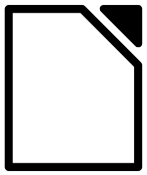

LibreOffice
The Document Foundation

Base Handbook

Chapter 8
Database tasks

Copyright

Contributors

Jochen Schiffers
Robert Großkopf
Jost Lange
Hazel Russman
Dan Lewis

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 3 June 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 remarks on database tasks

This chapter describes some solutions for problems that arise for many datsbase users.

Data filtering

Data filtering using the GUI is described in the chapter on data entry into tables. Here we describe a solution to a problem which many users have raised: how to use listboxes to search for the content of fields in tables, which then appear filtered in the underlying form section and can be edited.

The basis for this filtering is an editable query (see the chapter on queries) and an additional table, in which the data to be filtered are stored. The query shows from its underlying table only the records that correspond to the filter values. If no filter value is given, the query shows all records.

The following example starts from a **MediaExample** table that includes, among others, the following fields: **ID** (primary key), **Title**, **Category**. The field types are **INTEGER**, **VARCHAR**, and **VARCHAR** respectively.

First we require a **FilterExample** table. This table contains a primary key and 2 filter fields (of course you can have more if you want): **"ID"** (primary key), **Filter_1**, **Filter_2**. As the fields of the **MediaExample** table, which is to be filtered, are of the type **VARCHAR**, the fields **Filter_1** and **Filter_2** are also of this type. **ID** can be the smallest numeric type, **TINYINT** because the **Filter** table will never contain more than one record.

You can also filter fields that occur in the **MediaExample** table only as foreign keys. In that case, you must give the corresponding fields in the **FilterExample** table the type appropriate for the foreign keys, usually **INTEGER**.

The following query is certainly editable:

```
SELECT * FROM "MediaExample"
```

All records from the **MediaExample** table are displayed, including the primary key.

```
SELECT * FROM "MediaExample" WHERE "Title" = IFNULL( ( SELECT
"Filter_1" FROM "FilterExample" ), "Title" )
```

If the field **Filter_1** is not **NULL**, those records are displayed for which the **Title** is the same as **Filter_1**. If the field **Filter_1** is **NULL**, the value of the **Title** field is used instead. As **Title** is the same as **"Title"**, all records are displayed. This assumption does not hold however if the **Title** field of any record is empty (contains **NULL)**. That means that those records will never be displayed that have no title entry. Therefore we need to improve the query:

```
SELECT * , IFNULL( "Title", '' ) AS "T" FROM "MediaExample" WHERE "T"
= IFNULL( ( SELECT "Filter_1" FROM "FilterExample" ), "T" )
```

Tip

IFNULL(expression, value) requires the **expression** has the same field type as the **value**.

- If the **expression** has the field type **VARCHAR**, use two single quotes **''** as the value.
- If it has **DATE** as its field type, enter a date as the value that is not contained in the field of the table to be filtered. Use this format: **{D 'YYYY-MM-DD'}**.
- If it is any of the numerical field types, use the **NUMERIC** field type for the value. Enter a number that does not appear in the field of the table to be filtered.

This variant will lead to the desired goal. Instead of filtering **Title** directly, a field is filtered which carries the alias **T**. This field has no content either but it is not **NULL**. In the conditions only the field **T** is considered. All records are therefore displayed even if **Title** is **NULL**.

Unfortunately you cannot do this using the GUI. This command is available only directly with SQL. To make it editable in the GUI, further modification is required:

```
SELECT "MediaExample".* , IFNULL( "MediaExample"."Title", '' ) AS "T"
FROM "MediaExample" WHERE "T" = IFNULL( ( SELECT "Filter_1" FROM
"FilterExample" ), "T" )
```

If the relationship of the table to the fields is now set up, the query becomes editable in the GUI.

As a test, you can put a title into **"Filter"."Filter_1"**. As **"Filter"."ID"** set the value **'0'**. The record is saved and the filtering can be comprehended. If **"Filter"."Filter_1"** is emptied, the GUI treats that as **NULL**. A new test yields a display of all the media. In any case, before a form is created and tested, just one record with a primary key should be entered into the **Filter** table. It must be only one record, since sub-queries as shown above can only transmit one value.

