Base Handbuch

Kapitel 8 Datenbankaufgaben

LibreOffice 7.6

Dokumentationen zu LibreOffice unter de.libreoffice.org

Copyright

Mitwirkende/Autoren

Robert Großkopf | Jost Lange | Jochen Schiffers

Michael Niedermair

Rückmeldung (Feedback)

Kommentare oder Vorschläge zu diesem Dokument können Sie in deutscher Sprache an die Adresse *discuss@de.libreoffice.org* senden.

Vorsicht

Alles, was an eine Mailingliste geschickt wird, inklusive der E-Mail-Adresse und anderer persönlicher Daten, die die E-Mail enthält, wird öffentlich archiviert und kann nicht gelöscht werden. Also, schreiben Sie mit Bedacht!

Datum der Veröffentlichung und Softwareversion

Veröffentlicht am 01.08.2023. Basierend auf der Version LibreOffice 7.6.

Inhalt

Allgemeines zu Datenbankaufgaben

Hier werden einige Lösungen für Problemstellungen vorgestellt, die im Laufe der Zeit viele Datenbankuser beschäftigen werden. Anfragen dazu kamen vor allem aus den Mailinglisten, insbesondere *users@de.libreoffice.org*,
sowie aus den Foren *http://de.openoffice.info/viewforum.php?f=8*
und *http://www.libreoffice-forum.de/viewforum.php?f=10* .

Datenfilterung

Die Datenfilterung mittels der GUI ist bereits bei der Dateneingabe in Tabellen beschrieben. Hier soll eine Lösung aufgezeigt werden, die bei vielen Nutzern gefragt ist: Mittels Listenfeldern werden Inhalte von Tabellenfeldern ausgesucht, die dann im darunterliegenden Formularteil herausgefiltert erscheinen und bearbeitet werden können.

Grundlage für diese Filterung ist neben einer bearbeitbaren Abfrage (siehe das Kapitel «Eingabemöglichkeit in Abfragen») eine weitere Tabelle, in der die zu filternden Daten abgespeichert werden. Die Abfrage zeigt aus der ihr zugrundeliegenden Tabelle nur die Datensätze an, die dem eingegebenen Filterwert entsprechen. Ist kein Filterwert angegeben, so zeigt die Abfrage alle Datensätze an.

Für das folgenden Beispiel wird von einer Tabelle **"Medien"** ausgegangen, die unter anderem die folgenden Felder beinhaltet: **"ID"** (Primärschlüssel), **"Titel"**, **"Kategorie"**.

Zuerst wird eine Tabelle **"Filter"** benötigt. Diese Tabelle erhält einen Primärschlüssel und 2 Filterfelder (das kann natürlich beliebig erweitert werden): **"ID"** (Primärschlüssel), **"Filter_1"**, **"Filter_2"**. Da die Felder der Tabelle **"Medien"**, die gefiltert werden sollen, vom Typ **VARCHAR** sind, haben auch die Felder **"Filter_1"** und **"Filter_2"** diesen Typ. **"ID"** kann ein **Ja/Nein**-Feld sein. Die Tabelle **"Filter"** wird sowieso nur einen Datensatz abspeichern.

Feldname	***Feldtyp***
ID	Ja/Nein [BOOLEAN]
Filter_1	Text [VARCHAR]
Filter_2	Text [VARCHAR]

Tipp

Wird keine Einbenutzer-Datenbank wie Base mit der internen HSQLDB genutzt, so würde so eine Filtertabelle erst einmal die Filterung auch bei anderen Nutzern erzeugen. Hier könnte einfach der Nutzername direkt als Primärschlüssel der Filtertabelle genutzt werden. Dann wäre das Feld ID kein Ja/Nein-Feld, sondern ein VARCHAR-Feld.
Über

```
001 SELECT CURRENT_USER From "Medien"
```

würde in diesem Falle der aktuelle Nutzer abgefragt. Bei der internen Datenbank ist das immer 'SA'. Entsprechend darf bei der Filterung dann allerdings nur der Datensatz ausgelesen werden, bei dem

```
001 "ID" = CURRENT_USER
```

ist. Unterscheiden sich die Nutzernamen nicht, so wäre mit **CURRENT_CONNECTION** der Integer-Wert der aktuellen Verbindung (FIREBIRD) nutzbar.

Natürlich kann auch nach Feldern gefiltert werden, die in der Tabelle "Medien" nur über einen Fremdschlüssel vertreten sind. Dann müssen die entsprechenden Felder in der Tabelle "Filter" natürlich dem Typ des Fremdschlüssels entsprechen, in der Regel also «Integer» sein.

Folgende Abfrageform bleibt sicher editierbar:

```
001 SELECT * FROM "Medien"
```

Alle Datensätze der Tabelle **"Medien"** werden angezeigt, auch der Primärschlüssel.

```
001 SELECT * FROM "Medien"
002 WHERE "Titel" = COALESCE(
003    ( SELECT "Filter_1" FROM "Filter" WHERE "ID" = TRUE), "Titel" )
```

Ist das Feld **"Filter_1"** nicht **NULL**, so werden die Datensätze angezeigt, bei denen der **"Titel"** gleich dem **"Filter_1"** ist. Wenn das Feld **"Filter_1" NULL** ist wird stattdessen der Wert des Feldes **"Titel"** genommen. Da **"Titel"** gleich **"Titel"** ist, werden so alle Datensätze angezeigt – sollte angenommen werden, trifft aber nicht zu, wenn im Feld **"Titel"** irgendwo ein leeres Feld **'NULL'** enthalten ist. Das bedeutet, dass die Datensätze nie angezeigt werden, die keinen Titeleintrag haben. Hier muss in der Abfrage nachgebessert werden.

```
001 SELECT * ,
002    COALESCE( "Titel", '' ) AS "T"
003 FROM "Medien"
004 WHERE "T" = COALESCE(
005    ( SELECT "Filter_1" FROM "Filter"  WHERE "ID" = TRUE), "T" )
```

Diese Variante würde zum Ziel führen. Statt **"Titel"** direkt zu filtern, wird ein Feld gefiltert, das den Alias-Namen **"T"** erhält. Dieses Feld ist zwar weiter ohne Inhalt, aber eben nicht **NULL**. In der Bedingung wird nur auf dieses Feld **"T"** Bezug genommen. Alle Datensätze werden angezeigt, auch wenn **"Titel" NULL** sein sollte.

Leider spielt hier die GUI nicht mit. Der Befehl ist nur direkt über SQL absetzbar. Um ihn mit der GUI editierbar zu machen, ist weitere Handarbeit erforderlich:

```
001 SELECT "Medien".* ,
002    COALESCE( "Medien"."Titel", '' ) AS "T"
003 FROM "Medien"
004 WHERE "T" = COALESCE(
005    ( SELECT "Filter_1" FROM "Filter"  WHERE "ID" = TRUE), "T" )
```

Wenn jetzt der Tabellenbezug zu den Feldern hergestellt ist, ist die Abfrage auch in der GUI editierbar.

