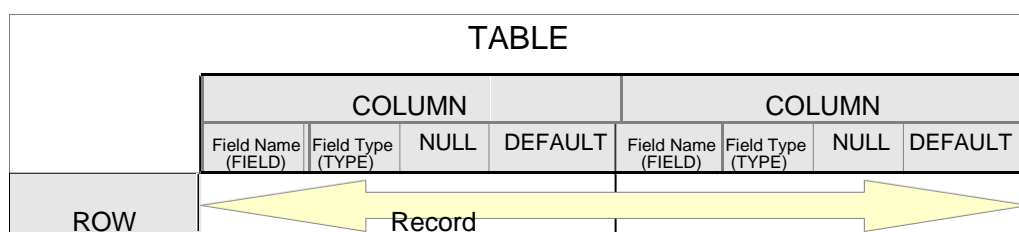# Base

Working with Tables

LibreOffice

Learning Contents

- Creating tables using GUI

- Creating tables using SQL commands

- Working with Relationships

- Entering data into tables

Creating tables

Most LibreOffice users will generally use the graphical user interface (GUI) exclusively to create tables. Direct entry of SQL commands becomes necessary when, for example, a field must subsequently be inserted at a particular position, or a standard value must be set after the table has been saved.

Table terminology: The picture below shows the standard division of tables into columns and rows.

TABLE							
	COLUMN				COLUMN		
	Field Name (FIELD)	Field Type (TYPE)	NULL	DEFAULT	Field Name (FIELD)	Field Type (TYPE)	NULL	DEFAULT
ROW	Record						

Data records are stored in a single row of the table. Individual columns are largely defined by the field, the type, and the rules that determine if the field can be empty. According to the type, the size of the field in characters can also be determined. In addition, a default value can be specified to be used when nothing was entered into the field.

In the Base GUI, the terms for a column are described somewhat differently, as shown below.

Notations of the Base-GUI			
		COLUMN	
		Field properties	
Field Name (FIELD)	Field Type (TYPE)	Entry required (NULL/NOT NULL)	Default value (DEFAULT)

Field becomes *Field Name*, *Type* becomes *Field Type*. Field Name and Field Type are entered into the upper area of the Table Design window. In the lower area you have the opportunity to set, under the Field properties the other column properties, in so far as they can be set using the GUI. Limitations include setting the default value of a date field to the actual date of entry. This is possible only by using the appropriate SQL command (see "Direct entry of SQL commands" on page 43).

Note	Default values: The term "Default value" in the GUI does not mean what the database user generally understands as a default value. The GUI displays a certain value visibly, which is saved with the data.
	The default value in a database is stored in the table definition. It is then written into the field whenever this is empty in a new data record. SQL default values do **not** appear *when editing table properties.*

Creation using the graphical user interface

Database creation using the graphical user interface is described in detail in Chapter 8, Getting Started with Base, in the *Getting Started with LibreOffice* book. Therefore only the main sources of error are described here.

When a table design is saved, you may be asked if a primary key should be created. This signifies that a necessary field is missing from the table. Without a primary key, the database cannot access the tables. Usually this field is given the abbreviation *ID* and the data type INTEGER and allowed to run as an AutoValue. You can make it the primary key with a right-click on the corresponding field.

If information from another table is to be pulled through into this one (for example, an address database with separate post codes and locations), you must include in the table a field of the same data type as the primary key of the other table. If the **Postcode** table has as its primary key the field *ID*, of data type *Tiny Integer*, the **Address** table must have a field *Postcode_ID* with the data type *Tiny Integer*. In other words, the **Address** table will always contain only the number that serves as the primary key in the **Postcode** table. This means that the **Address** table now has a foreign key in addition to its own primary key.

Basic rules for naming fields in the table: no two fields may have the same name. Therefore you cannot have a second field with the name *ID* as a foreign key in the Address table.

There are limits to the changes you can make in the field type. An upgrade (longer text field, larger numeric range) is unproblematic as all the values already entered will fit the new type. A downgrade creates problems and may cause loss of data.

Creating an index

Sometimes it is useful to index other fields or a combination of other fields in addition to the primary key. An index speeds up searching and can also be used to prevent duplicate entries.

Each index has a defined sort order. If a table is displayed without sorting, the sort order will be according to the content of the fields specified in the index.
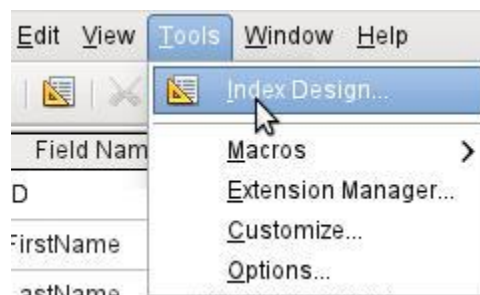

Figure 8: Access to Index Design

Open the table for editing by right-clicking and using the context menu. Then you can access index creation with **Tools > Index Design**.
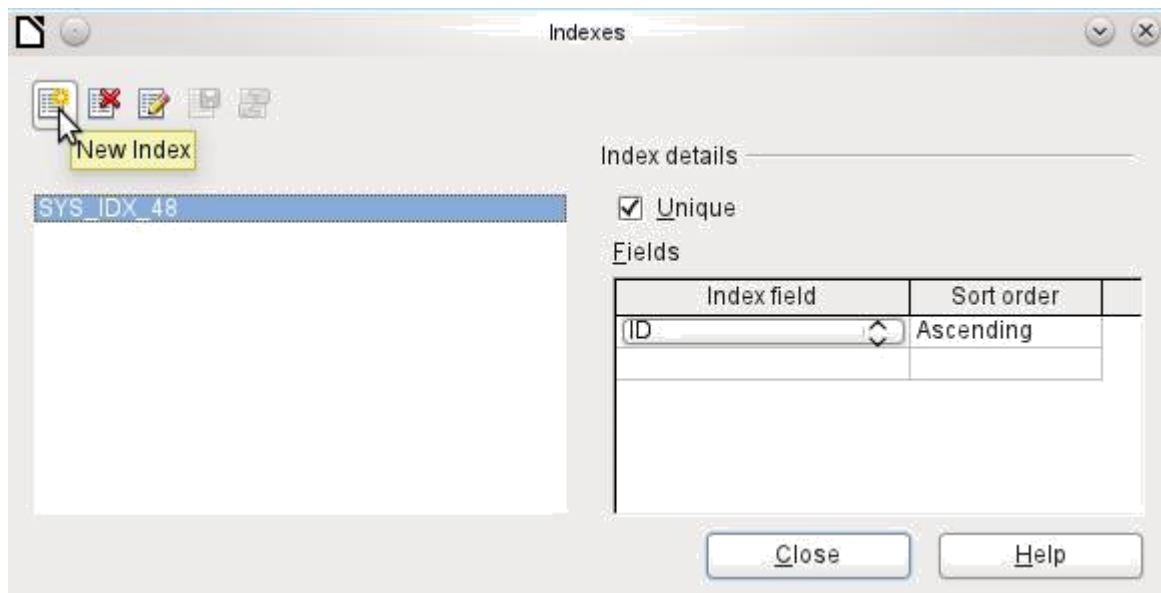
Figure 9: Creating a new Index

On the Indexes dialog (Figure 9), click **New Index** to create an index in addition to the primary key.

The new index is automatically given the name *index1*. The **Index field** specifies which field or fields are to be used for this index. At the same time you can choose the Sort order.

Figure 10: The Index is defined as Unique.

In principle, an index can also be created from table fields that do not contain unique values. However in Figure 10, the Index detail **Unique** has been checked, so that the *LastName* field together with the *FirstName* field can only have entries that do not already occur in that combination. So, for example, Robert Müller and Robert Maier are possible, and likewise Robert Müller and Eva Müller.

If an index is created for one field only, the uniqueness applies to that field. Such an index is usually the primary key. In this field each value may occur only once. Additionally, in the case of primary keys, the field cannot be NULL under any circumstances.

An exceptional circumstance for a unique index is when there is no entry into a field (the field is NULL). Since NULL can have any arbitrary value, an index using two fields is always allowed to have the same entry repeatedly in one of the fields as long as there is no entry in the other.