The query can now be enlarged to filter two fields:

```
SELECT "MediaExample".* , IFNULL( "MediaExample"."Title", '' ) AS "T",
IFNULL( "MediaExample"."Category", '' ) AS "K" FROM "MediaExample"
WHERE "T" = IFNULL( ( SELECT "Filter_1" FROM "FilterExample" ), "T" )
AND "K" = IFNULL( ( SELECT "Filter_2" FROM "FilterExample" ), "K" )
```

This concludes the creation of the editable query. Now for the basic query for the two listboxes:

```
SELECT DISTINCT "Title", "Title" FROM "MediaExample" ORDER BY "Title"
ASC
```

The listbox should show the **Title** and then also transmit that **Title** to the **Filter_1** field in the **Filter** table that underlies the form. Also no duplicate values should be shown (**'DISTINCT'** condition) . And the whole thing should of course be sorted into the correct order.

A corresponding query is then created for the **Category** field, which is to write its data into the **Filter_2** field in the **Filter** table.

If one of these fields contains a foreign key, the query is adapted so that the foreign key is passed to the underlying **Filter** table.

The form consists of two parts. Form 1 is the form based on the **Filter** table. Form 2 is the form based on the query. Form 1 has **no navigation bar** and the cycle is set to **Current record**. In addition, the **Allow additions** property is set to **No**. The first and only record for this form already exists.

Form 1 contains two listboxes with appropriate labels. Listbox 1 returns values for **Filter_1** and is linked to the query for the **Title** field. Listbox 2 returns values for **Filter_2** and relates to the query for the **Category** field.

Form 2 contains a table control field, in which all fields from the query can be listed except for the fields **T** und **K**. The form would still work if these fields were present; they are omitted to avoid a confusing duplication of field contents. In addition form 2 contains a button, linked to the **Update form** function. An additional navigation bar can be built in to prevent screen flicker every time the form changes, due to the navigation bar being present in one form and not in the other.

Once the form is finished, the test phase begins. When a listbox is changed, the button on form 2 is used to store this value and update Form 2. This now relates to the value which the listbox provides. The filtering can be made retrospective by choosing an empty field in the listbox.

Searching for data

The main difference between searching for data and filtering data is in the query technique. The aim is to deliver, in response to free language search terms, a resulting list of records that may only partially contain these actual terms. First the similar approaches to the table and form are described.

The table for the search content may be the same one that already contains the filter values. The **Filter** table is simply expanded to include a field named **Searchterm**. So, if required, the same table can be accessed and, using the forms, simultaneously filtered and searched. **Searchterm** has the field type **VARCHAR**.

The form is built just as for filtering. Instead of a listbox, we need a text entry field for the search term, and also perhaps a label field with the title Search. The field for the search term can stand alone in the form or together with the fields for filtering, if both functions are desired.

The difference between filtering and searching lies in the query technique. While filtering uses a term that already occurs in the underlying table, searching uses arbitrary entries. (After all, the listbox is constructed from the table content.)

```
SELECT * FROM "MediaExample" WHERE "Title" = ( SELECT "Searchterm"
FROM "FilterExample" )
```

This query normally leads to an empty result list for these reasons:

1) When entering search terms , people seldom know completely and accurately what the title is. Therefore the correct title does not get displayed. To find the book "Per Anhalter through the Galaxy" it should be sufficient to put "Anhalter" into the Search field or even just "Anh".
2) If the field “Searchterm” is empty, only records are displayed in which there is no title. The field “Searchterm” is empty, the **Title** field must be empty also. This only happens in one of two possibilities: the item does not have a title, or someone did not enter its title.

The last condition can be removed if the filtering condition is:

```
SELECT * FROM "MediaExample" WHERE "Title" = IFNULL( ( SELECT
"Searchterm" FROM "FilterExample" ), "Title" )
```

With this refinement of the filtering (what happens if the title is **NULL**?) we get a result more in line with expectations. But the first condition is still not fulfilled. Searching should work well when only fragmentary knowledge is available. The query technique must therefore use the **LIKE** condition:

```
SELECT * FROM "MediaExample" WHERE "Title" LIKE ( SELECT '%' ||
"Searchterm" ||'%' FROM "FilterExample" )
```

or better still:

```
SELECT * FROM "MediaExample" WHERE "Title" LIKE IFNULL( ( SELECT '%'
|| "Searchterm" ||'%' FROM "FilterExample" ), "Title" )
```