Zum Testen kann jetzt einfach ein Titel in **"Filter"."Filter_1"** eingegeben werden. Als **"Filter"."ID"** wird der Wert **'0'** gesetzt. Der Datensatz wird abgespeichert und die Filterung kann nachvollzogen werden. Wird **"Filter"."Filter_1"** wieder geleert, so macht die GUI daraus **NULL**. Ein erneuter Test ergibt, dass jetzt wieder alle Medien angezeigt werden. Bevor ein Formular erstellt und getestet wird, sollte auf jeden Fall ein Datensatz, aber wirklich nur einer, mit einem Primärschlüssel in der Tabelle **"Filter"** stehen. Nur ein Datensatz darf es sein, da Unterabfragen wie oben gezeigt nur einen Wert wiedergeben dürfen.

Die Abfrage wird jetzt erweitert, um auch ein 2. Feld zu filtern:

```
001 SELECT "Medien".* ,
002    COALESCE( "Medien"."Titel", '' ) AS "T",
003    COALESCE( "Medien"."Kategorie", '' ) AS "K"
004 FROM "Medien"
005 WHERE "T" = COALESCE(
006    ( SELECT "Filter_1" FROM "Filter" WHERE "ID" = TRUE), "T" )
007    AND "K" = COALESCE(
008    ( SELECT "Filter_2" FROM "Filter" WHERE "ID" = TRUE), "K" )
```

Damit ist die Erstellung der editierbaren Abfrage abgeschlossen. Jetzt wird noch die Grundlage für die beiden Listenfelder als Abfrage zusammengestellt:

```
001 SELECT DISTINCT "Titel", "Titel"
002 FROM "Medien" ORDER BY "Titel" ASC
```

Das Listenfeld soll sowohl die **"Titel"** anzeigen als auch die **"Titel"** an die dem Formular zugrundeliegende Tabelle **"Filter"** in das Feld **"Filter_1"** weitergeben. Dabei sollen keine doppelten Werte angezeigt werden (Anordnung «**DISTINCT**») . Und das Ganze soll natürlich richtig sortiert erscheinen. Dabei ist die Abfrage an die Standardeinstellung der Listenfelder

angepasst, die dem gebundenen Feld eine '1' zuweist. Wird stattdessen unter **Eigenschaften Listenfeld → Daten → Gebundenes Feld** eine '0' zugewiesen, so braucht nur einmal das Feld "Titel" abgefragt werden.

Eine entsprechende Abfrage wird dann auch für das Feld **"Kategorie"** erstellt, die ihre Daten in der Tabelle **"Filter"** in das Feld **"Filter_2"** schreiben soll.

Handelt es sich bei einem der Felder um ein Fremdschlüsselfeld, so ist die Abfrage entsprechend so anzupassen, dass der Fremdschlüssel an die zugrundeliegende Tabelle "Filter" weitergegeben wird.

Das Formular besteht aus zwei Teilformularen. Formular 1 ist das Formular, dem die Tabelle **"Filter"** zugrunde liegt. Formular 2 ist das Formular, dem die Abfrage zugrunde liegt. Formular 1 hat **keine Navigationsleiste** und den Zyklus «**Aktueller Datensatz**». Die Eigenschaft «**Daten hinzufügen**» ist außerdem auf «**Nein**» gestellt. Der erste und einzige Datensatz existiert ja bereits.

Formular 1 enthält 2 Listenfelder mit entsprechenden Überschriften. Listenfeld 1 soll Werte für **"Filter_1"** liefern und wird mit der Abfrage für das Feld **"Titel"** versorgt. Listenfeld 2 soll Werte für **"Filter_2"** weitergeben und beruht auf der Abfrage für das Feld **"Kategorie"**.

Formular 2 enthält ein Tabellenkontrollfeld, in dem alle Felder aus der Abfrage aufgelistet sein können – mit Ausnahme der Felder **"T"** und **"K"**. Mit den Feldern wäre der Betrieb auch möglich – sie würden aber wegen der doppelten Feldinhalte nur verwirren. Außerdem enthält das Formular 2 noch einen Button, der die Eigenschaft «**Formular aktualisieren**» hat. Zusätzlich kann noch eine Navigationsleiste eingebaut werden, damit nicht bei jedem Formularwechsel der Bildschirm aufflackert, weil die Navigationsleiste in einem Formular ein-, in dem anderen ausgestellt ist.

Wenn das Formular fertiggestellt ist, geht es zur Testphase. Wird ein Listenfeld geändert, so reicht die Betätigung des Buttons aus dem Formular 2 aus, um zuerst diesen Wert zu speichern und dann das Formular 2 zu aktualisieren. Das Formular 2 bezieht sich jetzt auf den Wert, den das Listenfeld angibt. Die Filterung kann über die Wahl des im Listenfeld enthaltenen leeren Feldes rückgängig gemacht werden.

Datensuche

Der Hauptunterschied zwischen der Suche von Daten und der Filterung von Daten liegt in der Abfragetechnik. Schließlich soll zu frei eingegebenen Begriffen ein Ergebnis geliefert werden, das diese Begriffe auch nur teilweise beinhaltet.

Suche mit LIKE

Die Tabelle für die Suchinhalte kann die gleiche sein, in die bereits die Filterwerte eingetragen werden. Die Tabelle **"Filter"** wird einfach ergänzt um ein Feld mit der Bezeichnung **"Suchbegriff"**. So kann gegebenenfalls auf die gleiche Tabelle zugegriffen werden und in Formularen gleichzeitig gefiltert und gesucht werden. **"Suchbegriff"** hat die Feldeigenschaft **VARCHAR**.

Das Formular wird wie bei der Filterung aufgebaut. Statt eines Listenfeldes muss für den Suchbegriff ein Texteingabefeld erstellt werden, zusätzlich vielleicht auch ein Beschriftungsfeld mit dem Titel «Suche». Das Feld für den Suchbegriff kann alleine in dem Formular stehen oder zusammen mit den Feldern für die Filterung, wenn eben beide Funktionen gewünscht sind.

Der Unterschied zwischen Filterung und Suche liegt in der Abfragetechnik. Während die Filterung bereits von einem Begriff ausgeht, den es in der zugrundeliegenden Tabelle gibt (schließlich baut das Listenfeld auf den Tabelleninhalten auf) geht die Suche von einer beliebigen Eingabe aus.

```
001 SELECT * FROM "Medien"
002 WHERE "Titel" = ( SELECT "Suchbegriff" FROM "Filter" WHERE "ID" = TRUE)
```

Diese Abfrage würde in der Regel ins Leere führen. Das hat mehrere Gründe:

- Selten weiß jemand bei der Eingabe des Suchbegriffs den kompletten Titel fehlerfrei auswendig. Damit würde der Titel nicht angezeigt. Um das Buch «Per Anhalter durch die Galaxis» zu finden müsste es ausreichen, in das Suchfeld 'Anhalter' einzugeben, vielleicht auch nur 'Anh'.
- Ist das Feld "Suchbegriff" leer, so würde überhaupt kein Datensatz angezeigt. Die Abfrage gäbe **NULL** zurück und **NULL** kann in einer Bedingung nur mittels **IS NULL** erscheinen.
- Selbst wenn dies ignoriert würde, so würde die Abfrage dazu führen, dass alle die Datensätze angezeigt würden, die keine Eingabe im Feld **"Titel"** haben.