Note	**NULL** is used in databases to designate an empty cell, one that contains nothing. No calculation is possible using a NULL field. This contrasts with spreadsheets, in which empty fields automatically contain the value 0 (zero).

Example: In a media database, the media number and the loan date are entered when the item is loaned out. When the item is returned, a return date is entered. In theory, an index using the fields Media_ID and ReturnDate could easily prevent the same item from being loaned out repeatedly without the return date being noted. Unfortunately this will not work because the return date initially has no value. The index will prevent an item from being marked as returned twice with the same date but it will do nothing else.

Limitations of graphical table design

The sequence of fields in a table cannot be changed after the database has been saved. To display a different sequence requires a query.

Only the entry of direct SQL commands can insert a field into a specific position in the table. However, fields already created cannot be moved by this method either.

The properties of the tables must be set at the beginning: for example which fields must not be NULL and which must contain a standard value (Default). These properties cannot subsequently be changed using the GUI.

The default values set here have nothing to do with the default values within the database itself. For example, you cannot define the default for a date field as being the date of entry. That is only possible with directly entered SQL commands.

Creating tables using SQL commands

Direct entry of SQL commands

To enter SQL commands directly, go to **Tools > SQL**.

You can then enter commands into the upper area of the window (shown in Figure 11). The lower area shows the result or, if appropriate, the reason why the command failed.

Queries cannot be created here. For these, the Query Design window offers an extra possibility of editing the query in SQL Mode.
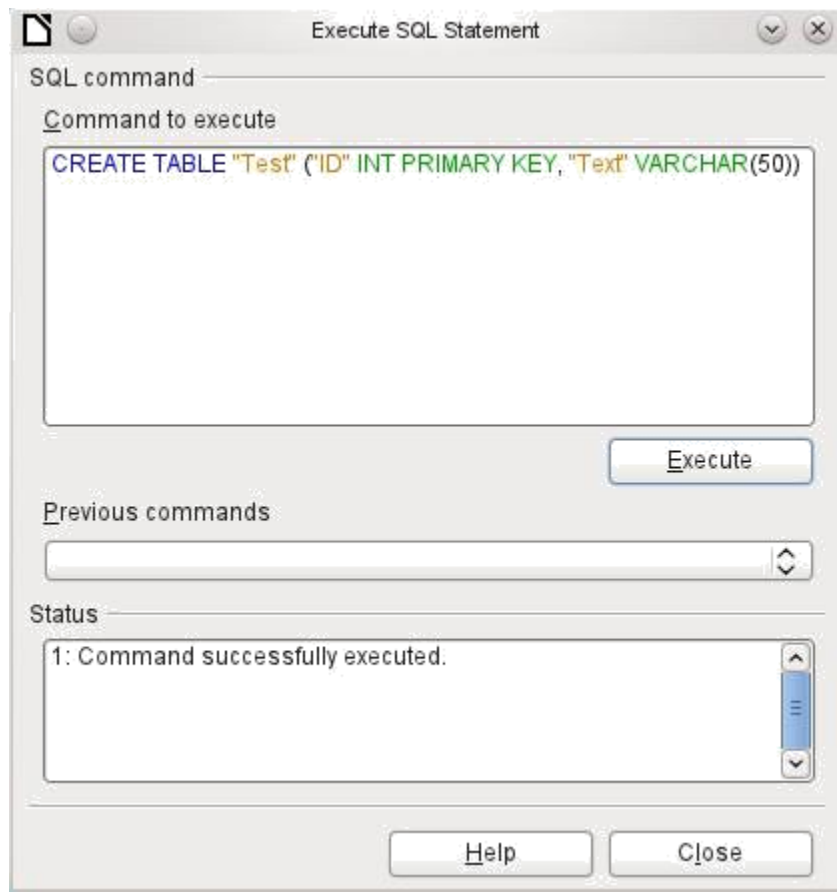
Figure 11: Dialog for direct entry of SQL commands

A summary of the possible commands for the built-in HSQLDB engine can be found at
http://www.hsqldb.org/doc/1.8/guide/ch09.html. The contents are described in the following
sections. Some commands only make sense when dealing with an external HSQLDB database
(Specify User, etc.). Where necessary, these are dealt with in the section "Working with
external HSQLDB" in the Appendix to this handbook.

Note	LibreOffice is based on Version 1.8.0 of HSQLDB. The currently available server version is 2.2. The functions of the new version are more extensive. A further description is given in the installation packages for HSQLDB, which can be downloaded from http://sourceforge.net/projects/hsqldb/files/hsqldb/.

Table creation

A simple command to create a usable table is:

```
    CREATE TABLE "Test" ("ID" INT PRIMARY KEY, "Text" VARCHAR(50));
```

CREATE TABLE "Test": Create a table with the name "Test".

( ): with the specified field names, field types and options.

"ID" INT PRIMARY KEY, "Text" VARCHAR(50): Field name "ID" with the numeric type
integer as the primary key, field name "Text" with the text type variable text length and the text
size limited to 50 characters.

```
    CREATE [MEMORY | CACHED | [GLOBAL] TEMPORARY | TEMP | TEXT]
    TABLE "Table name" ( <Field definition> [, ...] [,
    <Constraint Definition>...] ) [ON COMMIT {DELETE | PRESERVE} ROWS];
```

[MEMORY | CACHED | [GLOBAL] TEMPORARY | TEMP | TEXT]:

The default setting is **MEMORY**: HSQLDB creates all tables in core memory. This setting also applies to the tables that are written into the embedded database by LibreOffice Base. Another possibility would be to write the tables to the hard drive and use memory only to buffer access to the hard drive (**CACHED**). Tables in **TEXT** format (such as CSV) are not writable in internal databases that are set up purely in **MEMORY**, while Base cannot access **TEMPORARY** or **TEMP** tables. The SQL commands are carried out in this case but the tables are not displayed (and therefore cannot be deleted) using the GUI, and data entered via SQL is likewise not visible to the query module of the GUI, unless the automatic deletion of the contents after the final commit is prevented. Any request in this case shows a table without any contents.

Tables built directly with SQL are not immediately displayed. You must either use **View > Refresh Tables** or simply close the database and then reopen it.

<Field definition>:

```
"Field name" Data type [(Number of characters[,Decimal places])]
[{DEFAULT "Default value" | GENERATED BY DEFAULT AS IDENTITY
(START WITH <n>[, INCREMENT BY <m>])}] | [[NOT] NULL] [IDENTITY]
[PRIMARY KEY]
```

Allows default values to be included in the field definition.

For text fields, you can enter text in single quotes or **NULL**. The only SQL function allowed is **CURRENT_USER**. This only makes sense if HSQLDB is being used as an external Server database with several users.

For date and time fields, a date, a time, or a combination of the two can be entered in single quotes or else NULL. You must ensure that the date follows the American conventions (yyyy-mm-dd), that time has the format hh:mm:ss, and that a combined date/time value has the format yyyy-mm-dd hh:mm:ss.

Allowed SQL functions:

for the current date	**CURRENT_DATE, TODAY, CURDATE()**
for the current time	**CURRENT_TIME, NOW, CURTIME()**
for the current data time stamp	**CURRENT_TIMESTAMP, NOW.**

For boolean Fields (yes/no) the expressions **FALSE, TRUE, NULL** can be entered. These must be entered without single quotes.

For numeric fields, any valid number in the range, or NULL is possible. Here too, if you enter **NULL**, do not use quotes. When entering decimals, make sure that the decimal point is a dot (period) and not a comma.

For binary fields (images, etc.) any valid hexadecimal string in single quotes or **NULL** is possible. A hexadecimal example string is: '0004ff', which represents 3 bytes, first 0, then 4 and finally 255 (0xff) . As binary fields in practice need only be entered for images, you need to know the binary code of the image that is to serve as a default.

NOT NULL: The field value cannot be NULL. This condition can only be given in the field definition.

Note	Hexadecimal system: Numbers are based on 16. A mixed system consisting of the numbers 0 to 9 and the letters a to f provides 16 possible digits for each column. With two columns, you can have 16*16=256 possible values. This corresponds to 1 Byte (2^8).

