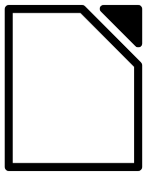


LibreOffice
The Document Foundation

Base Handbook

Chapter 5
Queries

Copyright

Contributors

Jochen Schiffers
Robert Großkopf
Jost Lange
Hazel Russman

Feedback

Please direct any comments or suggestions about this document to: documentation@global.libreoffice.org.

Caution

Everything you send to a mailing list, including your email address and any other personal information that is written in the mail, is publicly archived and cannot be deleted.

Acknowledgments

This chapter is based on an original German document and was translated by Hazel Russman.

Publication date and software version

Published 19 April 2013. Based on LibreOffice 3.5.

Note for Mac users

Some keystrokes and menu items are different on a Mac from those used in Windows and Linux. The table below gives some common substitutions for the instructions in this chapter. For a more detailed list, see the application Help.

Windows or Linux	*Mac equivalent*	*Effect*
Tools > Options menu selection	**LibreOffice > Preferences**	Access setup options
Right-click	*Control+click*	Open a context menu
Ctrl (Control)	*⌘ (Command)*	Used with other keys
F5	*Shift+⌘+F5*	Open the Navigator
F11	*⌘+T*	Open the Styles and Formatting window

Contents

General information on queries

Queries to a database are the most powerful tool that we have to use databases in a practical way. They can bring together data from different tables, calculate results where necessary, and quickly filter a specific record from a mass of data. The large Internet databases that people use every day exist mainly to deliver a quick and practical result for the user from a huge amount of information by thoughtful selection of keywords – including the search-related advertisements that encourage people to make purchases.

Entering queries

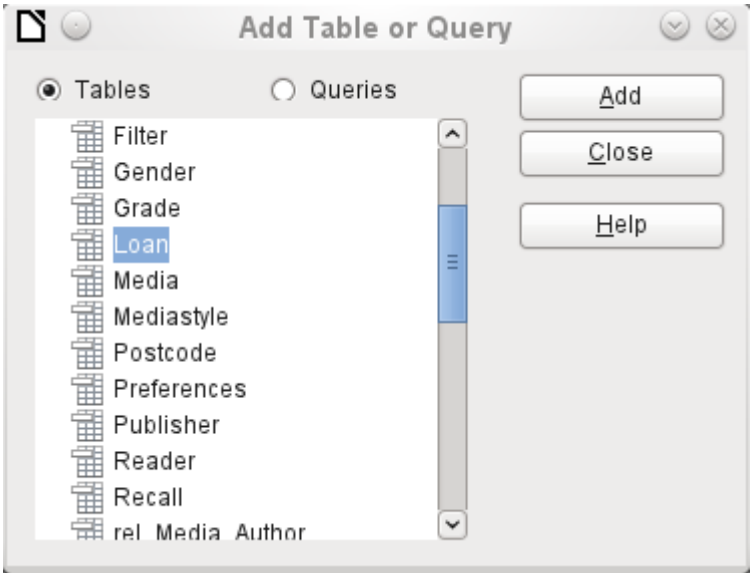
Queries can be entered both in the GUI and directly as SQL code. In both cases a window opens, where you can create a query and also correct it if necessary.

Creating queries using the Query Design dialog

The creation of queries using the Wizard is briefly described in Chapter 8 of the *Getting Started* guide, Getting Started with Base. Here we shall explain the direct creation of queries in Design View.

In the main database window, click the Queries icon in the Databases section, then in the Tasks section, click *Create Query in Design View*. Two dialogs appear. One provides the basis for a design-view creation of the query; the other serves to add tables, views, or queries to the current query.

As our simple form refers to the Loan table, we will first explain the creation of a query using this table.



From the tables available, select the Loan table. This window allows multiple tables (and also views and queries) to be combined. To select a table, click its name and then click the **Add** button. Or, double-click the table's name. Either method adds the table to the graphical area of the Query Design dialog.

When all necessary tables have been selected, click the **Close** button. Additional tables and queries can be added later if required. However no query can be created without at least one table, so a selection must be made at the beginning.

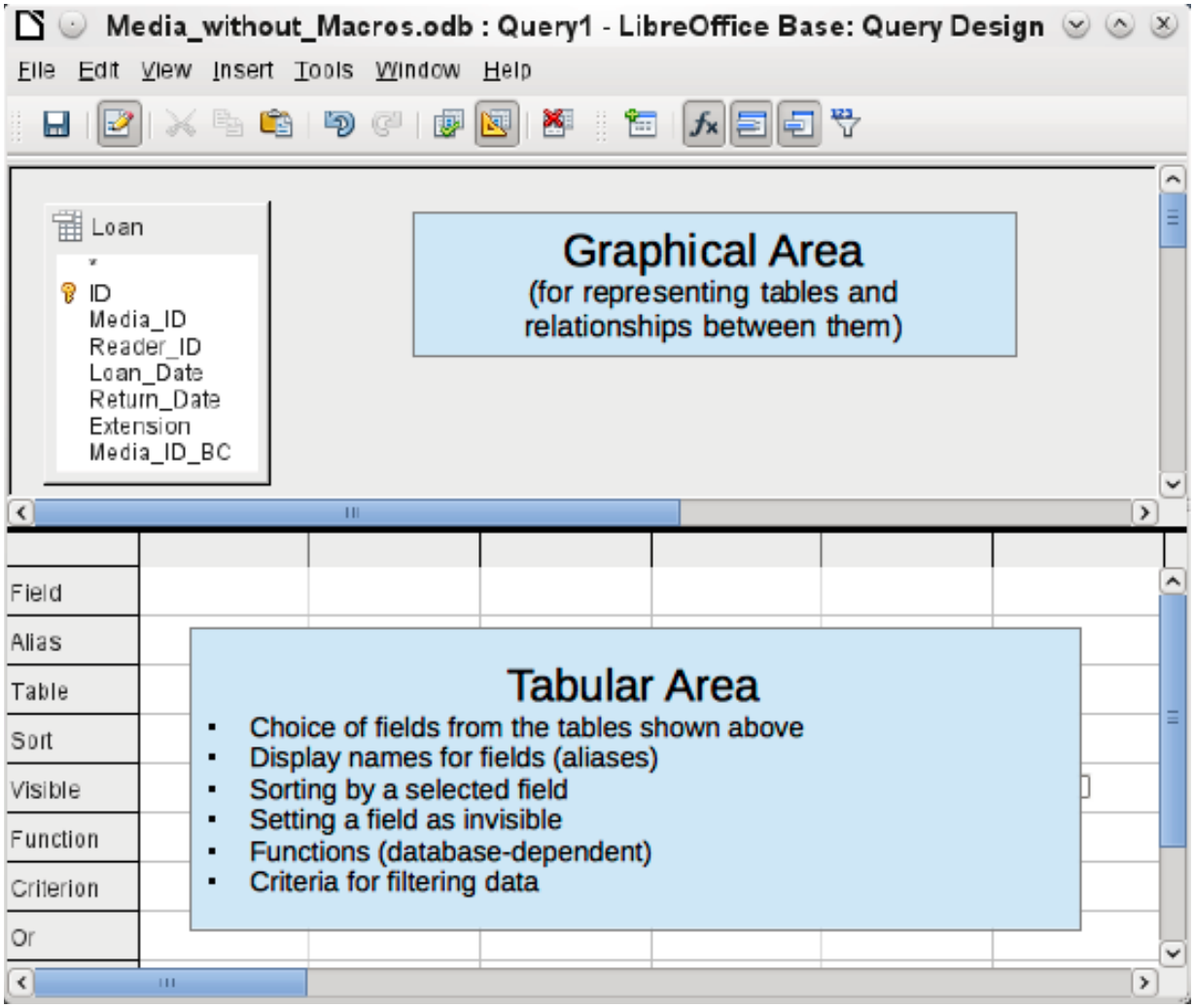
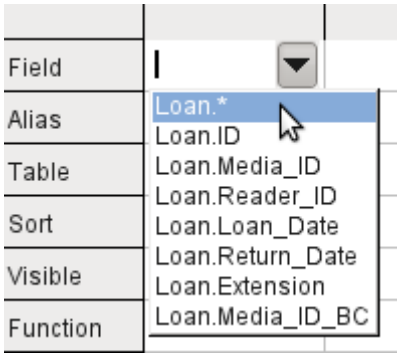