Die letzten beiden Bedingungen könnten erfüllt werden, indem wie bei der Filterung vorgegangen würde:

```
001 SELECT * FROM "Medien"
002 WHERE "Titel" = COALESCE(
003    ( SELECT "Suchbegriff" FROM "Filter" WHERE "ID" = TRUE), "Titel" )
```

Mit den entsprechenden Verfeinerungen aus der Filterung (was ist mit Titeln, die **NULL** sind?) würde das zum entsprechenden Ergebnis führen. Nur würde die erste Bedingung nicht erfüllt. Die Suche lebt ja schließlich davon, dass nur Bruchstücke geliefert werden. Die Abfragetechnik der Wahl müsste daher über den Begriff «**LIKE**» gehen:

```
001 SELECT * FROM "Medien"
002 WHERE "Titel" LIKE
003    ( SELECT '%' || "Suchbegriff" || '%' FROM "Filter" WHERE "ID" = TRUE)
```

oder besser:

```
001 SELECT * FROM "Medien"
002 WHERE "Titel" LIKE COALESCE(
003    ( SELECT '%' || "Suchbegriff" || '%' FROM "Filter" WHERE "ID" = TRUE),
004    "Titel" )
```

LIKE, gekoppelt mit '**%**', bedeutet ja, dass alle Datensätze gezeigt werden, die an irgendeiner Stelle den gesuchten Begriff stehen haben. '**%**' steht als Joker für beliebig viele Zeichen vor und hinter dem Suchbegriff. Verschiedene Baustellen bleiben nach dieser Abfrageversion:

- Besonders beliebt ist ja, in Suchformularen alles klein zu schreiben. Wie bekomme ich mit 'anhalter' statt 'Anhalter' auch noch ein Ergebnis?
- Welche anderen Schreibgewohnheiten gibt es noch, die vielleicht zu berücksichtigen wären?
- Wie sieht es mit Feldern aus, die nicht als Textfelder formatiert sind? Lassen sich auch Datumsanzeigen oder Zahlen mit dem gleichen Feld suchen?
- Und was ist, wenn, wie bei dem Filter, ausgeschlossen werden muss, dass **NULL**-Werte in dem Feld verhindern, dass alle Datensätze angezeigt werden?

Die folgende Variante deckt ein paar mehr Möglichkeiten ab:

```
001 SELECT * FROM "Medien"
002 WHERE LOWER("Titel") LIKE COALESCE(
003    ( SELECT '%' || LOWER("Suchbegriff") || '%'
004       FROM "Filter" WHERE "ID" = TRUE),
005    LOWER("Titel") )
```

Die Bedingung ändert den Suchbegriff und den Feldinhalt auf Kleinschreibweise. Damit werden auch ganze Sätze vergleichbar.

```
001 SELECT * FROM "Medien"
002 WHERE LOWER("Titel") LIKE COALESCE(
003    ( SELECT '%' || LOWER("Suchbegriff") || '%'
004       FROM "Filter" WHERE "ID" = TRUE), LOWER("Titel") )
```

```
005     OR LOWER("Kategorie") LIKE ( SELECT '%' || LOWER("Suchbegriff") || '%'
006        FROM "Filter" WHERE "ID" = TRUE)
```

Die **COALESCE**-Funktion muss nur einmal vorkommen, da bei dem **"Suchbegriff" NULL** ja dann **LOWER("Titel") LIKE LOWER("Titel")** abgefragt wird. Und da der Titel ein Feld sein soll, das nicht **NULL** sein darf, werden so auf jeden Fall alle Datensätze angezeigt. Für entsprechend viele Felder wird dieser Code natürlich entsprechend lang. Schöner geht so etwas mittels Makro, das dann den Code in einer Schleife über alle Felder erstellt.

Aber funktioniert der Code auch mit Feldern, die keine Textfelder sind? Obwohl die Bedingung **LIKE** ja eigentlich auf Texte zugeschnitten ist, brauchen Zahlen, Datums- oder Zeitangaben keine Umwandlung, um damit zusammen zu arbeiten. Allerdings können hierbei die Textumwandlungen unterbleiben. Nur wird natürlich ein Zeitfeld auf eine Mischung aus Text und Zahlen nicht mit einer Fundstelle reagieren können – es sei denn die Abfrage wird ausgeweitet, so dass der eine Suchbegriff an jeder Leerstelle unterteilt wird. Dies bläht allerdings die Abfrage noch wieder deutlich auf.

Tipp

Die Abfragen, die zur Filterung und zum Durchsuchen von Daten genutzt werden, lassen sich auch direkt in ein Formular einbauen.
Die gesamten obigen Bedingungen sind bei den Formular-Eigenschaften in der Zeile Filter eintragbar. Aus
SELECT * FROM "Medien" WHERE "Titel" = COALESCE((SELECT "Suchbegriff" FROM "Filter" WHERE "ID" = TRUE), "Titel")
wird dann ein Formular, das als Inhalt die Tabelle "Medien" nutzt.
Unter «Filter» steht dann
("Medien"."Titel" = COALESCE((SELECT "Suchbegriff" FROM "Filter" WHERE "ID" = TRUE), "Medien"."Titel"))
In den Filtereingaben ist darauf zu achten, dass die Bedingung in Klammern gesetzt wird und jeweils mit der Angabe "Tabelle"."Feld" arbeitet.
Vorteil dieser Variante ist, dass der Filter bei geöffnetem Formular ein- und wieder ausgeschaltet werden kann.

Suche mit LOCATE oder POSITION

Die Suche mit **LIKE** ist in der Regel völlig ausreichend für Datenbanken mit Feldern, die Text in überschaubarem Maße enthalten. Was aber, wenn der Inhalt über Memo-Felder eingegeben wird, also ohne weiteres auch einmal mehrere Seiten Text enthalten kann? Dann geht die Suche erst einmal los, wo denn nun der Text zu finden ist.