Working with Tables

<Constraint definition>:
```
[CONSTRAINT "Name"]
UNIQUE ( "Field_name 1" [,"Field_name 2"...] ) |
PRIMARY KEY ( "Field_name 1" [,"Field_name 2"...] ) |
FOREIGN KEY ( "Field_name 1" [,"Field_name 2"...] )
REFERENCES "other_table_name" ( "Field_name_1" [,"Field_name 2"...])
[ON {DELETE | UPDATE}
{CASCADE | SET DEFAULT | SET NULL}]
| CHECK(<Search_condition>)
```
Constraints define conditions that must be fulfilled when data is entered. Constraints can be given a name.

UNIQUE ("Field_name"): the field value must be unique within that field

PRIMARY KEY ("Field_name"): the field value must be unique and cannot be **NULL** (primary key)

FOREIGN KEY ("Field_name") REFERENCES <"other_table_name">
("Field_name"): The specified fields of this table are linked to the fields of another table. The field value must be tested for referential integrity as foreign keys; that is, there must be a corresponding primary key in the other table, if a value is entered here.

[ON {DELETE | UPDATE} {CASCADE | SET DEFAULT | SET NULL}]: In the case of a foreign key, this specifies what is to happen if, for example, the foreign record is deleted. It makes no sense, in a loan table for a library, to have a user number for which the user no longer exists. The corresponding record must be modified so that the relationship between the tables remains valid. Usually the record is simply deleted. This happens if you select **ON DELETE CASCADE**.

CHECK(<Search_condition>): Formulated as a **WHERE** condition, but only for the current record.

You need constraints when the relationship between tables or the index for certain fields must be defined.

[ON COMMIT {DELETE | PRESERVE} ROWS]:

The content of tables of the type **TEMPORARY** or **TEMP** is erased by default when you have finished working with a particular record (**ON COMMIT DELETE ROWS**). This allows you to create temporary records, which contain information for other actions to be carried out at the same time.

If you want a table of this type to contain data available for a whole session (from opening a database to closing it), choose **ON COMMIT PRESERVE ROWS**.

Table modification

Sometimes you might wish to insert an additional field into a particular position in the table. Suppose you have a table called **Addresses** with fields *ID*, *Name*, *Street*, and so on. You realize that perhaps it would be sensible to distinguish first names and last names.
```
ALTER TABLE "Addresses" ADD "First Name" VARCHAR(25) BEFORE "Name";
```
ALTER TABLE "Addresses": Alter the table with the name "Addresses".
ADD "First Name" VARCHAR(25): insert the field "FirstName" with a length of 25 characters.
BEFORE "Name": before the field "Name".

The possibility of specifying the position of additional fields after the creation of the table is not available in the GUI.
```
ALTER TABLE "Table_name" ADD [COLUMN] <Field_definition>
[BEFORE "already_existing_field_name"];
```
The additional designation **COLUMN** is not necessary in cases where no alternative choices are available.

```
ALTER TABLE "Table_name" DROP [COLUMN] "Field_name";
```
The field "Field name" is erased from the table *Table_name*. However this does not take place if the field is involved in a view or as a foreign key in another table.
```
ALTER TABLE "Table_name" ALTER COLUMN "Field_name" RENAME
TO "New_field_name"
```
Changes the name of a field.
```
ALTER TABLE "Table_name" ALTER COLUMN "Field_name" SET
DEFAULT <Standard value>};
```
Sets a specific default value for the field. **NULL** removes an existing default value.
```
ALTER TABLE "Table_name" ALTER COLUMN "Field_name" SET [NOT]
```
NULL Sets or removes a **NOT NULL** condition for a field.
```
ALTER TABLE "Table_name" ALTER COLUMN <Field definition>;
```
The field definition corresponds to the one from the Table creation with the following restrictions:

- The field must already be a primary key field to accept the property **IDENTITY**. **IDENTITY** means, that the field has the property AutoValue. This is possible only for **INTEGER** or **BIGINT** fields. For these field type descriptions, see the Appendix to this handbook.

- If the field already has the property **IDENTITY** but it is not repeated in the field definition, the existing **IDENTITY** property is removed.

- The default value will become that specified in the new field definition. If the definition of the default value is left blank, any default already defined is removed.

- The property **NOT NULL** continues into the new definition, if not otherwise defined. This is in contrast to the default value.

- In some cases, depending on the type of modification, the table must be empty in order for the change to occur. In all cases the change will have effect only if it is possible in principle (for example a change from **NOT NULL** to **NULL**) and the existing values can all be translated (for example a change from **TINYINT** to **INTEGER**).

```
ALTER TABLE "Table_name" ALTER COLUMN "Field_name" RESTART WITH
<New_field_value>
```
This command is used exclusively for an **IDENTITY** field. It determines the next value for a field with the Autovalue function set. It can be used, for example, when a database is initially used with test data, and subsequently provided with real data. This requires the contents of the tables to be deleted and a new value such as "1" to be set for the field.
```
ALTER TABLE "Table_name"
ADD [CONSTRAINT "Condition_name"] CHECK (<Search_condition>);
```
This adds a search condition introduced by the word **CHECK**. Such a condition will not apply retrospectively to existing records, but it will apply to all subsequent changes and newly entered records. If a constraint name is not defined, one will be assigned automatically.

Example:
```
ALTER TABLE "Loan" ADD CHECK
(IFNULL("Return_Date","Loan_Date")>="Loan_Date")
```
The **Loan** table needs to be protected from input errors. For example, you must prevent a return date being given that is earlier than the loan date. Now if this error occurs during the return process, you will get an error message **Check constraint violation** …
```
ALTER TABLE "Table_name"
ADD [CONSTRAINT "Constraint_name"] UNIQUE
("Field_name1", "Field_name2" ...);
```

Working with Tables

Here a condition is added that forces the named fields to have different values in each record. If several fields are named, this condition applies to the combination rather than the individual fields. **NULL** does not count here. A field can therefore have the same value repeatedly without causing any problems, if the other field in each of the records is **NULL**.

This command will not work if there is already a **UNIQUE** condition for the same field combination.

```
ALTER TABLE "Table_name"
ADD [CONSTRAINT "Constraint_name"] PRIMARY KEY
("Field_name1", "Field_name2" ...);
```
Adds a primary key, optionally with a constraint, to a table. The syntax of the constraint is the same as when a table is created.

```
ALTER TABLE "Table_name"
ADD [CONSTRAINT "Constraint_name"] FOREIGN KEY
("Field_name1", "Field_name2" ...)
REFERENCES "Table_name_of_another_table" ("Field_name1_other_table",
"Field_name2_other_table" ...)
[ON {DELETE | UPDATE} {CASCADE | SET DEFAULT | SET NULL}];
```
This adds a foreign key (**FOREIGN KEY**) to the table. The syntax is the same as when a table is created.

The operation will terminate with an error message, if any value in the table does not have a corresponding value in the table containing that primary key.

Example: The **Name** and **Address** tables are to be linked. The **Name** table contains a field with the name *Address_ID*. The value of this should be linked to the field *ID* in the **Address** table. If the value "1" is found in *Address_ID* but not in the *ID* field of the **Address** table, the link will not work. It will not work either if the two fields are of different types.

```
ALTER TABLE "Table_name" DROP CONSTRAINT "Constraint_name";
```
This command removes the named constraint (**UNIQUE**, **CHECK**, **FOREIGN KEY**) from a table.

```
ALTER TABLE "Table_name" RENAME TO "new_table_name";
```
Finally this command changes only the name of a table.

Deleting tables

```
DROP TABLE "Table name" [IF EXISTS] [RESTRICT |
```
CASCADE]; Deletes the table "Table name".

IF EXISTS prevents an error occurring if this table does not exist.

RESTRICT is the default arrangement and need not be explicitly chosen; it means that deletion does not occur if the table is linked to another table by the use of a foreign key or there is an active view of this table. Queries are not affected as they are not stored within HSQLDB.

If instead you choose **CASCADE**, all links to the table "Table_name" are deleted. In the linked tables, all foreign keys are set to NULL. All views referring to the named table are also completely deleted.

Working with Tables

Working with Relationships

Relationships between tables

This chapter explains many of these steps in detail, using an example database for a library: `media_without_macros`. Constructing the tables for this database is an extensive job, as it covers not only the addition of items into a media library but also the subsequent loan of them.