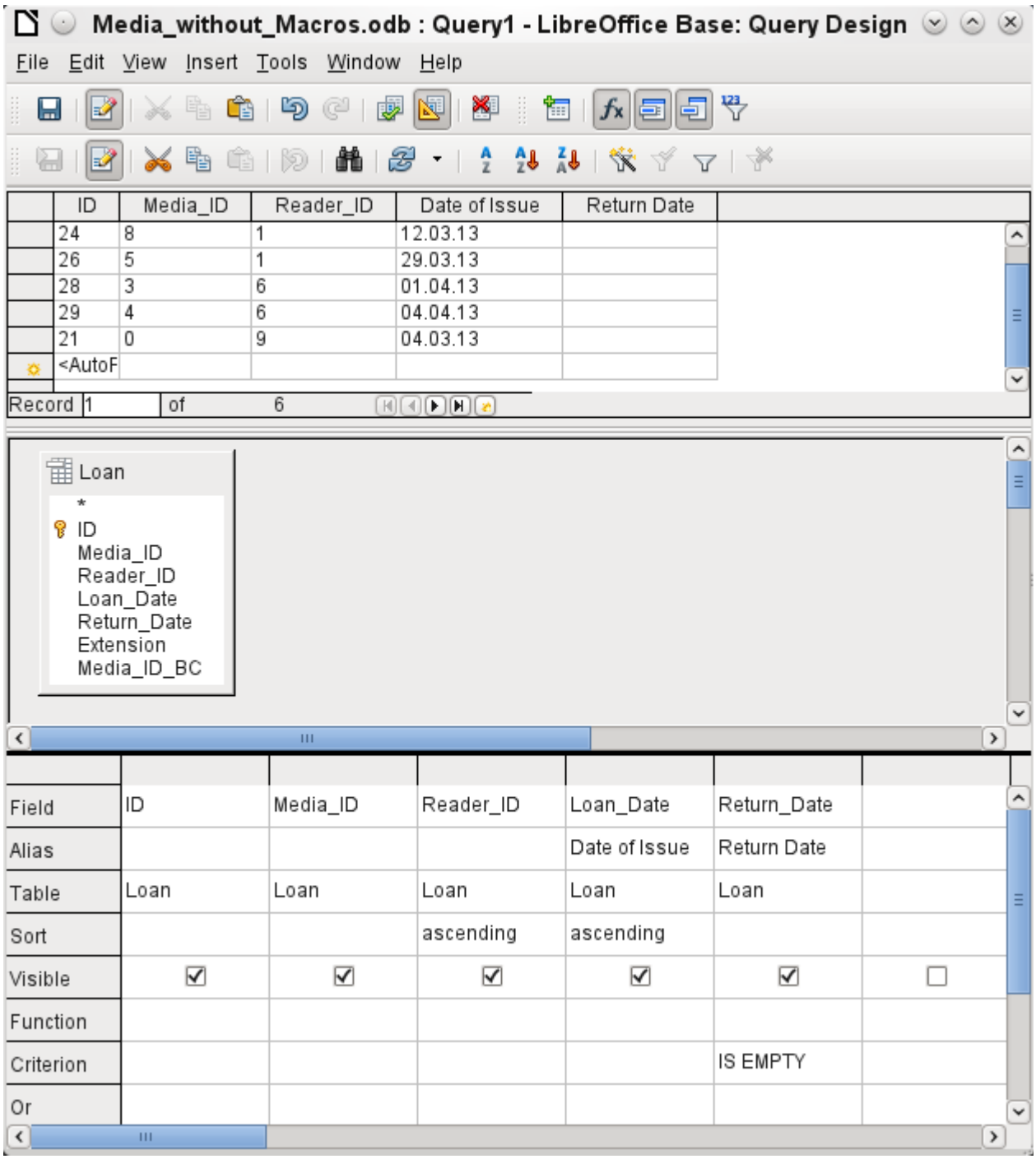
Figure 1: Areas of the Query Design dialog

Figure 1 shows the basic divisions of the Query Design dialog: the graphical area displays the tables that are to be linked to the query. Their relationships to each other in relation to the query may also be shown. The table area is for the selection of fields for display, or for setting conditions related to these fields.

Click on the field in the first column in the table area to reveal a down arrow. Click this arrow to open the drop-down list of available fields. The format is **`Table_name.Field_name`** – which is why all field names here begin with the word *Loan*.



The selected field designation **`Loan.*`** has a special meaning. Here one click allows you to add all fields from the underlying table to the query. When you use this field designation with the *wildcard* * for all fields, the query becomes indistinguishable from the table.



The first five fields of the Loan table are selected. Queries in Design Mode can always be run as tests. This causes a tabular view of the data to appear above the graphical view of the Loan table with its list of fields. A test run of a query is always useful before saving it, to clarify for the user whether the query actually achieves its goal. Often a logical error prevents a query from retrieving any data at all. In other cases it can happen that precisely those records are displayed that you wished to exclude.

In principle a query that produces an error message in the underlying database cannot be saved until the error is corrected.

	ID	Media_ID
▶	22	2
	24	8
	26	5
	28	3
	29	4
	21	0
✱	<AutoF	

Figure 2: An editable query

	Media_ID	Rea
▶	2	1
	8	1
	5	1
	3	6
	4	6
	0	9

Figure 3: A non-editable query

In the above test, special attention should be paid to the first column of the query result. The active record marker (green arrow) always appears on the left side of the table, here pointing to the first record as the active record. While the first field of the first record in Figure 2 is highlighted, the corresponding field in Figure 3 shows only a dashed border. The highlight indicates that this field can be modified. The records, in other words, are editable. The dashed border indicates that this field cannot be modified. Figure 2 also contains an extra line for the entry of a new record, with the ID field already marked as <*AutoField*>. This also shows that new entries are possible.

Tip

A basic rule is that no new entries are possible if the primary key in the queried table is not included in the query.

Field	ID	Media_ID	Reader_ID	Loan_Date	Return_Date
Alias				Date of Issue	Return Date

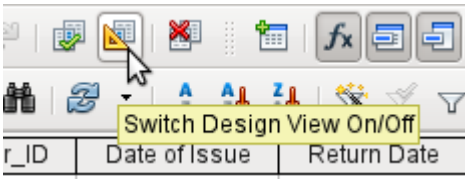
The Loan_Date and Return_Date fields are given aliases. This does not cause them to be renamed but only to appear under these names for the user of the query.

	ID	Media_ID	Reader_ID	Date of Issue	Return Date
▶	22	2	1	04.03.13	

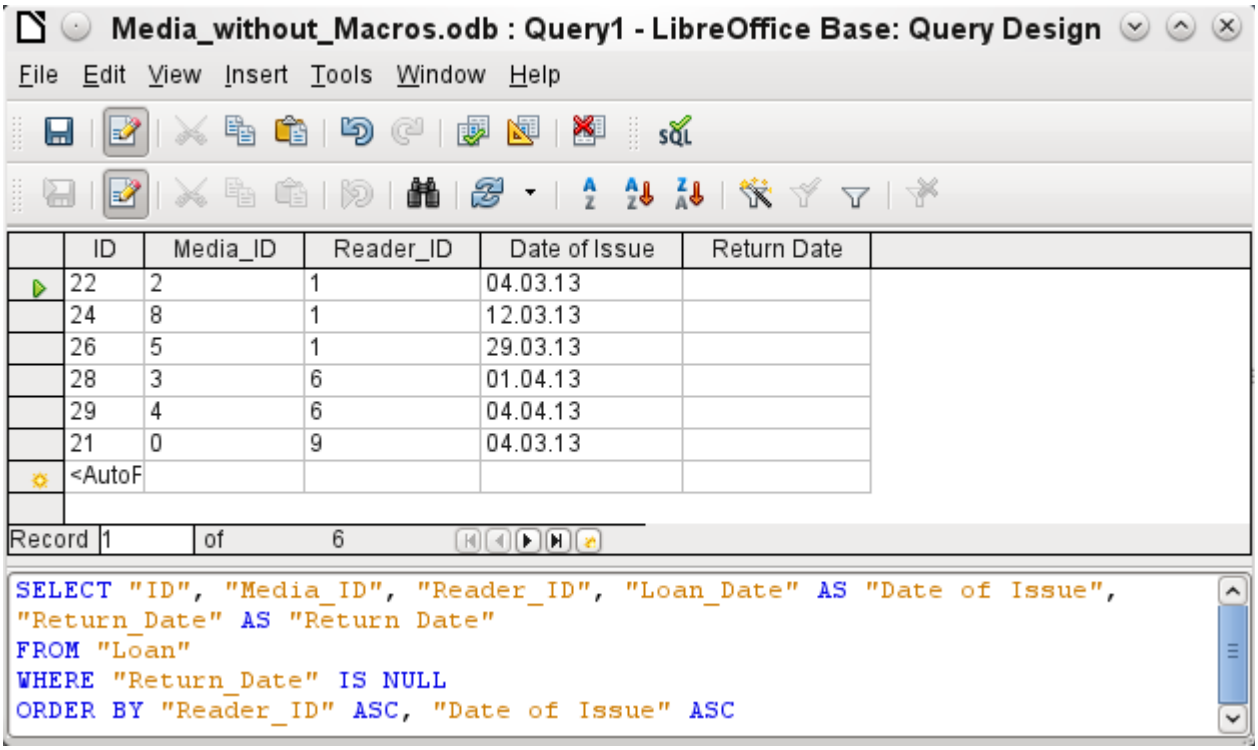
The table view above shows how the aliases replace the actual field names.

Return_Date
Return Date
Loan
☑
IS EMPTY

The Return_Date field is given not just an alias but also a search criterion, which will cause only those records to be displayed for which the Return_Date field is empty. (Enter *IS EMPTY* in the Criterion row of the Return_Date field.) This exclusion criterion will cause only those records to be displayed that relate to media that have not yet been returned from loan.



To learn the SQL language better, it is worth switching from time to time between Design Mode and SQL Mode.

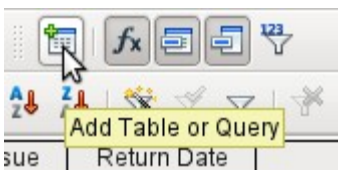


Here the SQL formula created by our previous choices is revealed. To make it easier to read, some line breaks have been included. Unfortunately the editor does not store these line breaks, so when the query is called up again, it will appear as a single continuous line breaking at the window edge.