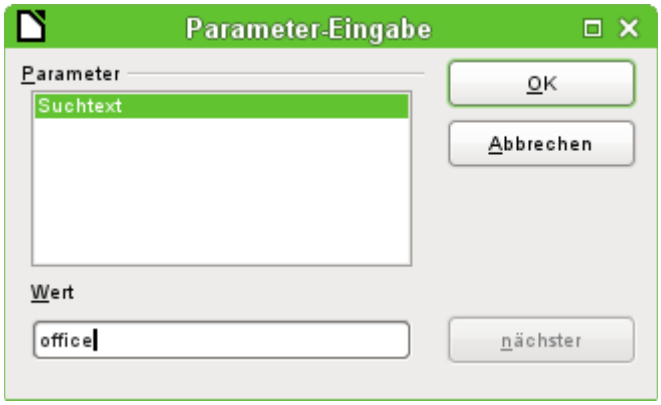
Um Text genau zu finden, gibt es in der HSQLDB die Funktion **LOCATE**. **LOCATE** erwartet einen Suchbegriff sowie den Text, der durchsucht werden soll, als Parameter. Zusätzlich *kann* noch angegeben werden, ab welcher Position gesucht werden soll. Kurz also: **LOCATE(Suchbegriff, Textfeld aus der Datenbank, Startposition der Suche)**.

In FIREBIRD gibt es die Funktion **LOCATE nicht**. Hier muss auf **POSITION** zurückgegriffen werden: **POSITION(Suchbegriff, Textfeld aus der Datenbank, Startposition der Suche)**. Die Startposition kann hier eingegeben werden, muss es aber nicht.

Auch die im weiteren verwendete Funktion **SUBSTRING** muss für FIREBIRD in einer andern Syntax geschrieben werden. Statt **SUBSTRING(Text,Startporition[,Länge])** ist dort **SUBSTRING(Text FROM Startposition [FOR Länge])** zu verwenden.

Für die folgende Erklärung wird eine Tabelle genutzt, die den Namen "Tabelle" hat. Der Primärschlüssel heißt "ID" und muss lediglich einzigartig sein. Zusätzlich gibt es noch ein Feld

"Memo", das als Feld des Typs **Memo (LONGVARCHAR)** erstellt wurde. In dem Feld "Memo" sind ein paar Absätze dieses Handbuchs gespeichert.[1]



Die Beispielabfragen sind als Parameterabfragen angelegt. Der einzugebende Suchtext ist jeweils 'office'.

	ID	Memo
▶	3	Jeder, der sich in das Modul Base von LibreOffice einarbeiten und tiefer einsteigen will, findet hier
	4	Dieses Buch, wie auch die anderen LibreOffice-Handbücher, das eingebaute Hilfesystem und die
	5	LibreOffice besitzt ein umfangreiches Hilfesystem.

Datensatz 1 von 3

```
SELECT "ID", "Memo" FROM "Tabelle"
WHERE LOWER ( "Memo" ) LIKE '%' || LOWER ( :Suchtext ) || '%'
```

Zuerst ein Zugriff über **LIKE**. **LIKE** kann nur in der Bedingung stehen. Wird der Suchtext irgendwo gefunden, dann wird der entsprechende Datensatz angezeigt. Durch den Vergleich von der Kleinschreibung des Feldinhaltes über **LOWER("Memo")** mit der Kleinschreibung des Suchtextes über **LOWER(:Suchtext)** werden die Inhalte unabhängig von der Schreibweise gefunden. Je länger der Text in dem Memo-Feld ist, desto schwerer wird es, den Begriff dann tatsächlich zu sehen.

	ID	Memo	Position
▶	1	Dieses Dokument unterliegt dem Copyright © 2014. Die Beitragenden sind unten aufgeführt. Sie	0
	2	Alles, was an eine Mailingliste geschickt wird, inklusive der E-Mail-Adresse und	0
	3	Jeder, der sich in das Modul Base von LibreOffice einarbeiten und tiefer einsteigen will, findet hier	44
	4	Dieses Buch, wie auch die anderen LibreOffice-Handbücher, das eingebaute Hilfesystem und die	40
	5	LibreOffice besitzt ein umfangreiches Hilfesystem.	6

Datensatz 1 von 5

```
SELECT "ID", "Memo", LOCATE( LOWER ( :Suchtext ), LOWER ( "Memo" ) ) AS "Position"
FROM "Tabelle"
```

LOCATE gibt genauer wieder, an welcher Stelle sich der Suchbegriff befindet. In Datensatz 1 und 2 ist der Suchbegriff nicht vorhanden. **LOCATE** gibt als Position hier '0' aus. Leicht nachzählen lässt sich das Ergebnis am Datensatz 5: Mit dem 6. Buchstaben beginnt hier die Textfolge 'Office'.

Natürlich wäre es auch möglich, das entsprechende Ergebnis wie bei **LIKE** auch über **LOCATE** zu erhalten:

```
001 SELECT "ID", "Memo"
002 FROM "Tabelle"
003 WHERE LOCATE(LOWER(:Suchtext),LOWER("Memo")) > 0
```

1 Die Screenshots zu diesem Kapitel entstammen der Datenbank «Beispiel_Autotext_Suchmarkierung_Rechtschreibung.odb», die dem Handbuch beiliegt.

Das Auffinden der Position allein ist im obigen Beispiel auch mit einem Blick auf das Feld "Memo" schon recht einfach. Komplizierter wird es aber, wenn der Inhalt eben nicht gerade, wie hier zur Demonstration, in den ersten 70 Zeichen enthalten ist. Dann wird es sinnvoll, Textstücke mit dem Inhalt direkt zu finden.

ID	Memo	Position	Treffer
1	Dieses Dokument	0	**keine Fundstelle**
2	Alles, was an eine	0	**keine Fundstelle**
3	Jeder, der sich in das	44	von LibreOffice einarbeiten und tiefer einsteigen will, findet hier die
4	Dieses Buch, wie auch	40	LibreOffice-Handbücher, das eingebaute Hilfesystem und die Benutzer-
5	LibreOffice besitzt ein	6	LibreOffice besitzt ein umfangreiches Hilfesystem.

Datensatz 1 von 5

```
SELECT "ID", "Memo", "Position",
    CASE
        WHEN "Position" = 0 THEN '**keine Fundstelle**'
        WHEN "Position" < 10 THEN SUBSTRING ( "Memo", 1 )
        ELSE SUBSTRING ( "Memo", LOCATE( ' ', "Memo", "Position" - 10 ) + 1 )
    END AS "Treffer"
FROM
    ( SELECT "ID", "Memo", LOCATE( LOWER ( :Suchtext ), LOWER ( "Memo" ) ) AS "Position"
    FROM "Tabelle" )
```

In der Spalte «Treffer» wird das Suchergebnis genauer dargestellt. Die vorherige Abfrage ist einfach als Basis für diese Abfrage genommen worden. Dies bewirkt, dass in der äußeren Abfrage nicht jedes Mal **LOCATE(LOWER(:Suchtext),LOWER("Memo"))** sondern einfach "Position" eingegeben werden kann. Vom Prinzip her ist dieses Verfahren nicht anders, als wenn die vorhergehende Abfrage gespeichert würde und diese Abfrage als Quellenangabe auf die vorherige Abfrage zugreift.

"Position" = 0 bedeutet, dass kein Suchergebnis vorhanden ist. In dem Fall also die Ausgabe **'**keine Fundstelle**'**.