LIKE, coupled with **%**, means that all records are displayed which have the search term anywhere within them. **%** is a wildcard for any number of characters before or after the search term. Various projects still remain after this version of the query has been built:

- It is common to use lower case letters for search terms. So how do I get a result if I type "anhalter" instead of "Anhalter"?
- What other conventions in writing need to be considered?
- What about fields that are not formatted as text fields? Can you search for dates or numbers with the same search field?
- And what if, as in the case of the filter, you want to prevent **NULL** values in the field from causing all the records to being displayed?

The following variant covers one or two of these possibilities:

```
SELECT * FROM "MediaExample" WHERE
LOWER("Title") LIKE IFNULL( ( SELECT '%' || LOWER("Searchterm") ||'%'
FROM "FilterExample" ), LOWER("Title") )
```

The condition changes the search term and the field content to lower case. This also allows whole sentences to be compared.

```
SELECT * FROM "MediaExample" WHERE
LOWER("Title") LIKE IFNULL( ( SELECT '%' || LOWER("Searchterm") ||'%'
FROM "FilterExample" ), LOWER("Title") ) OR
LOWER("Category") LIKE ( SELECT '%' || LOWER("Searchterm") ||'%' FROM
"FilterExample" )
```

The **IFNULL** function must occur only once, so that when the **Searchterm is NULL**, **LOWER("Title") LIKE LOWER("Title")** is queried. And as the title should be a field that cannot be **NULL**, in such cases all records are displayed. Of course, for multiple field searches, this code becomes correspondingly longer. In such cases it is better to use a macro, to allow the code to cover all the fields in one go.

But does the code still work with fields that are not text? Although the LIKE condition is really tailored to text, it also works for numbers, dates, and times without needing any alterations. So in fact text conversion need not take place. However, a time field that is a mixture of text and numbers cannot interact with the search results – unless the query is broadened, so that a single search term is subdivided across all the spaces between the text and numbers. This, however, will significantly bloat the query.

Code snippets

These code snippets come from queries to mailing lists. Particular problems arise that might perhaps be useful as solutions for your own database experiments.

Getting someone's current age

A query needs to calculate a person's actual age from a birth date. See also the functions in the appendix to this Base Handbook.

```
SELECT DATEDIFF('yy',"Birthdate",CURDATE()) AS "Age" FROM "Person"
```

This query gives the age as a difference in years. But, the age of a child born on 31 December 31 2011 would be given as 1 year on 1 January 2012. So we also need to consider the position of the day within the year. This is accessible using the **'DAYOFYEAR()'** function. Another function will carry out the comparison.

```
SELECT CASEWHEN
( DAYOFYEAR("Birthdate") > DAYOFYEAR(CURDATE()) ,
DATEDIFF ('yy',"Birthdate",CURDATE())-1,
DATEDIFF ('yy',"Birthdate",CURDATE()))
AS "Age" FROM "Person"
```

Now we get the correct current age in years.

CASEWHEN can also be used to make the text **Birthday today** appear in another field, if **DAYOFYEAR("Birthdate") = DAYOFYEAR(CURDATE())**.

A subtle objection might now arise: "What about leap years?". For persons born after 28 February, there will be an error of one day. Not a serious problem in everyday use, but should we not strive for accuracy?

```
CASEWHEN (
(MONTH("Birthdate") > MONTH(CURDATE())) OR
((MONTH("Birthdate") = MONTH(CURDATE())) AND (DAY("Birthdate") >
DAY(CURDATE()))) ,
DATEDIFF('yy',"Birthdate",CURDATE())-1,
DATEDIFF('yy',"Birthdate",CURDATE()))
```

The code above achieves this goal. As long as the month of the birth date is greater than the current month, the year difference function will subtract one year. Equally one year will be subtracted when the two months are the same, but the day of the month for the birth date is greater than the day in the current date. Unfortunately this formula is not comprehensible to the GUI. Only **'Direct SQL-Command'** will handle this query successfully and that would prevent our query from being edited. But the query needs to be editable, so here is how to trick the GUI:

```
CASE
WHEN MONTH("Birthdate") > MONTH(CURDATE())
THEN DATEDIFF('yy',"Birthdate",CURDATE())-1
WHEN (MONTH("Birthdate") = MONTH(CURDATE()) AND DAY("Birthdate") >
DAY(CURDATE()))
THEN DATEDIFF('yy',"Birthdate",CURDATE())-1
ELSE DATEDIFF('yy',"Birthdate",CURDATE())
END
```

With this formulation, the GUI no longer reacts with an error message. The age is now given accurately even in leap years and the query still remains editable.