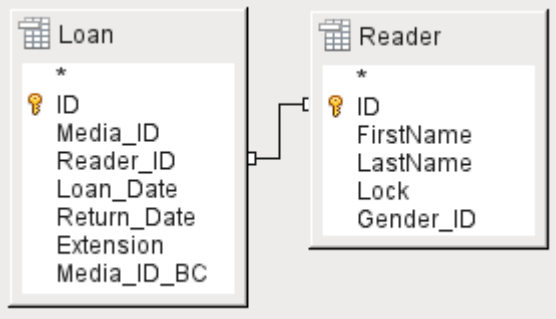
SELECT begins the selection criteria. *AS* specifies the field aliases to be used. *FROM* shows the table which is to be used as the source of the query. *WHERE* gives the conditions for the query, namely that the Return_date field is to be empty (*IS NULL*). *ORDER BY* defines the sort criteria, namely ascending order (ASC – ascending) for the two fields Reader_ID and Loan date. This sort specification illustrates how the alias for the Loan_Date field can be used within the query itself.

Tip	When working in Design View Mode, use *IS EMPTY* to require a field be empty. When working in SQL Mode, use *IS NULL* which is what SQL (Structured Query Language) requires. When you want to sort by descending order using SQL, use *DESC* instead of *ASC*.

So far the Media_ID and Reader_ID fields are only visible as numeric fields. The readers' names are unclear. To show these in a query, the Reader table must be included. For this purpose we return to Design Mode. Then a new table can be added to the Design view.



Here further tables or queries can subsequently be added and made visible in the graphical user interface. If links between the tables were declared at the time of their creation (see Chapter 3, Tables), then these tables are shown with the corresponding direct links.



If a link is absent, it can be created at this point by dragging the mouse from "Loan"."Reader_ID" to "Reader"."ID".

Now fields from the Reader table can be entered into the tabular area. The fields are initially added to the end of the query.

Reader_ID	Loan_Date	Return_Date	FirstName	LastName
	Date of Issue	Return Date		
Loan	Loan	Loan	Reader	Reader

The position of the fields can be corrected in the tabular area of the editor using the mouse. So for example, the First_name field has been dragged into position directly before the Loan_date field.

	ID	Media_ID	Reader_ID	FirstName	LastName	Date of Issue	Return Date
▶	22	2	1	Heinrich	Müller	04.03.13	
	24	8	1	Heinrich	Müller	12.03.13	
	26	5	1	Heinrich	Müller	29.03.13	
	28	3	6	Greta	Garbo	01.04.13	
	29	4	6	Greta	Garbo	04.04.13	
	21	0	9	Terence	Nobody	04.03.13	

Record 1 of 6

Now the names are visible. The Reader_ID has become superfluous. Also sorting by Surname and First_name makes more sense than sorting by Reader_ID.

This query is no longer suitable for use as a query that allows new entries into the resulting table, since it lacks the primary key for the added Reader table. Only if this primary key is built in, does the query become editable again. In fact it then becomes completely editable so that the readers' names can also be altered. For this reason, making query results editable is a facility that should be used with extreme caution, if necessary under the control of a form.

Caution

Having a query that you can edit can create problems. Editing data in the query also edits data in the underlying table and the records contained in the table. The data may not have the same meaning. For example, change the name of the reader, and you have also changed what books the reader has borrowed and returned.

If you have to edit data, do so in a form so you can see the effects of editing data.

Even when a query can be further edited, it is not so easy to use as a form with list boxes, which show the readers' names but contain the Reader_ID from the table. List boxes cannot be added to a query; they are only usable in forms.

```
SELECT "Loan"."ID", "Loan"."Media_ID", "Loan"."Reader_ID",
"Reader"."FirstName", "Reader"."LastName", "Loan"."Loan_Date" AS "Date of
Issue", "Loan"."Return_Date" AS "Return Date"
FROM "Loan", "Reader"
WHERE "Loan"."Reader_ID" = "Reader"."ID" AND "Loan"."Return_Date" IS NULL
ORDER BY "Loan"."Reader_ID" ASC, "Date of Issue" ASC
```

If we now switch back to SQL View, we see that all fields are now shown in double quotes: **"Table_name"."Field_name"**. This is necessary so that the database knows from which table the previously selected fields come from. After all, fields in different tables can easily have the same field names. In the above table structure this is particularly true of the ID field.

Note

The following query works without putting table names in front of the field names:

```
SELECT "ID", "Number", "Price" FROM "Stock", "Dispatch"
WHERE "Dispatch"."stockID" = "Stock"."ID"
```

Here the ID is taken from the table which comes first in the FROM definition. The table definition in the WHERE Formula is also superfluous, because stockID only occurs once (in the Dispatch table) and ID was clearly taken from the Stock table (from the position of the table in the query).

If a field in the query has an alias, it can be referred to – for example in sorting – by this alias without a table name being given. Sorting is carried out in the graphical user interface according to the sequence of fields in the tabular view. If instead you want to sort first by "Loan date" and then by "Loan"."Reader_ID", that can be done if:

- The sequence of fields in the table area of the graphical user interface is changed (drag and drop "Loan date" to the left of "Loan"."Reader_ID", or
- An additional field is added, set to be invisible, just for sorting (however, the editor will register this only temporarily if no alias was defined for it) [add another "Loan date" field just before "Loan"."Reader_ID" or add another "Loan"."Reader_ID" field just after "Loan date"], or
- The text for the ORDER BY command in the SQL editor is altered correspondingly (ORDER BY "Loan date", "Loan"."Reader_ID").

Specifying the sort order *may not be completely error-free*, depending on the LibreOffice version. From version 3.5.3, sorting from the SQL view is correctly registered and displayed in the graphical user interface, including the fields that are used in the query but are not visible in the query output. (These fields do not have a check in the Visible row.)

Tip

A query may require a field that is not part of the query output. In the graphic in the next section, Return_Date is an example. This query is searching for records that do **not** contain a return date. This field provides a criterion for the query but no useful visible data.

Using functions in a query

The use of functions allows a query to provide more than just a filtered view of the data in one or more tables. The following query calculates how many media have been loaned out, depending on the Reader_ID.

Field	ID	Reader_ID	Return_Date
Alias	Count		
Table	Loan	Loan	Loan
Sort			
Visible	☑	☑	☐
Function	Count	Group	
Criterion			IS EMPTY

For the ID of the Loan table, the *Count* function is selected. In principle it makes no difference which field of a table is chosen for this. The only condition is: *The field must not be empty in any of the records*. For this reason, the primary key field, which is never empty, is the most suitable choice. All fields with a content other than NULL are counted.

For the Reader_ID, which gives access to reader information, the *Grouping* function is chosen. In this way, the records with the same Reader_ID are grouped together. The result shows the number of records for each Reader_ID.

As a search criterion, the Return_Date is set to “IS EMPTY”, as in the previous example. (Below, the SQL for this is *WHERE “Return_Date” IS NULL*.)

	Count	Reader_ID
▶	1	9
	3	1
	2	6

Record 1 of 3

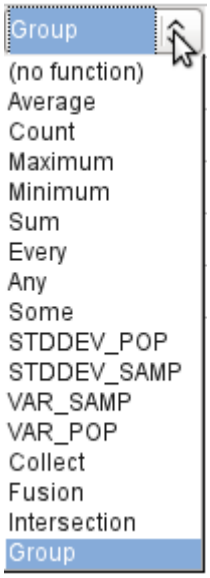
```
SELECT COUNT( "ID" ) AS "Count", "Reader_ID"
FROM "Loan"
WHERE "Return_Date" IS NULL
GROUP BY "Reader_ID"
```

The result of the query shows that Reader_ID '0' has a total of 3 media on loan. If the *Count* function had been assigned to the Return_Date instead of the ID, every Reader_ID would have '0' media on loan, since Return_date is predefined as NULL.

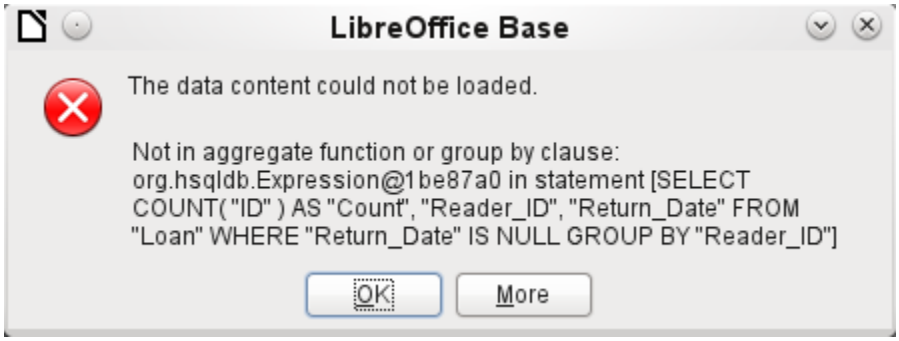
The corresponding Formula in SQL code is shown above.

Altogether the graphical user interface provides the following functions, which correspond to functions in the underlying HSQLDB.

For an explanation of the functions, see “Query enhancement using SQL Mode“ on page 16.



If one field in a query is associated with a function, all the remaining fields mentioned in the query must also be associated with functions if they are to be displayed. If this is not ensured, you get the following error message:



A somewhat free translation would be: The following expression contains no aggregate function or grouping.

Tip	When using Design View Mode, a field is only visible if the *Visible* row contains a check mark for the field. When using SQL Mode, a field is only visible when it follows the keyword, SELECT.

Note	When a field is **not** associated with a function, the number of rows in the query output is determined by the search conditions. When a field is associated with a function, the number of rows in the query output is determined by whether there is any grouping or not. If there is no grouping, there is only one row in the query output. If there is grouping, the number of rows matches the number of distinct values that the grouping field has. So, all of the visible fields must either be associated with a function or **not be associated with a function** to prevent this conflict in the query output. After this, the complete query is listed in the error message, but unfortunately without the offending field being named specifically. In this case the field Return_Date has been added as a displayed field. This field has no function associated with it and is not included in the grouping statement either. The information provided by using the More button is not very illuminating for the normal database user. It just displays the SQL error code.