"Position" < 10 bedeutet, dass sich der Suchbegriff direkt am Anfang des Textes befindet. 10 Zeichen können leicht überblickt werden. Es wird also der gesamte Text wiedergegeben. Hier könnte also auch statt **SUBSTRING("Memo",1)** direkt **"Memo"** stehen.

Für alle anderen Treffer wird ab 10 Zeichen vor der Position des Treffers nach einer Leerstelle **' '** gesucht. Der Text soll nicht mitten in einem Wort starten, sondern nach so einer Leerstelle beginnen. Über **SUBSTRING("Memo",LOCATE(' ',"Memo","Position"-10)+1)** wird erreicht, dass der Text mit dem Beginn eines Wortes startet, das maximal 10 Zeichen vor dem Begriff 'office' erscheint.

In der Praxis dürften hier mehr Zeichen erforderlich sein, da doch sehr viele Worte die Anzahl von 10 Zeichen übersteigen und selbst der Suchbegriff ja in einem Wort liegen kann, das noch 10 Zeichen vor dem eigentlich Begriff hat. 'LibreOffice' wird bei Suchbegriff 'office' so noch dargestellt, da das 'O' an der 6. Stelle steht. Stellen wir uns aber z.B. den Begriff 'hand' vor, so würde im 4. Datensatz bereits das Aus für die Darstellung stehen. 'LibreOffice-Handbücher' hat, von 'hand' aus nach links gezählt, 12 Zeichen. Wird aber höchsten 10 Zeichen nach links gesucht, so wird als erstes Leerzeichen das Zeichen hinter dem Komma gefunden. Die Darstellung in «Treffer» würde mit 'das eingebaute Hilfesystem ...' beginnen.

	ID	Memo	Position	Treffer
▶	1	Dieses Dokument unterliegt dem	0	**keine Fundstelle**
	2	Alles, was an eine Mailingliste	0	**keine Fundstelle**
	3	Jeder, der sich in das Modul Base von	44	von LibreOffice einarbeit
	4	Dieses Buch, wie auch die anderen	40	LibreOffice-Handbücher, d
	5	LibreOffice besitzt ein umfangreiches	6	LibreOffice besitzt ein u

Datensatz 1 von 5

```
SELECT "ID", "Memo", "Position",
    CASE
        WHEN "Position" = 0 THEN '**keine Fundstelle**'
        WHEN "Position" < 10 THEN SUBSTRING ( "Memo", 1 , 25 )
        ELSE SUBSTRING ( "Memo", LOCATE( ' ', "Memo", "Position" - 10 ) + 1 , 25 )
    END AS "Treffer"
FROM
    ( SELECT "ID", "Memo", LOCATE( LOWER ( :Suchtext ), LOWER ( "Memo" ) ) AS "Position"
    FROM "Tabelle" )
```

Die Abfragetechnik ist gegenüber der vorhergehenden Abfrage gleich geblieben. Lediglich die Länge des auszugebenden Treffers ist beschränkt worden. In diesem Falle erfolgte die Beschränkung hart auf 25 Zeichen. Die Funktion **SUBSTRING** erfordert als erstes die Angabe des zu durchsuchenden Textes, als zweites dann die Startposition der Ausgabe und als drittes optional die Länge des auszugebenden Textes. Natürlich hier auch reichlich kurz gehalten, aber eben nur zu Demonstrationszwecken. Vorteil der Verkürzung ist natürlich ein deutlich geringerer Speicherverbrauch bei entsprechend großen Datenmengen und ein direkter Blick auf die Fundstelle. Sichtbarer Nachteil dieser Form der Verkürzung ist aber, dass der Schnitt rigoros nach dem 25. Zeichen gemacht wird – ohne Rücksicht auf einen Wortbeginn.

	ID	Memo	Position	Treffer
▶	1	Dieses Dokument unterliegt dem	0	**keine Fundstelle**
	2	Alles, was an eine Mailingliste	0	**keine Fundstelle**
	3	Jeder, der sich in das Modul Base von	44	von LibreOffice einarbeiten und
	4	Dieses Buch, wie auch die anderen	40	LibreOffice-Handbücher, das
	5	LibreOffice besitzt ein umfangreiches	6	LibreOffice besitzt ein umfangreiches

Datensatz 1 von 5

```
SELECT "ID", "Memo", "Position",
    CASE
        WHEN "Position" = 0 THEN '**keine Fundstelle**'
        WHEN "Position" < 10 THEN SUBSTRING ( "Memo", 1, LOCATE( ' ', "Memo", 25 ) )
        ELSE SUBSTRING ( "Memo", LOCATE( ' ', "Memo", "Position" - 10 ) + 1,
            ( LOCATE( ' ', "Memo", "Position" + 20 ) -
                ( LOCATE( ' ', "Memo", "Position" - 10 ) + 1 ) )
            )
        END AS "Treffer"
FROM
    ( SELECT "ID", "Memo", LOCATE( LOWER ( :Suchtext ), LOWER ( "Memo" ) ) AS "Position"
    FROM "Tabelle" )
```

Hier wird ab dem 25. Zeichen in dem darzustellenden «Treffer» nach dem nächsten Leerzeichen gesucht. Der auszugebende Inhalt wird dann durch die gefundene Position begrenzt.

Recht einfach gestaltet sich dies noch, wenn der Treffer am Anfang liegt. Hier gibt **LOCATE(' ","Memo",25)** genau die Position vom Anfang des gesamten Textes an wieder. Sie entspricht, da der Text von Anfang an ausgegeben werden soll, auch genau der Länge des auszugebenden Begriffes.

Die Suche der dem Suchbegriff folgenden Leerzeichen ist auch bei einem weiter hinten liegenden Treffer nicht weiter kompliziert. Die Suche beginnt einfach an der Position des Treffers. Hinzugezählt werden noch 20 Zeichen, die auf jeden Fall folgen sollen. Danach wird das nächste Leerzeichen ausgemacht: **LOCATE(' ","Memo","Position"+20)**. Hiermit ist aber nur die Position im Gesamtfeld ausgemacht. Die ermittelte Position gibt also auf keinen Fall die Länge des auszugebenden Textes wieder. Von dem ermittelten Positionswert muss hingegen der Positionswert abgezogen werden, bei dem die Ausgabe des Treffers starten soll. Dies wurde durch

LOCATE(' ","Memo","Position"-10)+1 vorher bereits einmal abgefragt. Erst so kann dann die korrekte Länge des Textes dargestellt werden.