Relationships for tables in databases

The more relationships there are between tables, the more complex is the design task. Figure 2 shows the overall table structure of this example database as an overview, scaled to fit the page size of this document. To read the content, zoom the page to 200%.
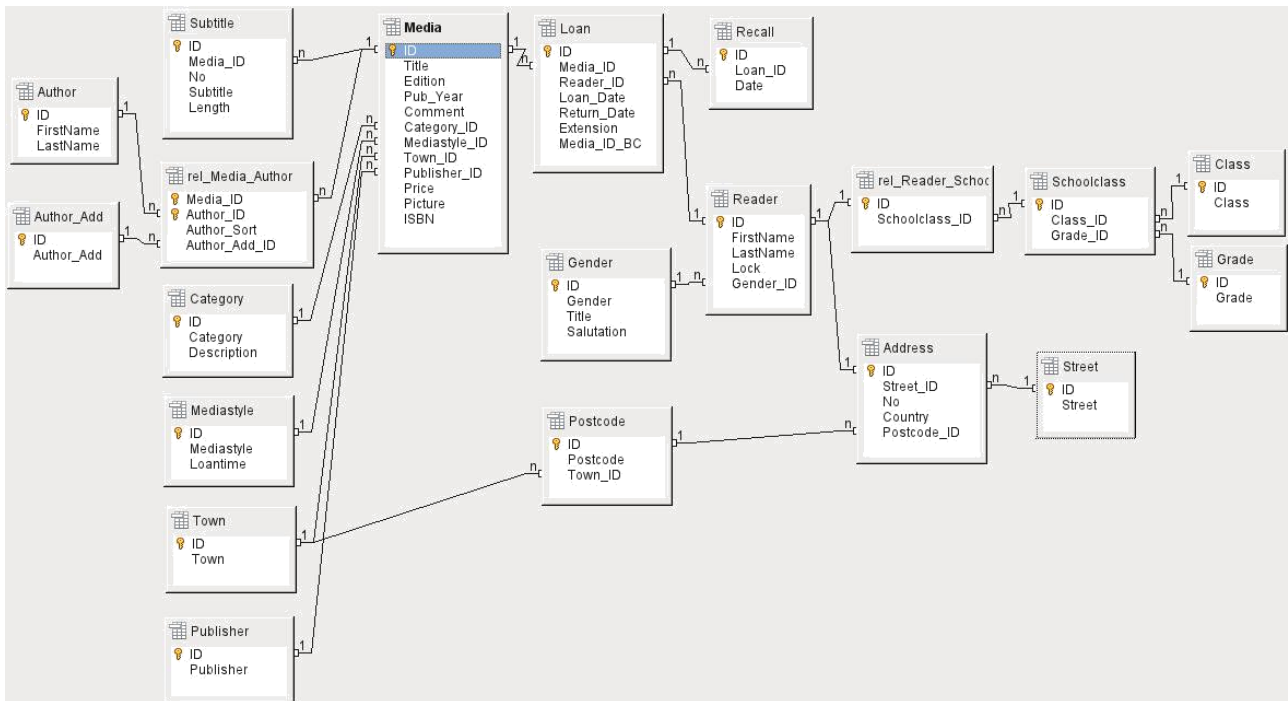
Figure 2: Relationship diagram for the example database media_without_macros

One-to-many relationships

The media_without_macros database lists the titles of the media in one table. Because titles can have multiple subtitles or sometimes none at all, the subtitles are stored in a separate table.

This relationship is known as one-to-many (1:n). Many subtitles may be assigned to one medium, for example the many track titles for a music CD. The primary key for the *Media* table is stored as a foreign key in the *Subtitle* table. The majority of relationships between tables in a database are one-to-many relationships.
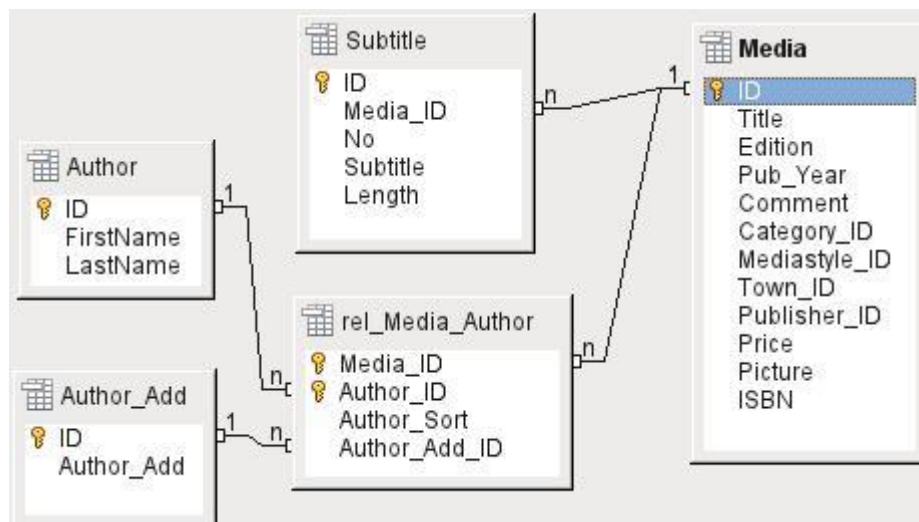
Figure 3: Example 1:n relationship; n:m relationship

Many-to-many relationships

A database for a library might contain a table for authors' names and a table for the media. The connection between an author and, for example, books that author has written, is obvious. The library might contain more than one book by one author. It might also contain books with multiple authors. This relationship is known as many-to- many (n:m). Such relationships require a table that acts as an intermediary between the two tables concerned. This is represented in Figure 3 by the *rel_Media_Author* table.

Thus, in practice, the n:m relationship is solved by treating it as two 1:n relationships. In the intermediate table, the *Media_ID* can occur more than once, as can the *Author_ID*. But when using them as a pair, there is no duplication: no two pairs are identical. So this pair meets the requirements for the primary key for the intermediate table.

Note	For a given value for Media_ID, there is only one title of the media and one ISBN.
	For a given value for Author_ID, there is only one Author's first and last name. So, for a given pair of these values, there is only one ISBN and only one Author. This makes the pair unique.

One-to-one relationships
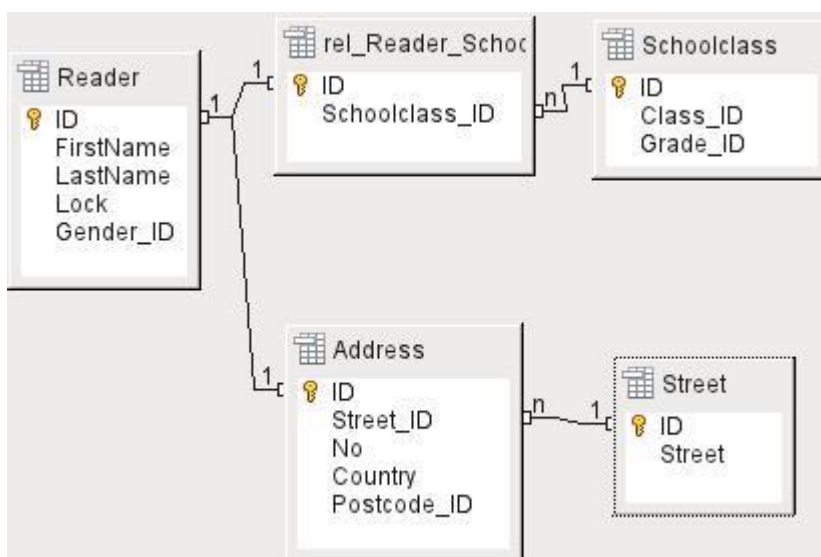
Figure 4: Example 1:1 relationship

The library database described above requires a table for readers. In this table only the fields that are directly necessary were planned in advance. But for a school database, the school class is also required. From the school class records, you can find borrowers' addresses where necessary. Therefore it is not necessary to include these addresses in the database. The school class relationship of pupils is separated from the reader table, because mapping to classes is not appropriate in all areas. From this arises a 1:1 relationship between the reader and the individual school class assignment.

In a database for a public library, the addresses of readers are required. For each reader there is a single address. If there are multiple readers at the same address, this structure would require the address to be entered again, since the primary key of the *Reader* table is entered directly as the primary key in the *Address* table. Primary key and foreign key are one and the same in the *Address* table. This is therefore a 1:1 relationship.