To correct the error, remove the check mark in the Visible row for the Return_Date field. Its search condition (Criterion) is applied when the query is run, but it is not visible in the query output.

Using the GUI, basic calculations and additional functions can be used.

Field	ID	Media_ID	Reader_ID	Date	Count("Recall"."Date") * 2	Return_Date
Alias				RecallCount	RecallAmount	
Table	Loan	Loan	Loan	Recall		Loan
Sort						
Visible	☑	☑	☑	☑	☑	☐
Function	Group	Group	Group	Count		
Criterion						IS EMPTY

Suppose that a library does not issue recall notices when an item is due for return, but issues overdue notices in cases where the loan period has expired and the item has not been returned. This is common practice in school and public libraries that issue loans only for short, fixed periods. In this case the issue of an overdue notice automatically means that a fine must be paid. How do we calculate these fines?

In the query shown above, the Loan and Recalls tables are queried jointly. From the count of the data entries in the table Recalls, the total number of recall notices is determined. The fine for overdue media is set in the query to 2.00 €. Instead of a field name, the field designation is given as **`Count(Recalls.Date)*2`**. The graphical user interface adds the quotation marks and converts the term “count” into the appropriate SQL command.

Caution

Only for people who use a comma for their decimal separator:

If you wish to enter numbers with decimal places using the graphical user interface, you must ensure that a decimal point rather than a comma is used to separate the decimal places within the final SQL statement. Commas are used as field separators, so new query fields are created for the decimal part.

An entry with a comma in the SQL view always leads to a further field containing the numerical value of the decimal part.

	ID	Media_ID	Reader_ID	RecallCount	RecallAmount
▶	24	8	1	1	2
	22	2	1	1	2

Record 1 of 2

```
SELECT "Loan"."ID", "Loan"."Media_ID", "Loan"."Reader_ID",
COUNT( "Recall"."Date" ) AS "RecallCount",
COUNT( "Recall"."Date" ) * 2 AS "RecallAmount"
FROM "Recall", "Loan"
WHERE "Recall"."Loan_ID" = "Loan"."ID"
AND "Loan"."Return_Date" IS NULL
GROUP BY "Loan"."ID", "Loan"."Media_ID", "Loan"."Reader_ID"
```

The query now yields for each medium still on loan the fines that have accrued, based on the recall notices issued and the additional multiplication field. The following query structure will also be useful for calculating the fines due from individual users.

Field	Reader_ID	Date	Count("Recall"."Date") * 2	Return_Date
Alias		RecallCount	RecallAmount	
Table	Loan	Recall		Loan
Sort				
Visible	☑	☑	☑	☐
Function	Group	Count		
Criterion				IS EMPTY

The "Loan"."ID" and "Loan"."Media_ID" fields have been removed. They were used in the previous query to create by grouping a separate record for each medium. Now we will be grouping only by the reader. The result of the query looks like this:

	Reader_ID	RecallCount	RecallAmount
▶	1	2	4

Record 1 of 1

Instead of listing the media for Reader_ID = 0 separately, all the "Recalls"."Date" fields have been counted and the total of 8.00€ entered as the fine due.

Relationship definition in the query

When data is searched for in tables or forms, the search is usually limited to one table or one form. Even the path from a main form to a subform is not navigable by the built-in search function. For such purposes, the data to be searched for are better collected by using a query.

	Title
▶	Der kleine Hobbit
	Das sogennante Böse
	Eine kurze Geschichte der Zeit
	Traditionelle und kritische Theorie
	Die neue deutsche Rechtschreibung
	I hear you knocking
	Datenbanken mit OpenOffice.org 3
	Das Postfix-Buch
	Im Augenblick

Record 1 of 9

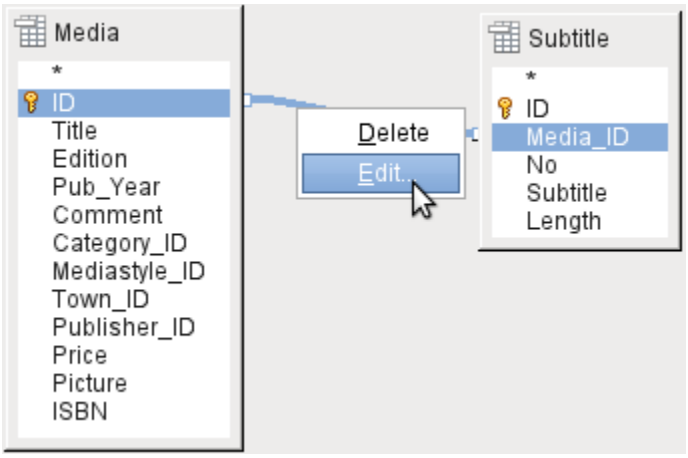
Field	Title	
Alias		
Table	Media	
Sort		
Visible	☑	☐

	Title	Subtitle
▶	I hear you knocking	Youn can't catch me
	I hear you knocking	The stumble
	I hear you knocking	Sabre dance (Single version)
	Im Augenblick	Amsterdam
	Im Augenblick	Hier unten am Deich
	Im Augenblick	Köln-Ehrenfeld
	Im Augenblick	Bei Mir
	Im Augenblick	Gott sei Dank

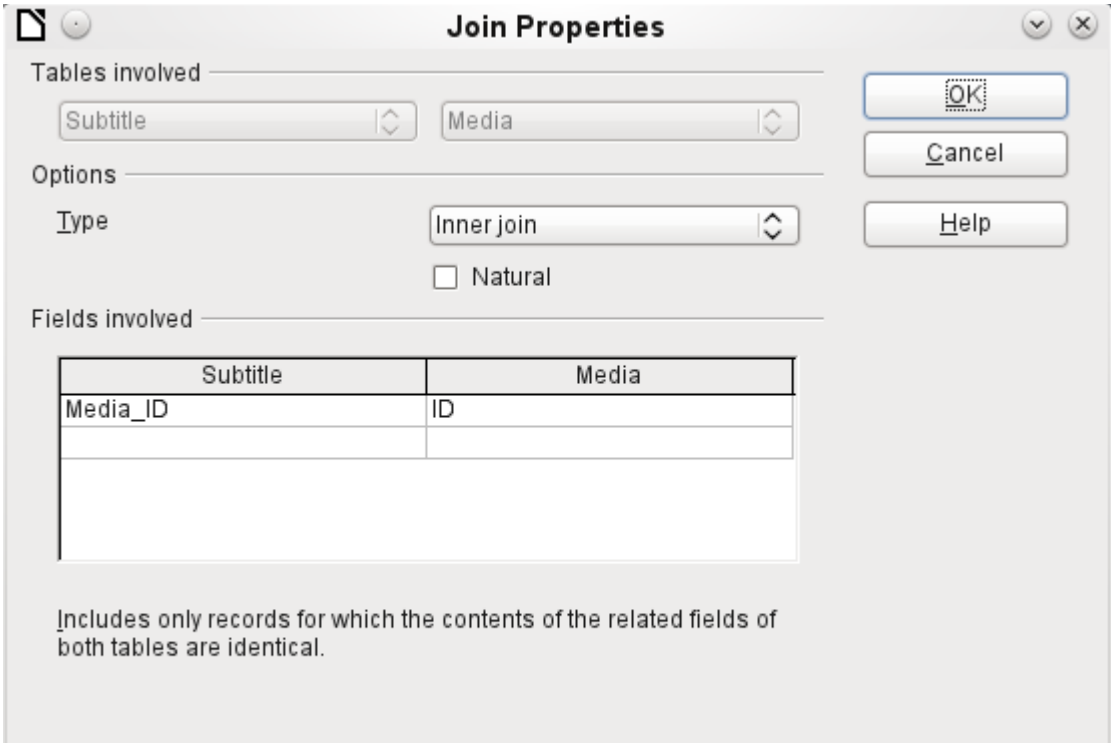
Record 1 of 8

Field	Title	Subtitle	
Alias			
Table	Media	Subtitle	
Sort			
Visible	☑	☑	☐

The simple query for the Title field from the Media table shows the test entries for this table, 9 records in all. But if you enter Subtitle into the query table, the record content of the Media table is reduced to only 2 Titles. Only for these two Titles are there also Subtitles in the table. For all the other Titles, no subtitles exist. This corresponds to the join condition that only those records for which the Media_ID field in the Subtitle table is equal to the ID field in the Media table should be shown. All other records are excluded.



The join conditions must be opened for editing to display all the desired records. We refer here **not** to joins between tables in *Relationship design* **but** to joins within *queries*.



By default, relationships are set as *Inner Joins*. The window provides information on the way this type of join works in practice.