Getting a running balance by categories

Instead of using a household book, a database on a PC can simplify the tiresome business of adding up expenses for food, clothing, transport and so on. We want most of these details to be immediately visible in the database, so our example assumes that income and expenditure will be stored as signed values in one field called Amount. In principle, the whole thing can be expanded to cover separate fields and a relevant summation for each.

```
SELECT "ID", "Amount", ( SELECT SUM( "Amount" ) FROM "Cash" WHERE "ID"
<= "a"."ID" ) AS "Balance" FROM "Cash" AS "a" ORDER BY "ID" ASC
```

This query causes for each new record a direct calculation of the current account balance. At the same time the query remains editable because the "Balance" field is created through a correlating sub-query. The query depends on the automatically created primary key "ID" to calculate the state of the account. However balances are usually calculated on a daily basis. So we need a date query.

```
SELECT "ID", "Date", "Amount", ( SELECT SUM( "Amount" ) FROM "Cash"
WHERE "Date" <= "a"."Date" ) AS "Balance" FROM "Cash" AS "a" ORDER BY
"Date", "ID" ASC
```

The expenditure now appears sorted and summed by date. There still remains the question of the category, since we want corresponding balances for the individual categories of expenditure.

```
SELECT "ID", "Date", "Amount", "Acct_ID",
( SELECT "Acct" FROM "Acct" WHERE "ID" = "a"."Acct_ID" ) AS
"Acct_name",
( SELECT SUM( "Amount" ) FROM "Cash" WHERE "Date" <= "a"."Date" AND
"Acct_ID" = "a"."Acct_ID" ) AS "Balance",
( SELECT SUM( "Amount" ) FROM "Cash" WHERE "Date" <= "a"."Date" ) AS
"Total_balance"
FROM "Cash" AS "a" ORDER BY "Date", "ID" ASC
```

This creates an editable query in which, in addition to the entry fields (Date, Amount, Acct_ID), the account name, the relevant balance, and the total balance appear together. As the correlating subqueries are partially based on previous entries ("Date" <= "a"."Date") only new entries will go through smoothly. Alterations to a previous record are initially detectable only in that record. The query must be updated if later calculations dependent on it are to be carried out.

Line numbering

Automatically incrementing fields are fine. However, they do not tell you definitely how many records are present in the database or are actually available to be queried. Records are often deleted and many users try in vain to determine which numbers are no longer present in order to make the running number match up.

```
SELECT "ID", ( SELECT COUNT( "ID" ) FROM "Table" WHERE "ID" <=
"a"."ID" ) AS "Nr." FROM "Table" AS "a"
```

The ID field is read, and the second field is determined by a correlating sub-query, which seeks to determine how many field values in ID are smaller than or equal to the current field value. From this a running line number is created.

Each record to which you want to apply this query contains fields. To apply this query to the records, you must first add these fields to the query. You can place them in whatever order you desire in the **SELECT** clause. If you have the records in a form, you need to modify the form so that the data for the form comes from this query.

For example the record contains field1, field2, and field3. The complete query would be:

```
SELECT "ID", "field1", "field2", "field3", ( SELECT COUNT( "ID" ) FROM
"Table" WHERE "ID" <= "a"."ID" ) AS "Nr." FROM "Table" AS "a"
```

A numbering for a corresponding grouping is also possible:

```
SELECT "ID", "Calculation", ( SELECT COUNT( "ID" ) FROM "Table" WHERE
"ID" <= "a"."ID" AND "Calculation" = "a"."Calculation" ) AS "Nr." FROM
"Table" AS "a" ORDER BY "ID" ASC, "Nr." ASC
```

Here one table contains different calculated numbers. ("Calculation"). For each calculated number, **"Nr."** is separately expressed in ascending order after sorting on the ID field. This produces a numbering from 1 upwards.