A 1:1 relationship does not signify that for every record in a table, there will be a corresponding record in another table. But **at most** there will be only one corresponding record. A 1:1 relationship

therefore leads to fields being exported which will be filled with content for only some of the records.

Tables and relationships for the example database

The example database (media_without_macros) must satisfy three requirements: media additions and removals, loans, and user administration.

Media addition table

First, media must be added into the database so that a library can work with them. However, for a simple summary of a media collection at home, you could create easier databases with the wizard; that might be sufficient for home use.

The central table for Media addition is the *Media* table (see Figure 5)**.**
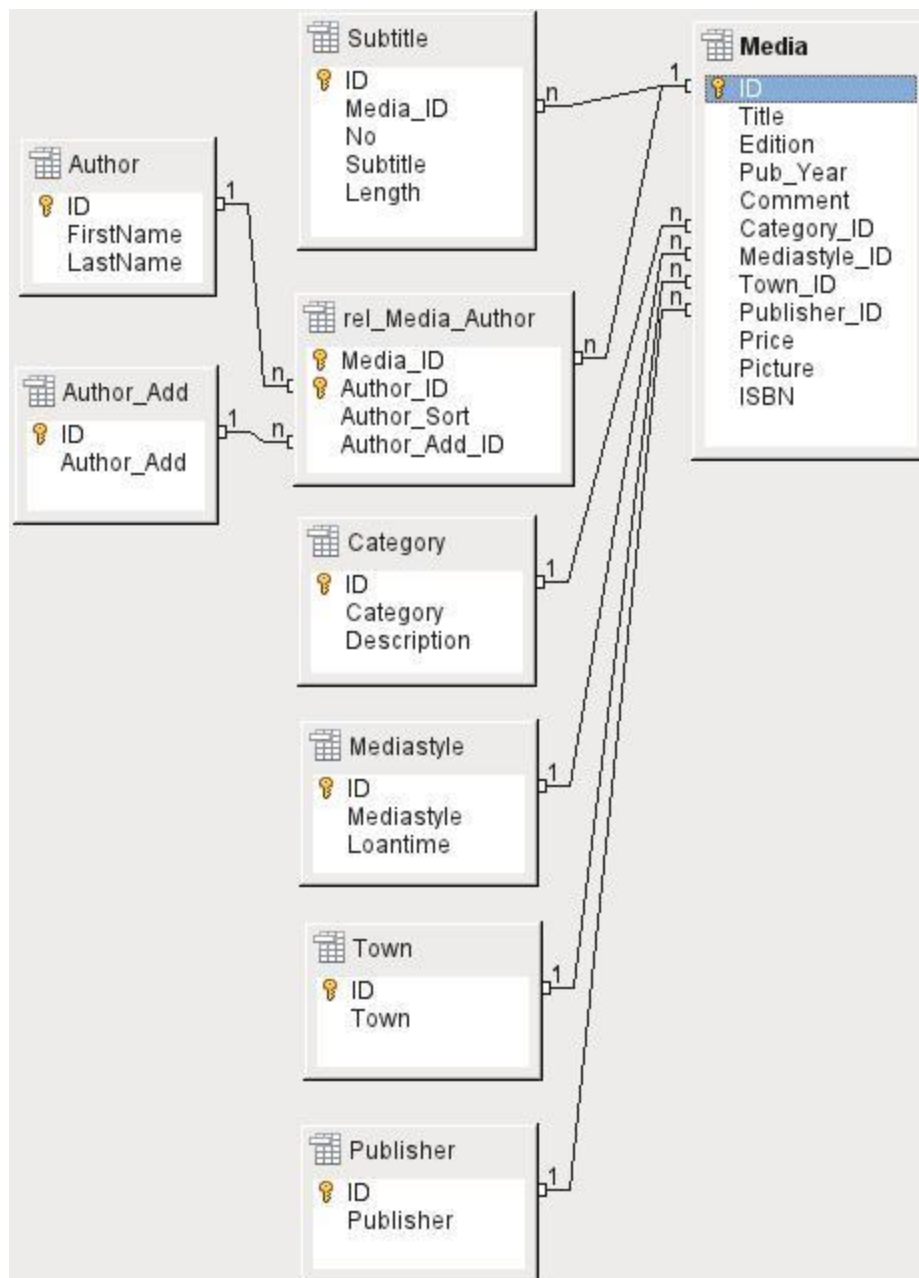
Figure 5: Media addition

In this table all fields that are directly entered are assumed not to be also in use for other media with the same content. Duplication should therefore be avoided.

For this reason, planned fields in the table include the title, the ISBN, an image of the cover, and the year of publication. The list of fields can be extended if required. So, for instance, librarians might want to include fields for the size (number of pages), the series title, and so on.

The *Subtitle* table contains the detailed content of CDs. As a CD can contain several pieces of music, a record of the individual pieces in the main table would require a lot of additional fields (Subtitle 1, Subtitle 2, etc.) or the same item would have to be entered many times. The *Subtitle* table therefore stands in a n:1 relationship to the *Media* table.

The fields of the *Subtitle* table are (in addition to the subtitle itself) the sequence number of the subtitle and the duration of the track. The *Length* field must first be defined as a time field. In this way, the total duration of the CD can be calculated and displayed in a summary if necessary.

The authors have a n:m relationship to the media. One item can have several authors, and one author might have created several items. This relationship is controlled by the *rel_Media_Author* table. The primary key of this linking table is the foreign key, formed from the *Author* and *Media* tables. The *rel_Media_Author* table includes an additional sorting (Author_Sort) of authors, for example by the sequence in which they are named in the book. In addition, a supplementary label such as Producer, Photographer and so on is added to the author where necessary.

Category, Mediastyle, Town and Publisher have a 1:n relationship.

For the **Category,** a small library can use something like Art or Biology. For larger libraries, general systems for libraries are available. These systems provide both abbreviations and complete descriptions. Hence both fields appear under Category.

The **Mediastyle** is linked to the loan period *Loantime.* For example, video DVDs might on principle have a loan period of 7 days, but books might be loaned for 21 days. If the loan period is linked to any other criteria, there will be corresponding changes in your methodology.

The **Town** table serves not only to store location data from the media but also to store the locations used in the addresses of users.

Since **Publishers** also recur frequently, a separate table is provided for them.

The *Media* table has in total four foreign keys and one primary key, which is used as a foreign key in two tables, as shown in Figure 5.

Loan table

The central table is **Loan** (see Figure 6). It is the link between the Media and Reader tables. In case you need to find out retrospectively who has taken out a book (for example, if someone notices during the loan process that the book is damaged, or if you wish to make a list of the most popular media), the *Loan_Date* in the loan record is not simply deleted during the return process. Instead a *Return_Date* is recorded.

Similarly, Reminders are integrated into the loan procedure. Each reminder is separately entered into the **Recall** table so that the total number of reminders can be determined.

As well as an extension period in weeks, there is an extra field in the loan record that enables media to be loaned using a barcode scanner (*Media_ID_BC*). Barcodes contain, in addition to the individual Media_ID, a check digit which the scanner can use to determine if the value scanned in is correct. This barcode field is included here only for test purposes. It would be better if the primary key of the Media table could be directly entered in barcode form, or if a macro were used to remove the check digit from the entered barcode number before storage.

Finally we need to connect the **Reader** to the loan. In the actual reader table, only the name, an optional lock, and a foreign key linking to the Gender table are included in the plan.
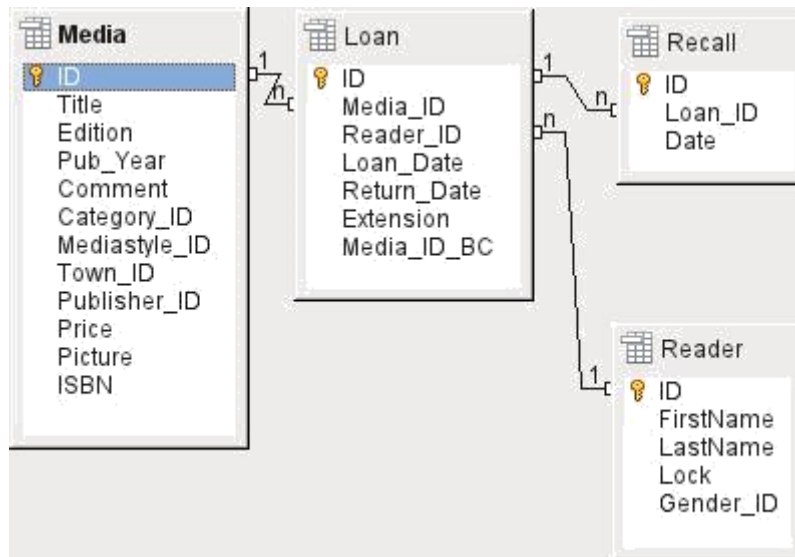
Working with Tables

Figure 6: Loan

User administration table

For this table design, two scenarios are envisaged. The chain of tables shown in Figure 6 is designed for school libraries. Here there is no need for addresses, as the pupils can be contacted through the school. Reminders do not need to be sent out by post but can be distributed internally.