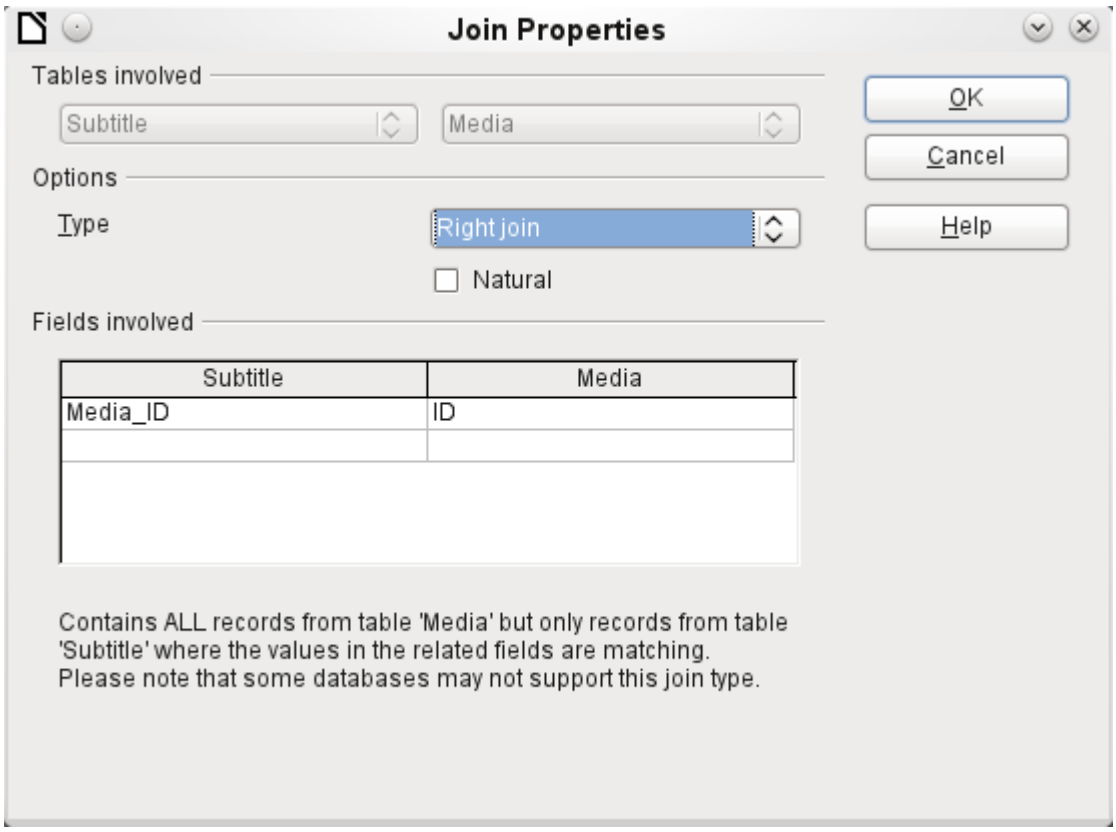
The two previously selected tables are listed as Tables Involved. They are not selectable here. The relevant fields from the two tables are read from the table definitions. If there is no relationship specified in the table definition, one can be created at this point for the query. However, if you have planned your database in an orderly manner using HSQLDB, there should be no need to alter these fields.

The most important setting is the *Join* option. Here relationships can be so chosen that all records from the Subtitle table are selected, but only those records from Media which have a subtitle entered in the Subtitle table.

Or you can choose the opposite: that in any case all records from the table Media are displayed, regardless of whether they have a subtitle.

The *Natural* option specifies that the linked fields in the tables are treated as equal. You can also avoid having to use this setting by defining your relationships properly at the very start of planning your database.

For the type *Right join*, the description shows that all records from the Media table will be displayed (Subtitle RIGHT JOIN Media). As there is no Subtitle that lacks a title in Media but there are certainly Titles in Media that lack a Subtitle, this is the right choice.



After confirming the *right join* the query results look as we wanted them. Title and Subtitle are displayed together in one query. Naturally Titles appear more than once as with the previous relationship. However as long as hits are not being counted, this query can be used further as a basis for a search function. See the code fragments in this chapter, in Chapter 9 (Macros), and in Chapter 8 (Database Tasks).

Title	Subtitle
Der kleine Hobbit	
Das sogenannte Böse	
Eine kurze Geschichte der Zeit	
Traditionelle und kritische Theorie	
Die neue deutsche Rechtschreibung	
I hear you knocking	Youn can't catch me
I hear you knocking	The stumble
I hear you knocking	Sabre dance (Single version)
Datenbanken mit OpenOffice.org 3	
Das Postfix-Buch	
Im Augenblick	Amsterdam
Im Augenblick	Hier unten am Deich
Im Augenblick	Köln-Ehrenfeld
Im Augenblick	Bei Mir
Im Augenblick	Gott sei Dank

Record 1 of 15

Query enhancement using SQL Mode

If during graphical entry you use **View > Switch Design View On/Off** to switch design view off, you see the SQL command for what previously appeared in Design View. This is the best way for beginners to learn the Standard Query Language for Databases. Sometimes it is also the only way

to enter a query into the database when the GUI cannot translate your requirements into the necessary SQL commands.

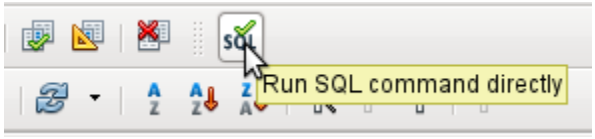
```
SELECT * FROM "Table_name"
```

This will show everything that is in the Table_name table. The "*" represents all the fields of the table.

```
SELECT * FROM "Table_name" WHERE "Field_name" = 'Karl'
```

Here there is a significant restriction. Only those records are displayed for which the field Field_name contains the term 'Karl' – the exact term, not for example 'Karl Egon'.

Sometimes queries in Base cannot be carried out using the GUI, as particular commands may not be recognized. In such cases it is necessary to switch Design View off and use **Edit > Run SQL command directly** for direct access to the database. This method has the disadvantage that you can only work with the query in SQL Mode.



Direct use of SQL commands is also accessible using the graphical user interface, as the above figure shows. Click the icon highlighted (*Run SQL command directly*) to turn the Design View Off/On icon off. Now when you click the *Run* icon, the query runs the SQL commands directly.

Here is an example of the extensive possibilities available for posing questions to the database and specifying the type of result required:

```
SELECT [{LIMIT <offset> <limit> | TOP <limit>}][ALL | DISTINCT]
{ <Select-Formulation> | "Table_name".* | * } [, ...]
[INTO [CACHED | TEMP  | TEXT] "new_Table"]
FROM "Table_list"
[WHERE SQL-Expression]
[GROUP BY SQL-Expression [, ...]]
[HAVING SQL-Expression]
[{ UNION [ALL | DISTINCT] | {MINUS [DISTINCT] | EXCEPT [DISTINCT] } |
INTERSECT [DISTINCT] } Query statement]
[ORDER BY Order-Expression [, ...]]
[LIMIT <limit> [OFFSET <offset>]];
```

[{LIMIT <offset> <limit> | TOP <limit>}]:

This limits the number of records to be displayed. **`LIMIT 10 20`** starts at the 11th record and shows the following 20 records. **`TOP 10`** always shows the first 10 records. This is the same as **`LIMIT 0 10`**. **`LIMIT 10 0`** omits the first 10 records and displays all records starting from the 11th.

You can do the same thing by using the last SELECT condition in the formula **`[LIMIT <limit> [OFFSET <offset>]]`**. **`LIMIT 10`** shows only 10 records. Adding **`OFFSET 20`** causes the display to begin at the 21st record. Before you use this final form of display limitation, you need to include a sorting condition (ORDER BY …).

Limiting the display of query results is only possible using direct SQL commands. By doing this, you determine how the query data is sorted and hence what data will be shown.

[ALL | DISTINCT]

`SELECT ALL` is the default. All records are displayed that fulfill the search conditions. Example: **`SELECT ALL "Name" FROM "Table_name"`** yields all names; if "Peter" occurs three times and "Egon" four times in the table, these names are displayed three and four times respectively. **`SELECT DISTINCT "Name" FROM "Table_name"`** suppresses query results

that have the same content. In this case, ‘Peter’ and ‘Egon’ occur only once. **DISTINCT** refers to the whole of the record that is accessed by the query. So if, for example, the surname is asked for as well, records for ‘Peter Müller’ and ‘Peter Maier’ will count as distinct. Even if you specify the **DISTINCT** condition, both will be displayed.

<Select-Formulation>

```
{ Expression | COUNT(*) |
{ COUNT | MIN | MAX | SUM | AVG | SOME | EVERY | VAR_POP | VAR_SAMP |
STDDEV_POP | STDDEV_SAMP }
([ALL | DISTINCT]] Expression) } [[AS] "display_name"]
```

Field names, calculations, record totals are all possible entries. In addition different functions are available for the field shown. Except for **COUNT(*)** (which counts all the records) none of these functions access **NULL** fields.

```
COUNT | MIN | MAX | SUM | AVG | SOME | EVERY | VAR_POP | VAR_SAMP |
STDDEV_POP | STDDEV_SAMP
```

COUNT("Name") counts all entries for the field Name.
MIN("Name") shows the first name alphabetically. The result of this function is always formatted just as it occurs in the field. Text is shown as text, integers as integers, decimals as decimals and so on.
MAX("Name") shows the last name alphabetically.
SUM("Number") can add only the values in numerical fields. The function fails for date fields.
AVG("Number") shows the average of the contents of a column. This function too is limited to numerical fields.
SOME("Field_Name"), EVERY("Field_Name"): Fields used with these functions must have the Yes/No [BOOLEAN] field type (contains only 0 or 1). Furthermore, they produce a summary of the field content to which they are applied.

SOME returns *TRUE* (or *1*) if at least one entry for the field is *1*, and it returns *FALSE* (or *0*) only if all the entries are *0*. **EVERY** returns *1* only if every entry for the field is *1*, and returns *FALSE* if at least one entry is *0*.

Tip	The Boolean field type is Yes/No[BOOLEAN]. However, this field contains only 0 or 1. In query search conditions, use either *TRUE*, *1*, *FALSE* or *0*. For the *Yes* condition, you can use either *TRUE* or *1*. For the *No* condition, use either *FALSE* or *0*. If you try to use either “Yes” or “No” instead, you get an error message. Then you will have to correct your error.