ID	Memo	Position01	Treffer01	Position02	Treffer02	Position03
1	Dieses	0	**keine Fundstelle**	0	**keine Fundstelle**	0
2	Alles, was	0	**keine Fundstelle**	0	**keine Fundstelle**	0
3	Jeder, der	44	von LibreOffice	312	einer OfficeSuite oder ein	0
4	Dieses	40	LibreOffice-Handbücher,	0	**keine Fundstelle**	0
5	LibreOffice	6	LibreOffice besitzt ein	123	Sie LibreOffice Hilfe aus dem	223

Datensatz 1 von 5

```
SELECT "ID", "Memo", "Position01", "Treffer01", "Position02",
    CASE
        WHEN "Position02" = 0 THEN '**keine Fundstelle**'
        WHEN "Position02" < 10 THEN SUBSTRING ( "Memo", 1, LOCATE( ' ', "Memo", 25 ) )
        ELSE SUBSTRING ( "Memo", LOCATE( ' ', "Memo", "Position02" - 10 ) + 1,
            ( LOCATE( ' ', "Memo", "Position02" + 20 ) -
                ( LOCATE( ' ', "Memo", "Position02" - 10 ) + 1 ) )
            )
    END AS "Treffer02",
    CASE
        WHEN "Position02" = 0 THEN 0
        ELSE LOCATE( LOWER ( :Suchtext ), LOWER ( "Memo" ), "Position02" + 1 )
    END AS "Position03"
FROM
    ( SELECT "ID", "Memo", "Position01",
        CASE
            WHEN "Position01" = 0 THEN '**keine Fundstelle**'
            WHEN "Position01" < 10 THEN SUBSTRING ( "Memo", 1, LOCATE( ' ', "Memo", 25 ) )
            ELSE SUBSTRING ( "Memo", LOCATE( ' ', "Memo", "Position01" - 10 ) + 1,
                ( LOCATE( ' ', "Memo", "Position01" + 20 ) -
                    ( LOCATE( ' ', "Memo", "Position01" - 10 ) + 1 ) )
                )
        END AS "Treffer01",
        CASE
            WHEN "Position01" = 0 THEN 0
            ELSE LOCATE( LOWER ( :Suchtext ), LOWER ( "Memo" ), "Position01" + 1 )
        END AS "Position02"
    FROM
        ( SELECT "ID", "Memo", LOCATE( LOWER ( :Suchtext ), LOWER ( "Memo" ) ) "Position01"
        FROM "Tabelle" )
    )
```

Mit der gleichen Technik können mehrere Abfragen hintereinander geschachtelt erfolgen. Die vorhergehende Abfrage ist jetzt die Datenquelle dieser Abfrage. Sie ist komplett unterhalb des Begriffes **FROM** in Klammern eingefügt worden. Lediglich die Felder wurden etwas umbenannt, da ja jetzt mehrere Positionen und Treffer angegeben werden. Außerdem wurde die nächste Position einer Fundstelle über **LOCATE(LOWER(:Suchtext),LOWER("Memo"),"Position01"+1)** ermittelt. Es wird also mit dem Suchen der nächsten Stelle einfach eine Stelle hinter dem vorhergehenden Treffer gestartet.

Die äußerste Abfrage stellt die entsprechenden Felder der anderen beiden Abfragen dar und fügt zusätzlich «Treffer02» auf die gleiche Weise hinzu, mit der vorher «Treffer01» ermittelt wurde. Außerdem wird in der äußeren Abfrage schon ermittelt, ob es vielleicht noch weitere Treffer gibt. Die entsprechende Position wird in «Position03» ausgegeben. Lediglich Datensatz 5 hat noch weitere Positionen mit Treffern vorzuweisen und könnte also in einer weiteren Nachfrage noch weitere Treffer ermöglichen.

Die Staffelung der Abfragen ist hier beliebig weit möglich. Allerdings werden sie mit jeder weiteren äußeren Abfrage natürlich für das System immer belastender. Hier sind entsprechende Tests notwendig, was denn nun sinnvoll und was realistisch machbar ist. Wie mit Hilfe von Makros über ein Formular sogar eine Abfragetechnik machbar ist, die gleichzeitig alle Fundstellen im Text markiert, ist im Kapitel «Makros» erklärt.

Bilder und Dokumente mit Base verarbeiten

Base-Formulare bieten für die Verarbeitung von Bildern grafische Kontrollfelder an. Nur über diese grafischen Kontrollfelder ist ohne Einsatz von Makros möglich, Bilder in die Datenbank einzulesen. Das grafische Kontrollfeld kann aber auch dazu genutzt werden, nur die Verknüpfung zu Bildern außerhalb der Datenbankdatei zu ermöglichen.[2]

Bilder in die Datenbank einlesen

Die Datenbank benötigt eine Tabelle, die mindestens die folgenden Voraussetzungen erfüllt:

Feldname	***Feldtyp***	***Beschreibung***
ID	Integer	Die ID ist Primärschlüssel dieser Tabelle.
Bild	Bild	Nimmt das Bild als Binärdatenstrom auf.

Hinweis

Bei Verwendung der internen Firebird Datenbank sollte nicht der Feldtyp **`Bild`** sondern der Feldtyp **`BLOB`** gewählt werden. Nur mit diesem Feldtyp werden Bilder auch im Formular angezeigt.

Beim Primärschlüssel ist natürlich nicht unbedingt der Integer-Feldtyp bestimmend. Ein Primärschlüssel ist aber unabdingbar. Andere Felder, die zumindest Informationen zu dem Bild enthalten, sollten noch hinzugefügt werden.

Daten, die irgendwann in das Bild-Feld eingetragen werden, sind in den Tabellen nicht lesbar. Dort erscheint als Anzeige nur **`<OBJECT>`**. Entsprechend sind Bilder auch nicht direkt in die Tabelle eingebbar. Sie müssen über ein Formular und dort mit Hilfe des grafischen Kontrollfeldes eingegeben werden. Das grafische Kontrollfeld öffnet beim Mausklick auf das Feld eine Dateiauswahl. Es zeigt hinterher das Bild an, das über die Dateiauswahl in die Datenbank eingelesen wurde.

Bilder, die direkt in die Datenbank eingefügt werden, sollten möglichst klein sein. Da Base ohne den Einsatz von Makros auch keine Möglichkeit bietet, die Bilder in Originalgröße wieder aus der Datenbank heraus zu befördern, macht es aus dieser Warte erst einmal Sinn, als Maßstab für die Größe z.B. einen möglichen Ausdruck im Bericht anzusehen. Originalbilder im Megapixelbereich sind hier völlig unnötig und blähen die Datenbank stark auf. Bereits nach wenigen Bildern meldet bei der internen HSQLDB Base eine **`Java.NullPointerException`** und kann den Datensatz nicht mehr speichern. Auch wenn die Bilder nicht ganz so groß sind, kann es irgendwann passieren, dass die Datenbankdatei nicht mehr bedienbar wird.

Bilder sollten außerdem möglichst nicht in Tabellen integriert werden, die als Suchgrundlage gedacht werden. Wird z.B. in einer Datenbank zur Personenverwaltung auch das Passbild mit abgespeichert, so ist es besser in einer separaten Tabelle über einen Fremdschlüssel mit der Haupttabelle verbunden. Die Suche in der Haupttabelle kann dann deutlich schneller erfolgen, da die Tabelle selbst nicht so viel Speicher beansprucht.