The Address chain is necessary in the case of public libraries. Here you need to enter data that will be needed for the creation of reminder letters. See Figure 7.
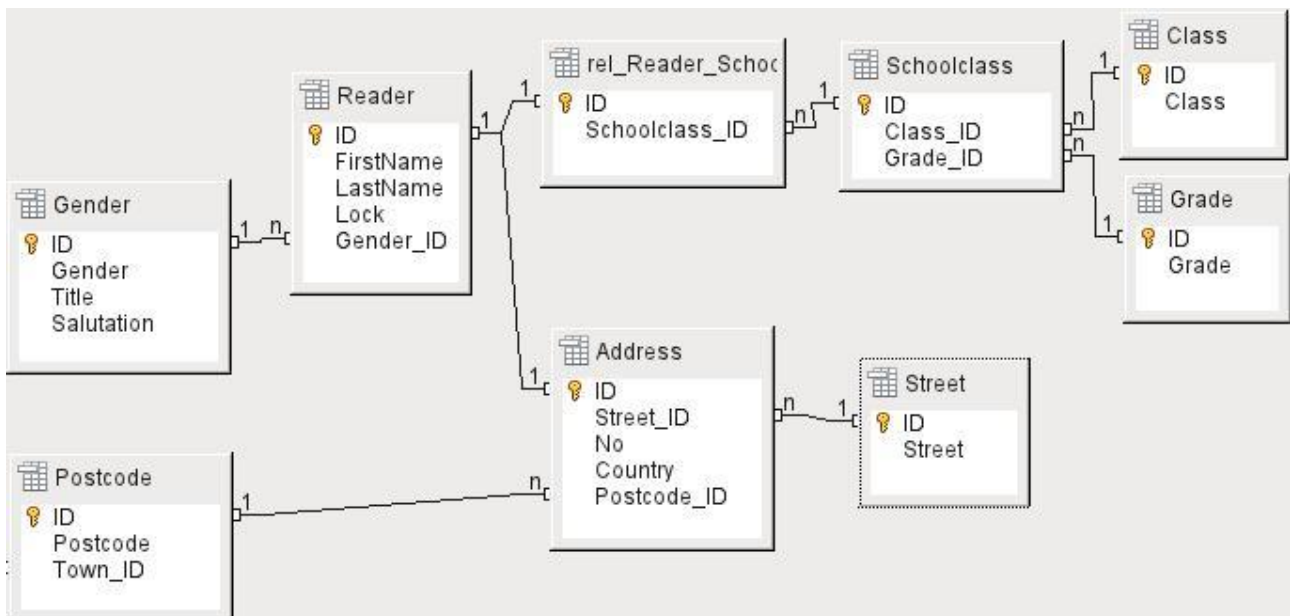

Figure 7: Readers - a School class chain and an Address chain

The **Gender** table ensures that the correct *Salutation* is used in reminders. The writing of reminders can then be automated as far as possible. In addition, some given names can be equally masculine or feminine. Therefore the separate listing of gender is required even when reminders are written out by hand.

The **rel_Reader_Schoolclass** table, like the **Address** table, has a 1:1 relationship with the **Reader** table. This was chosen because either the school class or the address might be required. Otherwise the *Schoolclass_ID* could be put directly into the pupil table; the same would be true of the complete content of the address table in a public library system.

A **School class** usually consists of a year designation and a stream suffix. In a 4- stream school, this suffix might run from **a** to **d**. The suffix is entered in the **Class** table. The year is in a separate **Grade** table. That way, if readers move up a class at the end of each school year, you can simply change the year entry for everyone.

The **Address** is also divided. **Street** is stored separately because street names within an area are often repeated. Post code and town are separated because there are often several post codes for a single area and therefore more post codes than towns. So compared with the **Address** table, the **Postcode** table contains significantly fewer records and the **Town** table even fewer.

How this table structure is put to use is explained further in the Chapter 4, Forms, in this handbook.

Linking tables

In principle you can have a database without links between tables. The user must then ensure during data entry, that the relationships between the tables remain correct. This usually occurs through the use of suitable input forms that manage this.

Deleting records in linked tables is not a simple matter. Suppose you wish to delete a particular street from the *Street* table in Figure 7, where this field is linked with the *Address* table as a foreign key in that table. The references in the *Address* table would disappear. The database does not allow this, once the relationship has been created. In order to delete the *Street*, the precondition must be fulfilled, that it is no longer referenced in the *Address* table.

Basic links are made using **Tools > Relationships**. This creates a connection line from the primary key in one table to the defined foreign key in the other.

You may receive the following error message when creating such a link:
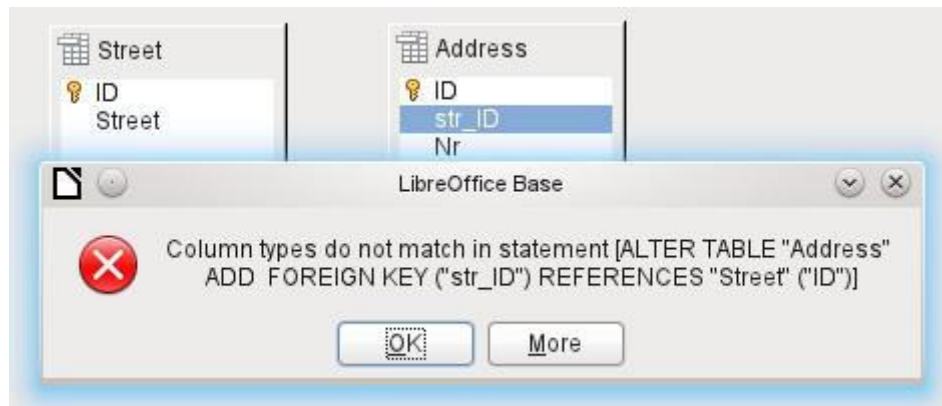
This message shows some text in English and the internal SQL command that caused the error.

Column types do not match in statement—As the SQL command is displayed as well, the reference is clearly to the columns *Address.str_ID* and *Street.ID*. For test purposes one of these fields was defined as an Integer, the other as Tiny Integer. Therefore no link could be created since the one field cannot have the same value as the other.

In this case the column types match. The SQL statement is the same as in the first example. But again there is an error:

Integrity constraint violation – no parent 1, table: Address ... —The integrity of the relationship is not ensured. In the field of the Address table, *Address.str_ID*, there is a number 1, which is not present in *Street.ID*. The parent table here is *Street*, since its primary key is the one that must exist. This error is very common, when two tables are to be linked and some fields in the table with the prospective foreign key already contain data. If the foreign key field contains an entry that is not present in the parent table (the table containing the primary key), this is an invalid entry.
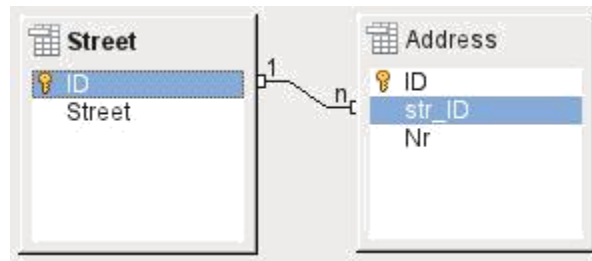
If the linking is carried out successfully and subsequently there is an attempt to enter a similarly invalid record into the table, you get the following error message:

Again this is an integrity violation. Base refuses to accept the value 1 for the field *str_ID* after the link has been made because the *Street* table contains no such *v*alue in the ID field.
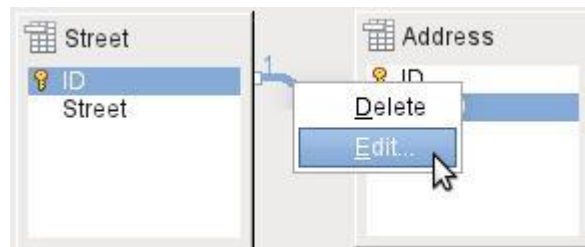

Figure 12: Links can be edited with a right-click

The properties of a link can be edited so that the deletion of a record from the *Street* table will simultaneously set to NULL the corresponding entries in the *Address* table.

