide	Edwards E.
3	Client/Server Survival Guide	Harkey D.
3	Client/Server Survival Guide	Orfali R.
2	Definitive Guide to Excel VBA	Kofler Michael
2	Definitive Guide to Excel VBA	Kramer David
1	Linux, 5th ed.	Kofler Michael
9	MySQL & mSQL	King T.
9	MySQL & mSQL	Reese G.
9	MySQL & mSQL	Yarger R.J.
4	Web Application Development with PHP 4.0	Gerken Till
4	Web Application Development with PHP 4.0	Ratschiller Tobias

At this point, you can use some client-side code to create the list of books by collecting authors' names for books with the same *titleID* number and displaying the book title only once.

The significant disadvantage of this modus operandi is that it is relatively difficult to represent the results in page format (for example, with twenty titles per page). Of course, you can limit the number of result records in the *SELECT* query with *LIMIT*, but you don't know in advance how many records will be required to obtain twenty titles (since it depends on the number of authors per title).

This problem can be circumvented if first a query is executed by which only the *titleID* values are determined. The second query then uses these *titleID* values. Unfortunately, MySQL 4.0 does not yet permit nested *SELECT* queries, and therefore either the *titleID* list must be processed by the client program in the

second query (*WHERE titleID IN(x, y, . . .*), or a temporary table must be used. Beginning with MySQL 4.1, such queries can be formulated as sub*SELECT*s.

- **Solution:** From the standpoint of efficiency, the second variant is significantly better than the first, but even it is not wholly satisfying. The most efficient solution would be to provide a column *authorsList* in the *titles* table, in which the list of authors is stored for each title.

  Such a column is no substitute for the *authors* table, which will be used and reused (for example, to search for all books on which a particular author has worked, or to store additional information about an author). The column *authorsList* must be updated each time a new title is stored or an old one is changed. It is much more burdensome that now title records must also be updated when changes in the *authors* table are made (such as when a typo in an author's name is corrected): a classical redundancy problem.

  You thus have the choice between redundancy and efficiency. Which variant is better for your application depends on the size of the database, whether it will be used primarily in read-only mode for queries or frequently altered, what type of queries will be used most frequently, etc. (In our realization of *mylibrary*, we have done without *authorsList*.)

Hierarchical Order in the categories *Table*

The necessity will continually arise to display the *categories* table in hierarchical representation similar to that of Figure 5-3. As we have already mentioned, such processing of the data is connected either with countless SQL queries or complex client-side code. Both of these are unacceptable if the hierarchical representation is needed frequently (for example, for a listbox in a form).

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 5-12 are displayed for both of these columns the values corresponding to the representation in Figure 5-3.

Table 5-12. categories *table with* 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
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
FROM categories ORDER BY hierNr
```

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

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*.

2. (a) 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*). See Figure 5-3. 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.

(b) 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.

(c) 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 description of this algorithm proves that inserting a data record is a relatively complex and costly process. (A concrete realization of the algorithm can be found in the form of a PHP script in Chapter 12.) It is much 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.

Nevertheless, the advantages probably outweigh the drawbacks, above all because changes in the category list will be executed very infrequently (and thus the expenditure of time becomes insignificant), while queries to determine the hierarchical order, on the other hand, must be executed frequently.

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. Since MySQL, alas, does not support sub*SELECT*s, for the second step a temporary table will be necessary.

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 5-13). 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 5-13. Some entries in the rel_cat_parent *table*

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, we decided to do without *rel_cat_parent*, because the associated PHP example (see Chapter 12) in any case does not plan for category searches.

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
```

The result is seen in Figure 5-4.

```
category
All books
  Computer books
    Programming
```

Figure 5-4. Query result in search for higher-ranking categories

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 0. This process is demonstrated in the PHP example program `categories.php` (see Chapter 12).