Example:

```
SELECT "Class", EVERY("Swimmer") FROM "Table1" GROUP BY "Class";
```

Class contains the names of the swimming class. *Swimmer* is a Boolean field describing whether a student can swim or not (1 or 0). *Students* contains the names of the students. *Table1* contains these fields: its primary key, Class, Swimmer, and Students. Only Class and Swimmer are needed for this query.

Because the query is grouped by the entries of the field Class, EVERY will return a value for the field, Swimmer, for each class. When every person in a swimming class can swim, EVERY returns TRUE. Otherwise EVERY returns FALSE because at least one student of the class can not swim. Since the output for the Swimmer field is a checkbox, A check mark indicates TRUE, and no check mark indicates FALSE.

VAR_POP | VAR_SAMP | STDDEV_POP | STDDEV_SAMP are statistical functions and affect only integer and decimal fields.
All these functions return 0, if the values within the group are all equal.

The statistical functions do not allow the **DISTINCT** limitation to be used. Basically they calculate over all values covered by the query, whereas **DISTINCT** excludes records with the same values from the display.

[AS] "display_name": The fields can be given a different designation (alias) within the query.

"Table_name".* | * [, ...]

Each field to be displayed is given with its field names, separated by commas. If fields from several tables are entered into the query, a combination of the field name with the table name is necessary: **"Table_name"."Field_name"**.

Instead of a detailed list of all the fields of a table, its total content can be displayed. For this you use the symbol "*". It is then unnecessary to use the table name, if the results will only apply to the one table. However, if the query includes all of the fields of one table and at least one field from a second table, use:
"Table_name 1".*, "Table_name 2"."Field_name".

[INTO [CACHED | TEMP | TEXT] "new_table"]

The result of this query is to be written directly into a new table which is named here. The field properties for the new table are defined from the field definitions contained in the query. Writing into a new table does not work from SQL Mode as this handles only displayable results. Instead you must use **Tools > SQL**. The resultant table is initially not editable as it lacks a primary key.

FROM <Table_list>

```
"Table_name 1" [{CROSS | INNER | LEFT OUTER | RIGHT OUTER} JOIN
"Table_name 2" ON Expression] [, ...]
```

The tables which are to be jointly searched are usually in a list separated by commas. The relationship of the tables to one another is then additionally defined by the keyword **WHERE**.

If the tables are bound through a **JOIN** rather than a comma, their relationship is defined by the term beginning with **ON** which occurs directly after the second table.

A simple **JOIN** has the effect that only those records are displayed for which the conditions in both the tables apply.

Example:

SELECT "Table1"."Name", "Table2"."Class" FROM "Table1", "Table2" WHERE "Table1"."ClassID" = "Table2"."ID"

is equivalent to:

```
SELECT "Table1"."Name", "Table2"."Class" FROM "Table1" JOIN "Table2"
ON "Table1"."ClassID" = "Table2"."ID"
```

Here the names and the corresponding classes are displayed. If a name has no class listed for it, that name is not included in the display. If a class has no names, it is also not displayed. The addition of **INNER** does not alter this.

```
SELECT "Table1"."Name", "Table2"."Class" FROM "Table1" LEFT JOIN
"Table2" ON "Table1"."ClassID" = "Table2"."ID"
```

If **LEFT** is added, all "Names" from "Table1" are displayed even if they have no "Class". If, on the contrary, **RIGHT** is added, all Classes are displayed even if they have no names in them. Addition of **OUTER** need not be shown here. (Right Outer Join is the same thing as Right Join; Left Outer Join is the same thing as Left Join.)

```
SELECT "Table1"."Player1", "Table2"."Player2" FROM "Table1" AS
"Table1" CROSS JOIN "Table2" AS "Table1" WHERE "Table1"."Player1" <>
"Table2"."Player2"
```

A **CROSS JOIN** requires the table to be supplied with an alias, but the addition of the term **ON** is not always necessary. All records from the first table are paired with all records from the second table. Thus the above query yields all possible pairings of records from the first table with those of the second table except for pairings between records for the same player. In the case of a **CROSS JOIN,** the condition must not include a link between the tables specified in the **ON** term. Instead, **WHERE** conditions can be entered. If the conditions are formulated exactly as in the case of a simple JOIN, you get the same result:

```
SELECT "Table1"."Name", "Table2"."Class" FROM "Table1" JOIN "Table2"
ON "Table1"."ClassID" = "Table2"."ID"
```

gives the same result as

```
SELECT "Table1"."Name", "Table2"."Class" FROM "Table1" AS "Table1"
CROSS JOIN "Table2" AS "Table2" WHERE "Table1"."ClassID" =
"Table2"."ID"
```

[WHERE SQL-Expression]

The standard introduction for conditions to request a more accurate filtering of the data. Here too the relationships between tables are usually defined if they are not linked together with JOIN.

[GROUP BY SQL-Expression [, ...]]

Use this when you want to divide the query data into groups before applying the functions to each one of the groups separately. The division is based upon the values of the field or fields contained in the GROUP BY term.

Example:

```
SELECT "Name", SUM("Input"-"Output") AS "Balance" FROM "Table1" GROUP
BY "Name";
```

Records with the same name are summed. In the query result, the sum of **Input – Output** is given for each person. This field is to be called **Balance**. Each row of the query result contains a value from the “Name” table and the calculated balance for that specific value.

Tip	When fields are processed using a particular function (for example **COUNT, SUM** …), all fields that are not processed with a function but should be displayed are grouped together using **GROUP BY**.

[HAVING SQL-Expression]

The **HAVING** formula closely resembles the **WHERE** formula. The difference is that the WHERE formula applies to the values of selected fields in the query. The HAVING formula applies to selected calculated values. Specifically, the WHERE formula can not use an aggregate function as part of a search condition; the HAVING formula does.

The HAVING formula serves two purposes as shown in the two examples below. In the first one, the search condition requires that the minimum run-time be less than 40 minutes. In the second example, the search condition requires that an individual's balance must be positive.

The query results for the first one lists the names of people whose run-time has been less than 40 minutes at least one time and the minimum run-time. People whose run-times have all be greater than 40 minutes are not listed.

The query results for the second one lists the names of people who have a total greater output than input and their balance. People whose balance is 0 or less are not listed.

Examples:

```
SELECT "Name", "Runtime" FROM "Table1" GROUP BY "Name", "Runtime"
HAVING MIN("Runtime") < '00:40:00';
SELECT "Name", SUM("Input"-"Output") AS "Balance" FROM "Table1" GROUP
BY "Name" HAVING SUM("Input"-"Output") > 0;
```

[SQL Expression]

SQL expressions are combined according to the following scheme:

```
[NOT] condition [{ OR | AND } condition]
```

Example:

```
SELECT * FROM "Table_name" WHERE NOT "Return_date" IS NULL AND
"ReaderID" = 2;
```

The records read from the table are those for which a “Return_date” has been entered and the "ReaderID" is 2. In practice this means that all media loaned to a specific person and returned can be retrieved. The conditions are only linked with **AND**. The **NOT** refers only to the first condition.

```
SELECT * FROM "Table_name" WHERE NOT ("Return_date" IS NULL AND
"ReaderID" = 2);
```

Parentheses around the condition, with **NOT** outside them shows only those records that do not fulfill the condition in parentheses completely. This would cover all records, except for those for "ReaderID" number 2, which have not yet been returned.

[SQL Expression]: conditions

```
{ value [|| value]
```

A value can be single or several values joined by two vertical lines **| |**. Naturally this applies to field contents as well.

```
SELECT "Surname" || ', ' || "First_name" AS "Name" FROM "Table_name"
```

The content of the fields "Surname" and "First_name" are displayed together in a field called "Name". Note that a comma and a space are inserted between "Surname" and "First_name".

```
| value { = | < | <= | > | >= | <> | != } value
```

These signs correspond to the well-known mathematical operators:
{ Equal to | Less than | Less than or equal to | Greater than | Greater than or equal to | Not equal to | Not equal to }

```
| value IS [NOT] NULL
```

The corresponding field has no content, because nothing has been written to it. This cannot be determined unambiguously in the GUI, since a visually empty text field does not mean that the field is completely without content. However the default set-up in Base is that empty fields in the database are set to **NULL**.

```
| EXISTS(Query_result)
```

Example:

```
SELECT "Name" FROM "Table1" WHERE EXISTS (SELECT "First_name" FROM
"Table2" WHERE "Table2"."First_name" = "Table1"."Name")
```

The names from Table1 are displayed for which first names are given in Table2.

```
| Value BETWEEN Value AND Value
```

BETWEEN value1 AND value2 yields all values from value1 up to and including value2. If the values are letters, an alphabetic sort is used in which lower-case letters have the same value as the corresponding upper-case ones.

```
SELECT "Name" FROM "Table_name" WHERE "Name" BETWEEN 'A' AND 'E';
```

This query yields all names beginning with A, B, C or D (and also with the corresponding lower-case letters). As E is set as the upper limit, names beginning with E are not included. The letter E itself occurs just *before* the names that begin with E.

```
| value [NOT] IN ( {value [, ...] | Query result } )
```

This requires either a list of values or a query. The condition is fulfilled if the value is included in the value list or the query result.

```
| value [NOT] LIKE value [ESCAPE] value }
```

The **LIKE** operator is one that is needed in many simple search functions. The value is entered using the following pattern:
'%' stands for any number of characters (including 0),
'_' replaces exactly one character.

To search for '%' or '_' itself, the characters must immediately follow another character defined as **ESCAPE**.

```
SELECT "Name" FROM "Table_name" WHERE "Name" LIKE '\_%' ESCAPE '\'
```

This query displays all names that begin with an underscore. '****' is defined here as the **ESCAPE** character.