The properties shown in Figure 12 always relate to an action linked to the change in a record from the table containing the corresponding primary key. In our case this is the *Street* table. If the **primary key of a record** in this table is **altered (Update)**, the following actions might take place.

No action
    Changing the primary key *Street.ID* is not allowed in this case, as it would break the relationship between the tables.

Update cascade
    If the primary key *Street.ID* is changed, the foreign key is automatically changed to its new value. This ensures that the linkage is not damaged. For example, if a value is changed from 3 to 4, all records from the *Address* table that contain the foreign key *Address.Street_ID* with the value 3, have it changed to 4.
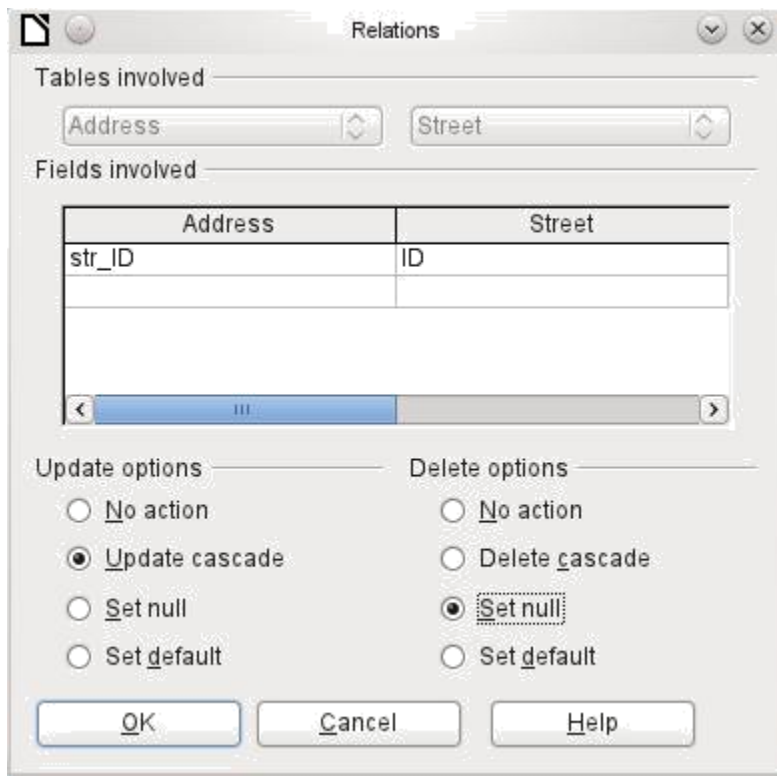
Figure 13: Editing the properties of a relationship

Set null

All records which contain this particular primary key will now have no entry in the foreign key field *Address.Street_ID;* the field will be NULL.

Set default

If the primary key *Street_ID* is changed, the value of *Address.Street_ID* originally linked to it is set to the previously defined default value. For this purpose we need an unambiguous definition of a default value. The LibreOffice Base GUI up to Version 3.5 does not seem to provide this. If the default is set using the SQL statement:

```
ALTER TABLE "Address" ALTER COLUMN "Street_ID" SET DEFAULT 1;
```

the link definition ensures that the field will return to this value in the case of an Update. So if the primary key in the *Street* table is changed, the corresponding foreign key in the *Address* table will be set to 1. This is useful when a record is required to have a street field, in other words this field cannot be NULL. But be careful! If 1 is not in use, you will have created a link to a non-existent value. In this case HSQLDB seems to have been designed without due thought. It is therefore possible to destroy the integrity of the relationship.

Attention	If the default value in a foreign key field is not linked to a primary key of the foreign table, a link to a value would be created, that isn't possible. The referential integrity of the database would be destroyed.

It would be better not to use the possibility to set the value to default.

If a record is **deleted** from the *Street* table, the following options are available.

No Action

No action takes place. If the requested deletion affects a record in the Address table, the request will be refused.

Cascading Delete

If a record is deleted from the *Street* table and this affects a record in the *Address* table, that record will also be deleted.

That might seem strange in this context but there are other table structures in which it makes a lot of sense. Suppose you have a table of CDs and a table which stores the titles on these CDs. Now if a record in the CD table is deleted, many titles in the other table have no meaning as they are no longer available to you. In such cases, a cascading deletion makes sense. It means that the user does not need to delete all these titles before deleting the CD from the database.

Set to Null

This is the same as for the update option.

Set to Default

This is the same as for the update option and requires the same precautions.

Tip	To avoid error messages from the database as far as possible, since these may not always be comprehensible to the user, the *No Action* option should definitely be avoided.

Working with Tables
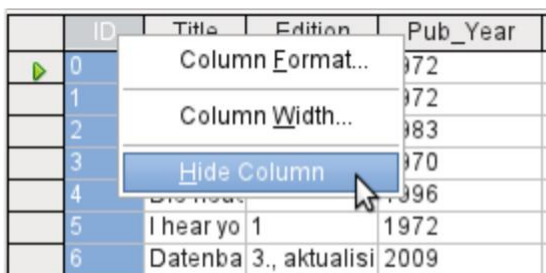
Entering data into tables

Entering data into tables

Databases that consist of only a single table usually do not require an input form unless they contain a field for images. However as soon as a table contains foreign keys from other tables, users must either remember which key numbers to enter or they must be able to look at the other tables simultaneously. In such cases, a form is useful.
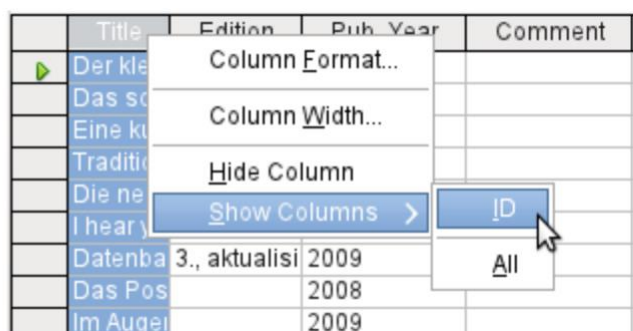
Entry using the Base GUI

Tables in the table container are opened by double-clicking them. If the primary key is an automatically incrementing field, one of the visible fields will contain the text *AutoValue* . No entry is possible into the *AutoValue* field. Its assigned value can be altered if required, but only after the record has been committed.

Figure 14: Entry into tables – Hiding columns

Figure 15: Entry into tables – Unhiding columns

Individual columns in the Table Data View can be hidden. For example, if the primary key field does not need to be visible, this can be specified in the table in data entry view by right- clicking on the column header. This setting is stored with the GUI. The column continues to exist in the table and can always be made visible again.

Entry into the table usually takes place from left to right using the keyboard with the *Tab or Enter* keys. You can also use the mouse.

Working with Tables

When you reach the last field of a record, the cursor automatically jumps to the next record. The previous entry is committed to storage. Additional storage using **File > Save** is not necessary and indeed not possible. The data is already in the database.

Caution	For the HSQLDB, data is in working memory. It will only be transferred to the hard drive when Base is closed (unfortunately from the viewpoint of data security). If Base for some reason does not close down in an orderly fashion, this can lead to loss of data.

If no data is entered into a field that has been previously defined during table design as mandatory (**NOT NULL**), the appropriate error message is displayed:
Attempt to insert null into a non-nullable column …

The corresponding column, the table and the SQL command (as translated by the GUI) are also displayed.