Indexes

All the *mylibrary* tables are supplied with a *PRIMARY* index for the *ID* column. Additionally, the following usual indexes are defined:

Table	Column	Purpose
authors	*author*	search/sort by author's name
categories	*catName*	search by category name
	hierNr	hierarchical sort of the category list
publishers	*publName*	search/sort by publisher's name
titles	*title*	search/sort by book title
	publID	search by titles from a particular publisher
	catID	search by titles from a particular category
	isbn	search by ISBN

Example *myforum* (Discussion Group)

Among the best-loved MySQL applications are guest books, discussion groups, and other web sites that offer users the possibility of creating a text and thereby adding their own voices to the web site. The database *myforum* creates 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, German, Zemblan, etc.).
- *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 e-mail address are stored. (One can imagine extending this to hold additional information.)
- *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.

The structure of the *myforum* database is depicted in Figure 5-5.

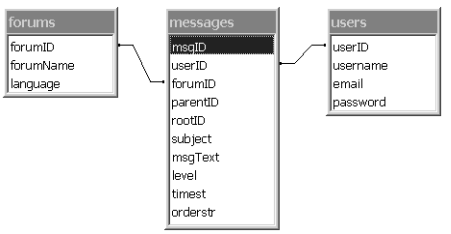

Figure 5-5. Structure of the myforum *database*

REMARK *Included in the code files for this book are two versions of the* myforum *database. The file* `myforum.sql` *contains the SQL instructions for creating a test version of* myforum *that contains only test contributions.*

In contrast, `bigforum.sql` *contains a larger version (18 forums, 1400 contributions) that is the result of the discussion group on my web site. For this book, contributions have been made anonymous. (The* users *table contains only first names.) The database* bigforum *offers the possibility of trying out various MySQL operations (such as full-text search) on a database that is more than just a toy example.*

The bigforum *database contains messages in both English and German. However, the majority are in German. If you want to see only the English entries, you can build your queries thus:*

```
SELECT * FROM messages, forums
WHERE messages.forumID=forums.forumID AND forums.language="english"
```

Data Types

All three tables are equipped with *AUTO_INCREMENT* columns, which also serve as primary index. In general, all fields have the attribute *NOT NULL*. Here we list the column names and data types of the three tables:

Column name	Data type
All ID fields	*INT*
forums.forumname	*VARCHAR(80)*
forums.language	*ENUM('german', 'english', 'zemblan')*
messages.subject	*VARCHAR(80)*
messages.msgText	*TEXT*
messages.level	*INT*
messages.timest	*TIMESTAMP*
messages.orderstr	*VARCHAR(128), BINARY*
users.username	*VARCHAR(30)*
users.email	*VARCHAR(120)*
users.password	*VARCHAR(30)*

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 Figure 5-6.

A: How do I sort MySQL tables? (17.1.2003 12:00)
B: first answer (1.17.2003 18:30)
A: thanks (1.17.2003 19:45)
C: better suggestion (1.19.2003 10:30)
D: second answer (1.18.2003 3:45)
A: I don't understand (1.18.2003 9:45)
D: the same explanation, but more extensive (1.18.2003 22:05)
E: third answer (1.18.2003 19:00)

Figure 5-6. A discussion thread

Here as illustration a description of the content is added to the title line (*subject*) of each contribution. In reality, the title line of a response is usually simply the letters *Re:* together with the original title line (e.g., *Re: How do I . . .*).

A representation in database format might look like that shown in Table 5-14. 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 5-14. messages *table with records of a discussion thread*

msgID	***userID***	***subject***	***parentID***	***timest***
301	201	How do I . . .	0	2001-01-17 12:00
302	202	first answer	301	2001-01-17 18:30
303	201	thanks	302	2001-01-17 19:45
304	204	second answer	301	2001-01-18 3:45
305	201	I don't . . .	304	2001-01-18 9:45
306	205	third answer	301	2001-01-18 19:00
307	204	the same . . .	305	2001-01-18 22:05
308	203	better suggestion	302	2001-01-19 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 0, then the contribution is one that began a new thread.

Optimization

From the standpoint of normalization, *parentID* is sufficient to express the hierarchy. However, this slick table design makes some frequently occurring queries difficult to answer:

- What are all the messages in this discussion thread?
- How should the messages be sorted? (A glance at Table 5-14 shows that a chronological sorting is not our goal.)

- How far should each message be indented if the messages are to be represented hierarchically?