If the actual sort order within the query is to agree with the line numbers, an appropriate type of sorting must be mapped out. For this purpose the sort field must have a unique value in all records. Otherwise two place numbers will have the same value. This can actually be useful if, for example, the place order in a competition is to be depicted, since identical results will then lead to a joint position. In order for the place order to be expressed in such a way that, in case of joint positions, the next value is omitted, the query needs to be be constructed somewhat differently:

```
SELECT "ID", ( SELECT COUNT( "ID" ) + 1 FROM "Table" WHERE "Time" <
"a"."Time" ) AS "Place" FROM "Table" AS "a"
```

All entries are evaluated for which the "Time" field has a smaller value. That covers all athletes who reached the winning post before the current athlete. To this value is added the number 1. This determines the place of the current athlete. If the time is identical with that of another athlete, they are placed jointly. This makes possible place orders such as 1st Place, 2nd Place, 2nd Place, 4. Place.

It would be more problematic, if line numbers were required as well as a place order. That might be useful if several records needed to be combined in one line.

```
SELECT "ID", ( SELECT COUNT( "ID" ) + 1 FROM "Table" WHERE "Time" <
"a"."Time" ) AS "Place",
CASE WHEN
( SELECT COUNT( "ID" ) + 1 FROM "Table" WHERE "Time" = "a"."Time" ) = 1
```

```
THEN ( SELECT COUNT( "ID" ) + 1 FROM "Table" WHERE "Time" < "a"."Time" )
ELSE (SELECT ( SELECT COUNT( "ID" ) + 1 FROM "Table" WHERE "Time" <
"a"."Time" ) + COUNT( "ID" ) FROM "Table" WHERE "Time" = "a"."Time" "ID"
< "a"."ID"
END
AS "LineNumber" FROM "Table" AS "a"
```

The second column still gives the place order. The third column checks first if only one person crossed the line with this time. If so, the place order is converted directly into a line number. Otherwise a further value is added to the place order. For the same time (**"Time" = "a"."Time"**) at least 1 is added, if there is a further person with the primary key ID, whose primary key is smaller than the primary key in the current record (**"ID" < "a"."ID"**). This query therefore yields identical values for the place order so long as no second person with the same time exists. If a second person with the same time does exist, the ID determines which person has the lesser line number.

Incidentally, this sorting by line number can serve whatever purpose the users of the database want. For example, if a series of records are sorted by name, records with the same name are not sorted randomly but according to their primary key, which is of course unique. In this way too, numbering can lead to a sorting of records.

Line numbering is also a good prelude to the combining of individual records into a single record. If a line-numbering query is created as a view, a further query can be applied to it without creating any problem. As a simple example here once more is the first numbering query with one extra field:

```
SELECT "ID", "Name", ( SELECT COUNT( "ID" ) FROM "Table" WHERE "ID" <=
"a"."ID" ) AS "Nr." FROM "Table" AS "a"
```

This query is turned into the view '*View1'*. The query can be used, for example, to put the first three names together in one line:

```
SELECT "Name" AS "Name_1", ( SELECT "Name" FROM "View1" WHERE "Nr." =
2 ) AS "Name_2", ( SELECT "Name" FROM "View1" WHERE "Nr." = 3 ) AS
"Name_3" FROM "View1"  WHERE "Nr." = 1
```

In this way several records can be converted into adjacent fields. This numbering simply runs from the first to the last record.

If all these individuals are to be assigned the same surname, this can be carried out as follows:

```
SELECT "ID", "Name", "Surname", ( SELECT COUNT( "ID" ) FROM "Table"
WHERE "ID" <= "a"."ID" AND "Surname" = "a"."Surname") AS "Nr." FROM
"Table" AS "a"
```

Now that the view has been created, the family can be assembled.

```
SELECT "Surname", "Name" AS "Name_1", ( SELECT "Name" FROM "View1"
WHERE "Nr." = 2  AND "Surname" = "a"."Surname") AS "Name_2", ( SELECT
"Name" FROM "View1" WHERE "Nr." = 3  AND "Surname" = "a"."Surname") AS
"Name_3" FROM "View1" AS "a" WHERE "Nr." = 1
```

In this way, in an address book, all members of one family ("Surname") can be collected together so that each address need be considered only once when sending a letter, but everyone who should receive the letter is listed.