[SQL Expression]: values

```
[+ | -] { Expression [{ + | - | * | / | || } Expression]
```

The values may have a preceding sign. Addition, subtraction, multiplication, division and concatenation of expressions are allowed. An example of concatenation:

```
SELECT "Surname"||', '||"First_name" FROM "Table"
```

In this way records are displayed by the query with a field containing "Surname, First_name". The concatenation operator can be qualified by the following expressions.

```
| ( Condition )
```

See the previous section for this.

```
| Function ( [Parameter] [,...] )
```

See the section on Functions in the appendix.

The following queries are also referred to as sub-queries (subselects).

```
| Query result which yields exactly one answer
```

As a record can only have one value in each field, only a query which yields precisely one value can be displayed in full.

```
| {ANY|ALL} (Queryresult which yields exactly one answer from a whole column)
```

Often there is a condition that compares an expression with a whole group of values.

Combined with **ANY** this signifies that the expression must occur at least once in the group. This can also be specified using the **IN** condition. **= ANY** yields the same result as **IN**.

Combined with **ALL** it signifies that all values of the group must correspond to the one expression.

[SQL Expression]: Expression

```
{ 'Text' | Integer | Floating-point number
| ["Table".]"Field" | TRUE | FALSE | NULL }
```

Basically values serve as arguments for various expressions, dependent on the source format. To search for the content of text fields, place the content in quotes. Integers are written without

quotes, as are floating-point numbers.
Fields stand for the values that occur in those fields in the table. Usually fields are compared either with each other or with specific values. In SQL, field names should be placed in double quotes, as they may not be correctly recognized otherwise. Usually SQL assumes that text without double quotes is without special characters, that is a single word without spaces and in upper case. If several tables are contained in the query, the table name must be given in addition to the field name, separated from the latter by a period.
TRUE and **FALSE** usually derive from Yes/No fields.
NULL means no content. It is not the same thing as 0 but rather corresponds to “empty”.

UNION [ALL | DISTINCT] Query_result

This links queries so that the content of the second query is written under the first. For this to work, all fields in both queries must match in type. This linkage of several queries functions only in direct SQL command mode.

```
SELECT "First_name" FROM "Table1" UNION DISTINCT SELECT "First_name"
FROM "Table2";
```

This query yields all first names from table1 and Table2; the additional term **DISTINCT** means that no duplicate first names will be displayed. **DISTINCT** is the default in this context. By default the first names are sorted alphabetically in ascending order. **ALL** causes all first names in Table1 to be displayed, followed by the first name in Table2. In this case the default is to sort by primary key.

MINUS [DISTINCT] | EXCEPT [DISTINCT] Query_result

```
SELECT "First_name" FROM "Table1" EXCEPT SELECT "First_name" FROM
"Table2";
```

Shows all first names from table1 except for the first names contained in Table 2. **MINUS** and **EXCEPT** lead to the same result. Sorting is alphabetic.

INTERSECT [DISTINCT] Query_result

```
SELECT "First_name" FROM "Table1" INTERSECT SELECT "First_name" FROM
"Table2";
```

This displays the first names that occur in both tables. Sorting is again alphabetic. At present this only works in direct SQL command mode.

[ORDER BY Ordering-Expression [, ...]]

The expression can be a field name, a column number (beginning with 1 from the left), an alias (formulated with AS `for example`) or a composite value expression (see [SQL Expression]: values). The sort order is usually ascending (**ASC**). If you want a descending sort you must specify **DESC** explicitly.

```
SELECT "First_name", "Surname" AS "Name" FROM "Table1" ORDER BY "Surname";
```

is identical to

```
SELECT "First_name", "Surname" AS "Name" FROM "Table1" ORDER BY 2;
```

is identical to

```
SELECT "First_name", "Surname" AS "Name" FROM "Table1" ORDER BY "Name";
```

Using an alias in a query

Queries can reproduce fields with changed names.

```
SELECT "First_name", "Surname" AS "Name" FROM "Table1"
```

The field *Surname* is called *Name* in the display.

If a query involves two tables, each field name must be preceded by the name of the table:

```
SELECT "Table1"."First_name", "Table1"."Surname" AS "Name",
"Table2"."Class" FROM "Table1", "Table2" WHERE "Table1"."Class_ID" =
"Table2"."ID"
```

The table name can also be given an alias, but this will not be reproduced in table view. If such an alias is set, all the table names in the query must be altered accordingly:

```
SELECT "a"."First_name", "a"."Surname" AS "Name", "b"."Class" FROM
"Table1" AS "a", "Table2" AS "b" WHERE "a"."Class_ID" = "b"."ID"
```

The assignment of an alias for a table can be carried out more briefly without using the term AS:

```
SELECT "a"."First_name", "a"."Surname" AS "Name", "b"."Class" FROM
"Table1" "a", "Table2" "b" WHERE "a"."Class_ID" = "b"."ID"
```

This however makes the code less readable. Because of this, the abbreviated form should be used only in exceptional circumstances.

Queries for the creation of list box fields

List box fields show a value that does not correspond to the content of the underlying table. They are used to display the value assigned by a user to a foreign key rather than the key itself. The value that is finally saved in the form must not occur in the first position of the list box field.

```
SELECT "First_name", "ID" FROM "Table1";
```

This query would show all first names and the primary key "ID" values that the form's underlying table provides. Of course it is not yet optimal. The first names appear unsorted and, in the case of identical first names, it is impossible to determine which person is intended.

```
SELECT "First_name"||' '||"Surname", "ID" FROM "Table1" ORDER BY
"First_name"||' '||"Surname";
```

Now the first name and the surname both appear, separated by a space. The names become distinguishable and are also sorted. But the sort follows the usual logic of starting with the first letter of the string, so it sorts by first name and only then by surname. A different sort order to that in which the fields are displayed would only be confusing.

```
SELECT "Surname"||', '||"First_name", "ID" FROM "Table1" ORDER BY
"Surname"||', '||"First_name";
```

This now leads to a sorting that corresponds better to normal custom. Family members appear together, one under another; however different families with the same surname would be interleaved. To distinguish them, we would need to group them differently within the table.

There is one final problem: if two people have the same surname *and* first name, they will still not be distinguishable. One solution might be to use a name suffix. But imagine how it would look if a salutation read Mr "Müller II"!

```
SELECT "Surname"||', '||"First_name"||' - ID:'||"ID", "ID" FROM
"Table1" ORDER BY "Surname"||', '||"First_name"||"ID";
```

Here all records are distinguishable. What is actually displayed is "Surname, First name – ID:ID value".

In the loan form, there is a list box which only shows the media that have not yet been loaned out. It is created using the following SQL formula:

```
SELECT "Title" || ' - Nr. ' || "ID", "ID" FROM "Media" WHERE "ID" NOT
IN (SELECT "Media_ID" FROM  "Loan" WHERE "Return_Date" IS  NULL) ORDER
BY  "Title" || ' - Nr. ' || "ID"  ASC
```

Queries as a basis for additional information in forms

If you wish a form to display additional Information that would otherwise not be visible, there are various query possibilities. The simplest is to retrieve this information with independent queries and insert the results into the form. The disadvantage of this method is that changes in the records may affect the query result, but unfortunately these changes are not automatically displayed.

Here is an example from the sphere of stock control for a simple checkout.

The checkout table contains totals and foreign keys for stock items, and a receipt number. The shopper has very little information if there is no additional query result printed on the receipt. After all, the items are identified only by reading in a barcode. Without a query, the form shows only:

Total	***Barcode***
3	17
2	24

What is hidden behind the numbers cannot be made visible by using a list box, as the foreign key is input directly using the barcode. In the same way, it is impossible to use a list box next to the item to show at least the unit price.

Here a query can help.

```
SELECT "Checkout"."Receipt_ID", "Checkout"."Total", "Stock"."Item",
"Stock"."Unit_Price", "Checkout"."Total"*"Stock"."Unit_price" AS
"Total_Price" FROM "Checkout", "Item" WHERE "Stock"."ID" =
"Checkout"."Item_ID";
```

Now at least after the information has been entered, we know how much needs to be paid for 3 * Item'17'. In addition only the information relevant to the corresponding Receipt_ID needs to be filtered through the form. What is still lacking is what the customer needs to pay overall.

```
SELECT "Checkout"."Receipt_ID",
SUM("Checkout"."Total"*"Stock"."Unit_price") AS "Sum" FROM "Checkout",
"Stock" WHERE "Stock"."ID" = "Checkout"."Item_ID" GROUP BY
"Checkout"."Receipt_ID";
```

Design the form to show one record of the query at a time. Since the query is grouped by Receipt_ID, the form shows information about one customer at a time.

Data entry possibilities within queries

To make entries into a query, the primary key for the table underlying the query must be present. This also applies to a query that links several tables together.

In the loaning of media, it makes no sense to display for a reader items that have already been returned some time ago.

```
SELECT "ID", "Reader_ID", "Media_ID", "Loan_date" FROM "Loan" WHERE
"Return_Date" IS NULL;
```

In this way a form can show within a table control field everything that a particular reader has borrowed over time. Here too the query must filter using the appropriate form structure (reader in the main form, query in the sub-form), so that only media that are actually on loan are displayed. The query is suitable for data entry since the primary key is included in the query.

The query ceases to be editable, if it consists of more than one table and the tables are accessed via an alias. It makes no difference in that case whether or not the primary key is present in the query.

```
SELECT "Media"."ID", "Media"."Title", "Category"."Category",
"Category"."ID" AS "Kat_ID" FROM "Media", "Category" WHERE
"Media"."Category_ID" = "Category"."ID";
```

This query is editable as both primary keys are included and can be accessed in the tables without using an alias.

```
SELECT "m"."ID", "m"."Title", "Category"."Category", "Category"."ID"
AS "Kat_ID" FROM "Media" AS "m", "Category" WHERE "m"."Category_ID" =
"Category"."ID";
```

In this query the "Media" table is accessed using an alias. The query cannot be edited.