To answer the first question, it is necessary to search (in a recursive loop) through all *parentID*/*msgID* pairs. With long threads, this will require a large number of *SELECT* queries. Of course, each of these queries can be quickly executed, but taken together, they represent a considerable degree of inefficiency.

Recursion also comes into play in the second question: A sorting by the time stored in *timest* leads to our goal, but the answers must be sorted by group for each branch of the hierarchical tree and inserted at the correct place in the list.

The answer to the third question is achieved by following the *parentID* pointer back to the start of the discussion thread.

To increase efficiency, we must back off from our wishful goal of zero redundancy. We shall add three new columns:

- *rootID* refers to the original message that begins the discussion thread and thus makes it possible to answer the first question easily.
- *orderstr* contains a binary concatenation of the *timest* times of all higher-ranking messages. As we shall soon see, this character string can be used for sorting messages within a thread.
- *level* specifies the level of the hierarchy and thus immediately answers the third question.

Having absorbed these new columns into our database, we have a new representation of our database, as depicted in Table 5-15. In comparison with Table 5-14, you can see in the column *timest* that now there are only three digits, which should correspond to the internal binary representation. (In fact, four bytes are used, which permits 256 different values. For greater clarity we have used three-digit numbers here.)

While the meaning of the contents of the columns *rootID* and *level* is clear, that of *orderstr* might be a bit more opaque. In this binary column is stored a binary sequence of *TIMESTAMP* values of all higher-ranking messages. This simple measure ensures that associated groups of messages are not separated during sorting.

Table 5-15. messages *table with records of a discussion thread*

msgID	***userID***	***subject***	***parentID***	***timest***	***rootID***	***level***	***orderstr***
301	201	How . . .	0	712	0	0	712
302	202	first . . .	301	718	301	1	712718
303	201	thanks	302	719	301	2	712718719
304	204	second . . .	301	803	301	1	712803
305	201	I don't . . .	304	809	301	2	712803809
306	205	third . . .	301	819	301	1	712819
307	204	the same . . .	305	822	301	3	712803809822
308	203	better . . .	302	910	301	2	712718910

Pluses and Minuses

The obvious advantage of the three additional columns is the increased efficiency in displaying the messages. Since the reading operation occurs much more frequently than the writing operation in a discussion forum, reading efficiency has a correspondingly high status in the design of our database. (Furthermore, saving a new data record is efficient and not at all problematic. The values for the three supplementary columns can be easily determined for a response from the associated higher-ranking message. If the message begins a new thread, then *rootID* and *level* are 0, while *orderstr* contains the binary representation of the current time.)

The single drawback is the additional storage space required for the three columns. However, in comparison to the typical storage requirement of a message with several lines of text, this additional storage requirement can be considered negligible. (The maximum length of *orderstr* is limited to 128 characters. That is, at most 32 hierarchical levels can be represented.)

Due to the nature of discussion groups, the additional disadvantages due to redundancy, that is, the problems of subsequent changes in data, are insignificant. Changes in existing messages are not anticipated, and neither are additions and deletions after the fact, which would have the effect of changing the sequence of messages.

> **REMARK** *Please note that the* orderstr *character string is based on* TIMESTAMP *values, which change only once per second. If it should chance that two messages are received at the same second, the sort order could become confused.*

Indexes

All *myforum* tables are supplied with a *PRIMARY* index for the *ID* column. Furthermore, the following usual indexes are defined:

Table	Column	Purpose
users	*userName*	search for user name (at login)
messages	*orderStr*	sort messages in discussion order
	forumID	search for messages in a discussion group
	rootID	search for messages in a thread

The PHP example for *myforum* does not provide for searching for messages. For such a search, a full-text index covering the columns *subject* and *msgText* is recommended.

Example *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.

`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*

`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*

`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*.

Contents: 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.

`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 *2000-12-07, 09:06:29, 2000-12-07 09:06:29*, and *20001207090649*.

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

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'

test_order_by

With this table you can test *ORDER BY* and the character set in use (including sort order). The table consists of two columns: *id* and *a_char*. The column *id* contains sequential numbers from 1 to 255, while *a_char* contains the associated character code. An index was deliberately not defined for the column *a_char*.

Content:

id	*a_char*
...	
65	'A'
66	'B'
...	

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 10.