We need to be careful here, as we do not want an endlessly looping function. The query in the above example limits the parallel records that are to be converted into fields to 3. This limit was chosen deliberately. No further names will appear even if the value of *"Nr."* is greater than 3.

In a few cases such a limit is clearly understandable. For example, if we are creating a calendar, the lines might represent the weeks of the year and the columns the weekdays. As in the original calendar only the date determines the field content, line numbering is used to number the days of each week continuously and then the weeks in the year become the records. This does require that the table contains a date field with continuous dates and a field for the events. Also the earliest

date will always create an “Nr.” = 1. So, if you want the calendar to begin on Monday, the earliest date must be on Monday. Column 1 is then Monday, column 2 Tuesday and so on. The subquery then ends at *"Nr."* = *7*. In this way all seven days of the week can be shown alongside each other and a corresponding calendar view created.

Getting a line break through a query

Sometimes it is useful to assemble several fields using a query and separate them by line breaks, for example when reading a complete address into a report.

The line break within the query is represented by **`'Char(13)'`**. Example:

```
SELECT "Firstname"||' '||"Surname"||Char(13)||"Road"||Char(13)||"Town"
FROM "Table"
```

This yields:

```
Firstname Surname
Road
Town
```

Such a query, with a line numbering up to 3, allows you to print address labels in three columns by creating a report. The numbering is necessary in this connection so that three addresses can be placed next to one another in one record. That is the only way they will remain next to each other when read into the report.

Grouping and summarizing

For other databases, and for newer versions of HSQLDB, the **`Group_Concat()`** command is available. It can be used to group individual fields in a record into one field. So, for example, it is possible to store first names and surnames in one table, then to present the data in such a way that one field shows the surnames as family names while a second field contains all the relevant first names sequentially, separated by commas.

This example is similar in many ways to line numbering. The grouping into a common field is a kind of supplement to this.

Surname	***Firstname***
Müller	Karin
Schneider	Gerd
Müller	Egon
Schneider	Volker
Müller	Monika
Müller	Rita

is converted by the query to:

Surname	***Firstnames***
Müller	Karin, Egon, Monika, Rita
Schneider	Gerd, Volker

This procedure can, within limits, be expressed in HSQLDB. The following example refers to a table called Name with the fields ID, Firstname and Surname. The following query is first run on the table and saved as a view called View_Group.

```
SELECT "Surname", "Firstname", ( SELECT COUNT( "ID" ) FROM "Name"
WHERE "ID" <= "a"."ID" AND "Surname" = "a"."Surname" ) AS "GroupNr"
FROM "Name" AS "a"
```

You can read in the Queries chapter how this query accesses the field content in the same query line. It yields an ascending numbered sequence, grouped by *Surname*. This numbering is necessary for the following query, so that in the example a maximum of 5 first names is listed.

```
SELECT "Surname",
( SELECT "Firstname" FROM "View_Group" WHERE "Surname" = "a"."Surname"
AND "GroupNr" = 1 ) ||
IFNULL( ( SELECT ', ' || "Firstname" FROM "View_Group" WHERE "Surname"
= "a"."Surname" AND "GroupNr" = 2 ), '' ) ||
IFNULL( ( SELECT ', ' || "Firstname" FROM "View_Group" WHERE "Surname"
= "a"."Surname" AND "GroupNr" = 3 ), '' ) ||
IFNULL( ( SELECT ', ' || "Firstname" FROM "View_Group" WHERE "Surname"
= "a"."Surname" AND "GroupNr" = 4 ), '' ) ||
IFNULL( ( SELECT ', ' || "Firstname" FROM "View_Group" WHERE "Surname"
= "a"."Surname" AND "GroupNr" = 5 ), '' )
AS "Firstnames"
FROM "View_Group" AS "a"
```

Using sub-queries, the first names of the group members are searched for one after another and combined. From the second sub-query onward you must ensure that **'NULL'** values do not set the whole combination to **'NULL'**. That is why a result of **''** rather than **'NULL'** is shown.