Changing a record is easy: find the field, enter a different value, and leave the row again.
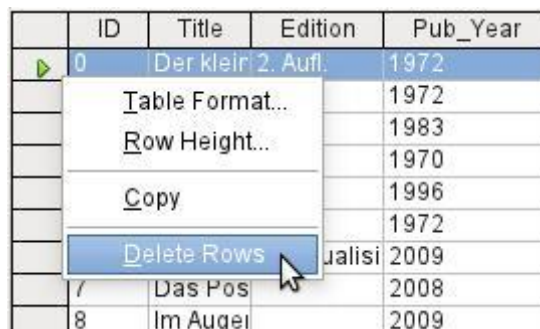
To delete a record, select the row by clicking its header (the grey area to the left), right-click and choose **Delete Rows**.

The Sort, Search, and Filter functions are very useful for retrieving particular records.
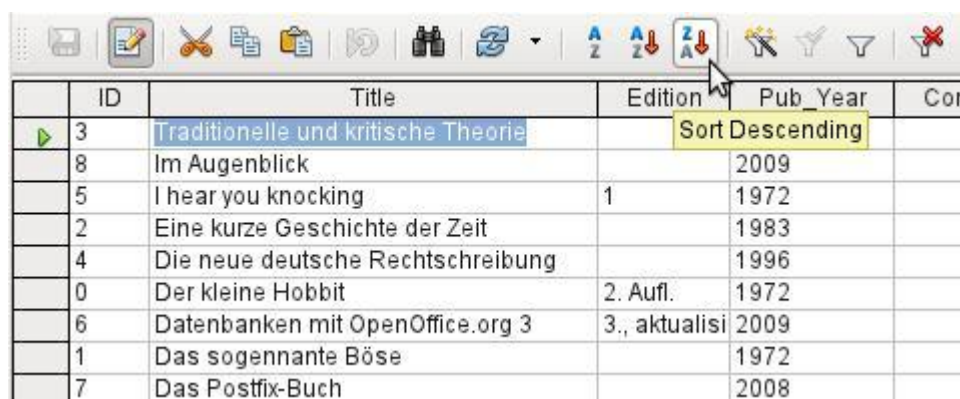
Sorting tables

Figure 16: Quick Sort

The quick sort option hides behind the **A→Z** and **Z→A b**uttons. One field is selected, one click on the button, and the data is sorted by that column. The figure shows a descending sort by the *Title* field.

Quick sort sorts only by one column. To sort by several columns simultaneously, a further sort function is provided.

The field name of the column and the current sort order are selected. If a previous quick sort has been carried out, the first row will already contain the corresponding field name and sort order.
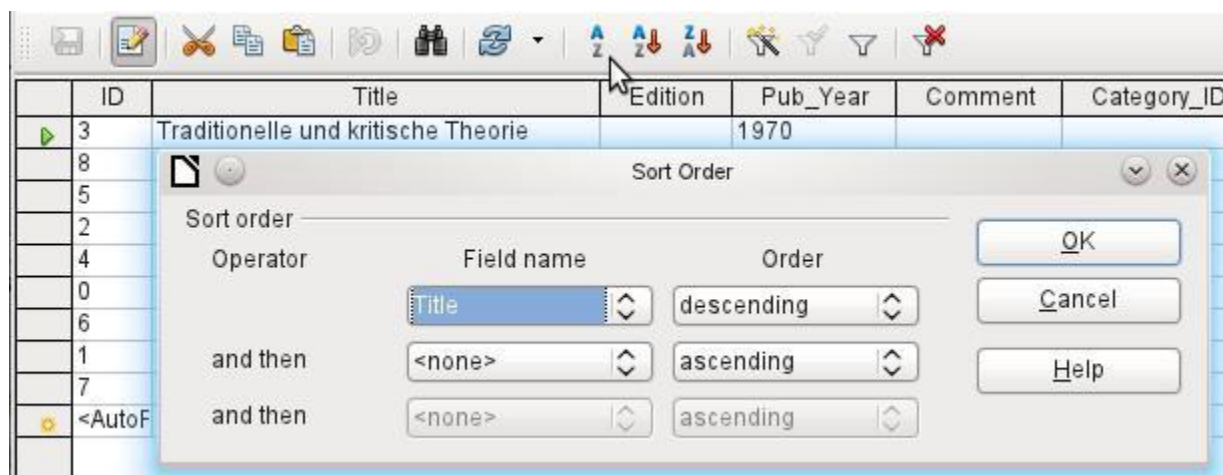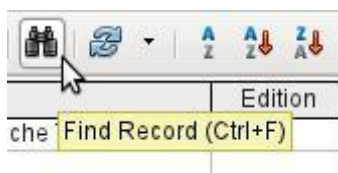

Figure 17: Sorting by more than one column

Searching tables

Functions for searching records are extensive and perhaps not the first choice for locating a particular record for users accustomed to search engines.

Tip	Before you search, make sure the columns you will be searching are wide enough to show correctly the records that you will find. The search window remains in the foreground and you will not be able to correct the settings for column width in the underlying table. To reach the table, you must break off the search.

The search takes over terms from the field from which it was invoked.

To make the search effective, the search area should be limited as far as possible. It would be pointless to search for the above text from the *Title* field in the *Author* field. Instead, the field name *Title* is already suggested as the single Field name.

Further settings for the search can make things easier through specific combinations. You can use the normal SQL placeholders ("_" for a variable character, "%" for an arbitrary number of variable characters, "\" as an escape character, to enable these special characters themselves to be searched for).

Regular expressions are described in detail in LibreOffice Help. Apart from that, the Help available for this module is rather sparse.

Working with Tables

Figure 18: Entry mask for a Record search

Figure 19: Limiting the similarity search

The similarity search function is useful when you need to exclude spelling mistakes. The higher the values that you set, the more records will be shown in the final list.

This search module is most suitable for people who know, from regular use, exactly how to achieve a given result. Most users are more likely to succeed in finding records by using a filter.

Chapter 4 of this handbook describes the use of forms for searching, and how the use of SQL and macros can accomplish a keyword search.

Working with Tables

Filtering tables
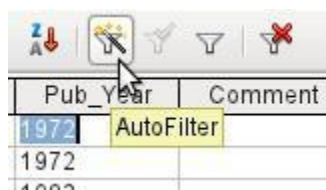
You can filter a table quickly by using the **AutoFilter**. Place the cursor in a field, and one click on the icon causes the filter to take over the content of this field. Only those records are shown for which the chosen field has the same content. The figure below shows filtering according to an entry in the **Pub_Year** column.

The filter is active, as shown by the filter icon with a green check mark. The filter symbol is shown pressed in. If the button is clicked again, the filter continues to exist, but all records are now shown. So, if you want, you can always return to the filtered state.

Clicking on the **Remove Filter/Sort** icon at the extreme right causes all existing filters and sorts to be removed. The filters become inactive and can no longer be recovered with their old values.

Tip	You can still enter records normally into a filtered table or one that has been restricted by a search. They remain visible in the table view until the table is updated by pressing the **Refresh** button.

The **Standard Filter** icon opens a dialog inwhich you can filter using several simultaneous criteria, similar to doing a sort. If AutoFilter is in use, the first line of the Standard Filter will already show this existing filter criterion.

Figure 20: Multiple Data Filtering using the Standard Filter

The Standard Filter provides many of the functions of SQL data filtering. The following SQL commands are available.

GUI Condition	Description
=	Exact equality; corresponds to **like**, but without any additional placeholders
<>	Unequal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
like	For text, written in quotation marks (' '); "_" for a variable character, "%" for an arbitrary number of variable characters
not like	Opposite of **like**, in SQL **NOT LIKE**
empty	No entry, not even a space character. In SQL this is expressed by the term **NULL**
Not empty	Opposite of empty, in SQL **NOT NULL**

Before one filter criterion can be combined with another, the following row must have at least one field name selected. In Figure 20, the word – *none* – is shown instead of a field name, so the combination is not active. The combination operators available are **AND** and **OR**.

The field name can be a new field name or a previously selected one.

Even for large data collections, the number of retrieved records can be reduced to a manageable set with skillful filtering using these three possible conditions.

In the case of filtering forms too, there are some further possibilities (described in the following chapter) which are not provided by the GUI.

Source: - Libre Office Base Handbook Version 4.0
https://documentation.libreoffice.org/assets/Uploads/Documentation/en/BH4.0/PDF/BH40BaseHandbook.pdf