In the above example, this problem is easily avoided. If, however, a correlated subquery (see page 27) is used, you need to use a table alias. A query is only editable in that case if it contains only one table in the main query.

Use of parameters in queries

If you often use the same basic query but with different values each time, queries with parameters can be used. In principle queries with parameters function just like queries for a subform:

```
SELECT "ID", "ReaderID", "Media_ID", "Loan_date" FROM "Loans" WHERE
"Return_Date" IS NULL AND "Reader_ID"=2;
```

This query shows only the media on loan to the reader with the number 2.

```
SELECT "ID", "Reader_ID", "Media_ID", "Loan_date" FROM "Loans" WHERE
"Return_Date" IS NULL AND "Reader_ID" = :Readernumber;
```

Now when you run the query, an entry field appears. It prompts you to enter a reader number. Whatever value you enter here, the media currently on loan to that reader will be displayed.

When using forms, the parameter can be passed from the main form to a subform. However it sometimes happens that queries using parameters in subforms are not updated, if data is changed or newly entered.

Often it would be nice to alter the contents of list boxes using settings in the main form. So for example, we could prevent library media from being loaned to individuals who are currently banned from borrowing media. Unfortunately controlling list box settings in this personalized way by using parameters is not possible.

Subqueries

Subqueries built into fields can always only return one record. The field can also return only one value.

```
SELECT "ID", "Income", "Expenditure", ( SELECT SUM( "Income" ) -
SUM( "Expenditure" ) FROM "Checkout") AS "Balance" FROM "Checkout";
```

This query allows data entry (primary key included). The subquery yields precisely one value, namely the total balance. This allows the balance at the till to be read after each entry. This is still not comparable with the supermarket checkout form described in “Queries as a basis for additional information in forms“. Naturally it lacks the individual calculations of Total * Unit_price, but also the presence of the receipt number. Only the total sum is given. At least the receipt number can be included by using a query parameter:

```
SELECT "ID", "Income", "Expenditure", ( SELECT SUM( "Income" ) -
SUM( "Expenditure" ) FROM "Checkout" WHERE "Receipt_ID" =
:Receipt_Number) AS "Balance" FROM "Checkout" WHERE "Receipt_ID" =
:Receipt_Number;
```

In a query with parameters, the parameter must be the same in both query statements if it is to be recognized as a parameter.

For subforms such parameters can be included. The subform then receives, instead of a field name, the corresponding parameter name. This link can only be entered in the properties of the subform, and not when using the Wizard.

Note	Subforms based on queries are not automatically updated on the basis of their parameters. It is more appropriate to pass on the parameter directly from the main form.

Correlated subqueries

Using a still more refined query, an editable query allows one to even carry the running balance for the till:

```
SELECT "ID", "Income", "Expenditure", ( SELECT SUM( "Income" ) -
SUM( "Expenditure" ) FROM "Checkout" WHERE "ID" <= "a"."ID" ) AS
"Balance" FROM "Checkout" AS "a" ORDER BY "ID" ASC
```

The Checkout table is the same as Table "a". "a" however yields only the relationship to the current values in this record. In this way the current value of ID from the outer query can be evaluated within the subquery. Thus, depending on the ID, the previous balance at the corresponding time is determined, if you start from the fact that the ID, which is an autovalue, increments by itself.

Queries as source tables for queries

A query is required to set a lock against all readers who have received a third overdue notice for a medium.

```
SELECT "Loan"."Reader_ID", '3rd Overdue – the reader is blacklisted'
AS "Lock"
FROM (SELECT COUNT( "Date" ) AS "Total_Count", "Loan_ID"  FROM
"Recalls" GROUP BY "Loan_ID") AS "a", "Loan"
WHERE "a"."Loan_ID" = "Loan"."ID" AND "a"."Total_Count" > 2
```

First let us examine the **inner query**, to which the outer query relates. In this query the number of date entries, grouped by the foreign key Loan_ID is determined. This must not be made dependent on the Reader_ID, as that would cause not only three overdue notices for a single medium but also three media with one overdue notice each to be counted. The inner query is given an alias so that it can be linked to the Reader_ID from the outer query.

The **outer query** relates in this case only to the conditional formula from the inner query. It shows only a Reader_ID and the text for the Lock field when the "Loan"."ID" *and "*a"."Loan_ID" are *equal* and "a"."Total_Count" > *2* .

In principle all fields in the inner query are available to the outer one. So for example the sum "a"."Total_Count" can be merged into the outer query to give the actual fines total.

However it can happen, in the Query Design dialog, that the Design View Mode no longer works after such a construction. If you try to open the query for editing again you get the following warning:

If you then open the query for editing in SQL view and try to switch from there into the Design View, you get the error message:

The Design View Mode cannot find the field contained in the inner query "Loan_ID", which governs the relationship between the inner and outer queries.

When the query is run in SQL Mode, the corresponding content from the subquery is reproduced without error. Therefore you do *not* have to use direct SQL mode in this case.

The **outer query** used the results of the **inner query** to produce the final results. These are a list of the "Loan_ID" values that should be locked and why. If you want to further limit the final results, use the sort and filter functions of the graphical user interface.

Summarizing data with queries

When data is searched for over a whole database, the use of simple form functions often leads to problems. A form refers after all to only one table, and the search function moves only through the underlying records for this form.

Getting at all the data is simpler when you use queries, which can provide a picture of all the records. The section on “Relationship definition in the query” suggests such a query construction. This is constructed for the example database as follows:

```
SELECT "Media"."Title", "Subtitle"."Subtitle", "Author"."Author"
FROM "Media"
LEFT JOIN "Subtitle" ON "Media"."ID" = "Subtitle"."Media_ID"
LEFT JOIN "rel_Media_Author" ON "Media"."ID" =
"rel_Media_Author"."Media_ID"
LEFT JOIN "Author" ON "rel_Media_Author"."Author_ID" = "Author"."ID"
```

Here all "Titles", "Subtitles" and "Authors" are shown together.

The Media table contains a total of 9 Titles. For two of these titles, there are a total of 8 Subtitles. Without a **`LEFT JOIN`**, both tables displayed together yield only 8 records. For each Subtitle, the corresponding Title is searched for, and that is the end of the query. Titles *without* Subtitle are not shown.

Now to show all Media including those *without* a Subtitle: Media is on the left side of the assignment, Subtitle on the right side. A **`LEFT JOIN`** will show every Title from Media, but only a Subtitle for those that have a Title. Media becomes the decisive table for determining which records are to be displayed. This was already planned when the table was constructed (see

Chapter 3, Tables). As Subtitles exist for two of the nine Titles, the query now displays 9 + 8 – 2 = 15 records.

Note	The normal linking of tables, after all tables have been counted, follows the keyword WHERE. If there is a **`LEFT JOIN`** or a **`RIGHT JOIN`**, the assignment is defined directly after the two table names using **`ON`**. The sequence is therefore always **`Table1 LEFT JOIN Table2 ON Table1.Field1 = Table2.Field1 LEFT JOIN Table3 ON Table2.Field1 = Table3.Field1 ...`**

Two titles of the Media table do not yet have an author entry or a Subtitle. At the same time one Title has a total of three Authors. If the Author Table is linked without a **`LEFT JOIN,`** the two Media *without* an Author will not be shown. But as one medium has three authors instead of one, the total number of records displayed will still be 15.

Only by using **`LEFT JOIN`** will the query be instructed to use the Media table to determine which records to show. Now the records without Subtitle or Author appear again, giving a total of 17 records.

Using appropriate Joins usually increases the amount of data displayed. But this enlarged data set can easily be scanned, since authors and subtitles are displayed in addition to the titles. In the example database, all of the media-dependent tables can be accessed.

More rapid access to queries using table views

Views in SQL are quicker than queries, especially for external databases, as they are anchored directly into the database and the server returns only the results. By contrast queries are first sent to the server and processed there.

If a new query relates to another query, the SQL view in Base makes the other query look like a table. If you create a View from it, you can see that you are actually working with a subquery (Select used within another Select). Because of this, a Query 2 that relates to another Query 1 cannot be run by using **Edit > Run SQL command directly**, since only the graphical user interface and not the database itself knows about Query 1.

The database gives you no direct access to queries. This also applies to access using macros. Views, on the other hand, can be accessed from both macros and tables. However, no records can be edited in a view. (They must be edited in a table or form.)

Tip	A query created using **Create Query in SQL View** has the disadvantage that it cannot be sorted or filtered using the GUI. There are therefore limits to its use. A *View* on the other hand can be managed in Base just like a normal table – with the exception that no change in the data is possible. Here therefore even in direct SQL-commands all possibilities for sorting and filtering are available.

Views are a solution for many queries, if you want to get any results at all. If for example a Subselect is to be used on the results of a query, create a **View** that gives you these results. Then use the subselect on the View. Corresponding examples are to be found in Chapter 8, Database Tasks.

Creating a View from a query is rather easy and straight forward.

1) Click the **Table** object in the Database section.
2) Click **Create View**.
3) Close the Add Table dialog.

4) Click the Design View On/OFF icon. (This is the SQL Mode for a View.)
5) Getting the SQL for the View:
   a) Edit the query in SQL View.
   b) Use *Control+A* to highlight the query's SQL.
   c) Use *Control+C* to copy the SQL.
6) In the SQL Mode of the View use *Control+V* to paste the SQL.
7) Close, Save, and Name the View.