Bilder und Dokumente verknüpfen

Mit einer entsprechend durchdachten Ordnerstruktur ist es günstiger, direkt auf die Dateien von außerhalb zuzugreifen. Dateien außerhalb der Datenbank können beliebig groß sein, ohne dass die Funktionen der Datenbank selbst beeinflusst werden. Leider bedeutet dies aber auch, dass eine Umbenennung von Ordnern auf dem eigenen Rechner oder im Internet dazu führt, dass der Zugriff auf die entsprechenden Dateien verloren geht.

2 «Externe_Dateien.zip» ist als gepacktes Verzeichnis den zusätzlichen Beispieldatenbanken beigefügt.

Hinweis

Sollen die Bilder später über den Report-Designer ausgelesen werden, so dürfen die Dateinamen keine Sonderzeichen wie **[ ] { } \ < > % "** und Leerzeichen enthalten. Ein Bericht mit diesen Bildern wird nicht erstellt.

Um Bilder nicht in eine Datenbankdatei einzulesen, sondern nur zu verknüpfen, bedarf es nur einer kleinen Änderung gegenüber der vorhergehenden Tabelle:

Feldname	***Feldtyp***	***Beschreibung***
ID	Integer	Die ID ist Primärschlüssel dieser Tabelle.
Bild	Text	Nimmt den Pfad zu dem Bild auf.

Wird einfach statt des Feldtyps **Bild** der Feldtyp **Text** gewählt, so wird über das grafische Kontrollfeld der Pfad zu dem Bild eingetragen. Das Bild kann über das Kontrollfeld genauso betrachtet werden wie ein Bild, das in die Datenbank eingefügt wurde.

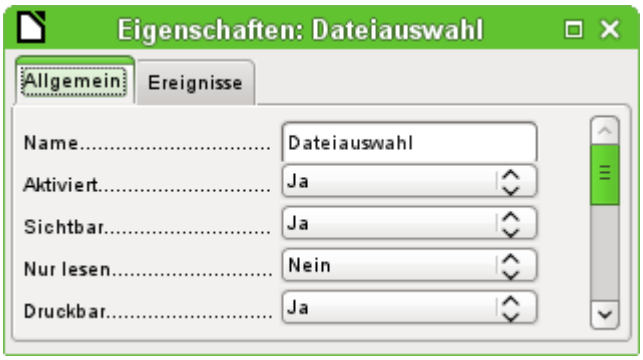
Hinweis

Seit LO 5.0 können, abhängig von der verwendeten Benutzeroberfläche, auch andere Dateien über das grafische Kontrollfeld eingebunden werden. Hier funktionierte das mit gtk3 allerdings erst ab der Version LO 6.3. **Für diese Versionen ist der Einsatz einer Dateiauswahl also weitgehend überflüssig.**
Bei *.pdf-Dateien wird die erste Seite in dem grafischen Kontrollfeld als Bild angezeigt.

Bei einem Bild kann über den Pfad wenigstens noch der Inhalt auf dem grafischen Kontrollfeld gelesen werden. Bei einem Dokument (mit Ausnahme des *.pdf-Dokumentes) kann aber keine Anzeige erfolgen, selbst wenn der Pfad in der Tabelle verzeichnet ist. Zuerst ist deshalb die Tabelle etwas zu erweitern, damit wenigstens ein geringfügiges Maß an Information zu dem Dokument sichtbar wird.

Feldname	***Feldtyp***	***Beschreibung***
ID	Integer	Die ID ist Primärschlüssel dieser Tabelle.
Beschreibung	Text	Beschreibung des Dokumentes, Suchbegriffe ...
Datei	Text	Nimmt den Pfad zu dem Bild auf.

Damit der Pfad zur Datei sichtbar wird, muss in dem Formular ein Dateiauswahlfeld mit eingebaut werden.



Ein Dateiauswahlfeld hat in seinen Eigenschaften keinen Reiter für Daten, ist also auch nicht mit irgendeinem Feld der dem Formular zugrundeliegenden Tabelle verbunden.

Dokumente mit absoluter Pfadangabe verknüpfen

Über das Dateiauswahlfeld kann zwar der Pfad angezeigt, aber nicht gespeichert werden. Hierfür ist eine gesonderte Prozedur erforderlich, die über **Ereignisse → Text modifiziert** ausgelöst wird:

```
001 SUB Pfad_Einlesen(oEvent AS OBJECT)
002    DIM oForm AS OBJECT
003    DIM oFeld AS OBJECT
004    DIM oFeld2 AS OBJECT
005    DIM stUrl AS STRING
006    oFeld = oEvent.Source.Model
007    oForm = oFeld.Parent
008    oFeld2 = oForm.getByName("GraphischesFeld")
009    IF oFeld.Text <> "" THEN
010       stUrl = ConvertToUrl(oFeld.Text)
011       oFeld2.BoundField.updateString(stUrl)
012    END IF
013 END SUB
```

Das auslösende Ereignis wird beim Aufruf der Prozedur mitgeliefert und hilft dabei, das Formular und auch das Feld zu finden, in dem der Pfad gespeichert werden soll. Mit **oEvent AS OBJECT** ist der Zugriff vor allem dann einfacher, wenn ein anderer User ein gleichlautendes Makro in irgendeinem Unterformular verwenden will. Das Dateiauswahlfeld ist von dort aus über **oEvent.Source.Model** zu erreichen. Das Formular ist **Parent** (Elternteil) zu dem Dateiauswahlfeld. Der Name des Formulars spielt also keine Rolle. Vom Formular aus geht dann der Zugriff auf das Feld mit dem Namen «GraphischesFeld». Über dieses Feld werden die Dateipfade der Bilder abgespeichert. Jetzt wird in das Feld die URL der ausgesuchten Datei geschrieben. Damit die URL betriebssystemunabhängig verwendet werden kann, wird der in dem Dateiauswahlfeld stehende Text vorher über **ConvertToUrl** in eine allgemeingültige URL-Schreibweise überführt.

In der Tabelle der Datenbank steht jetzt ein Pfad mit absoluter Schreibweise: **file:///...** .

Werden allerdings über ein graphisches Kontrollfeld Pfadeingaben eingelesen, so erfolgt die Aufnahme in relativer Pfadangabe. Dies ist von Vorteil, wenn die Datenbank in ein anderes System übertragen werden soll. So können z.B. in einem Unterverzeichnis der Datenbankdatei alle Bilder liegen und die Verbindung dazu wird auch auf anderen Rechner gefunden. Um dies mit der Dateiauswahl zu bewerkstelligen, müsste nachgebessert werden. Die Prozedur für diesen Zweck sieht wesentlich umfangreicher aus, da ein Vergleich der Pfadeingabe mit der aktuellen Speicherposition der Datenbankdatei nötig ist.

Dokumente mit relativer Pfadangabe verknüpfen

Das Dateiauswahlfeld liefert zuerst einmal nur den absoluten Pfad, der zwar für das Aufrufen der Datei gut nutzbar ist, aber eben den Transport der Datenbankdatei zusammen mit den Dokumenten erschwert. Die folgende Prozedur ist, wie im vorhergehenden Abschnitt, über **Ereignisse → Text modifiziert** an das Dateiauswahlfeld gebunden.[3]

```
001 SUB SaveFilePath(oEvent AS OBJECT)
002    DIM oField AS OBJECT
003    DIM oForm AS OBJECT
004    DIM oDoc AS OBJECT
005    DIM stUrlNew AS STRING
006    DIM stUrl AS STRING
007    DIM i AS INTEGER, ina AS INTEGER, inb AS INTEGER, inx AS INTEGER, iny AS INTEGER
008    oField = oEvent.Source.Model
009    oForm = oField.Parent
```

Nach der Deklaration der Variablen und dem Aufsuchen des Feldes und des Formulars wird erst einmal geklärt, ob das Feld überhaupt einen Inhalt aufweisen kann. Ohne Inhalt braucht auch nichts weiter eingelesen zu werden.

3 Dieses Makro ist in der Datenbank «Beispiel_Formular_Eingabekontrolle.odb» enthalten. Bei Verwendung des grafischen Kontrollfeldes ist der Einsatz des Makros nicht mehr nötig.

```
010     IF oField.Text <> "" THEN
011        stUrl = ConvertToUrl(oField.Text)
```

Bei dem Dateiauswahlfeld fehlt **file:///** am Anfang. Ansonsten ist dieser Pfad absolut. Der Pfad der Dateiauswahl wird jetzt mit dem Pfad der geöffneten Base-Datei verglichen.

```
012        oDoc = ThisComponent
013        a = split(oDoc.Parent.Url,"/")
014        b = split(stUrl,"/")
015        ina = UBound(a()) - 1
```

Beide Pfade werden zu Arrays aufgesplittet. Der Trenner ist der Frontslash. Der Dateiname der Base-Datei wird von der URL des Base-Dokumentes abgetrennt, da er bei der Pfadermittlung nicht mitgezählt werden darf.

Die Größe der Arrays wird verglichen und damit die maximale Anzahl der gleichen Elemente festgestellt.

```
016        inb = UBound(b())
017        IF ina > inb THEN
018           inx = inb
019        ELSE
020           inx = ina
021        END IF
```

Die beiden Arrays werden je Element miteinander verglichen. Die gleichen Elemente werden gezählt. Der Zähler wird um 1 erhöht, damit die gleichen Elemente anschließend beim Auslesen der Arrays nicht mehr berücksichtigt werden.

```
022        FOR i = 0 TO inx
023           IF a(i) = b(i) THEN
024              iny = i + 1
025           ELSE
026              EXIT FOR
027           END IF
028        NEXT
```

Für jedes Element, das nach dem Vergleich in der URL der Base-Datei liegt, wird ein Schritt aufwärts benötigt: **../**

```
029        FOR i = iny TO ina
030           stUrlNew = stUrlNew & "../"
031        NEXT
```

An die Aufwärtsschritte wird der verbleibende Pfad mit der ausgewählten Datei angehängt. Auch diese Schleife startet mit dem ersten ermittelten Unterschied der beiden Arrays.

```
032        FOR i = iny TO inb
033           stUrlNew = stUrlNew & b(i) & "/"
034        NEXT
```

Die Pfadangabe ist um einen Frontslash zu groß und wird daher um diesen Frontslash gekürzt. Anschließend wird der relative Pfad in das Zielfeld des Formulars übertragen. In dem aufrufenden Feld zur Dateiauswahl wird in den Zusatzinformationen (**Tag**) der Name des Tabellenfeldes der Datenquelle aufgeführt. So kann über das Tabellenfeld der Datenquelle direkt die neue URL in das Formular geschrieben werden – und dies unabhängig davon, ob das Formularfeld ein Textfeld ist, sich in einem Tabellenkontrollfeld befindet oder eventuell gar nicht in dem Formular sichtbar ist, sondern nur zur Datenquelle des Formulars gehört.

```
035        stUrlNew = Left(stUrlNew, len(stUrlNew) - 1)
036        oForm.updateString(oForm.findColumn(oField.Tag), stUrlNew)
037     END IF
038  END SUB
```

Verknüpfte Bilder und Dokumente anzeigen

Verknüpfte Bilder können direkt in dem kleinen graphischen Kontrollfeld angezeigt werden. Besser wäre aber ein größere Anzeige, um auch Details erkennen zu können.

Dokumente lassen sich standardmäßig in Base überhaupt nicht anzeigen.

Um dennoch eine Anzeige zu ermöglichen, bedarf es wieder einer Makrolösung. Dieses Makro wird über einen Button in dem Formular gestartet, in dem das graphische Kontrollfeld liegt.[4]

```
001 SUB Betrachten(oEvent AS OBJECT)
002    DIM oDoc AS OBJECT
003    DIM oForm AS OBJECT
004    DIM oFeld AS OBJECT
005    DIM oShell AS OBJECT
006    DIM stUrl AS STRING
007    DIM stFeld AS STRING
008    DIM arUrl_Start()
009    oDoc = thisComponent
010    oForm = oEvent.Source.Model.Parent
011    oFeld = oForm.getByName("GraphischesFeld")
012    stUrl = oFeld.BoundField.getString
```

Das graphische Kontrollfeld im Formular wird aufgesucht. Da in der Tabelle nicht das Bild selbst, sondern nur der Pfad als Text gespeichert wird, wird hier über **getString** dieser Text ausgelesen.

Anschließend wird der Pfad zu der Datenbankdatei ermittelt. Mit **oDoc.Parent** wird die *.odb-Datei erreicht. Sie ist der Container für die Formulare. Über **oDoc.Parent.Url** wird schließlich die gesamte URL incl. Dateinamen ausgelesen. Der Dateiname ist auch zu sehen in **oDoc.Parent.Title**. Der Text wird jetzt mit der Funktion **split** aufgetrennt, wobei als Trenner der Dateiname, umgewandelt in Url-Schreibweise, benutzt wird. Die Auftrennung gibt so nur als erstes und einziges Element des Arrays den Pfad zur *.odb-Datei wieder.

```
013    arUrl_Start = split(oDoc.Parent.Url,right(convertToUrl(oDoc.Parent.Title),
014       len(convertToUrl(oDoc.Parent.Title))-8))
015    oShell = createUnoService("com.sun.star.system.SystemShellExecute")
016    stFeld = convertToUrl(arUrl_Start(0) + stUrl)
017    oShell.execute(stFeld,,0)
018 END SUB
```

Externe Programme können über das **Struct com.sun.star.system.SystemShellExecute** gestartet werden. Dem externen Programm wird hier nur der Pfad zur Datei mitgegeben, der aus dem Pfad zur Datenbankdatei und dem intern gespeicherten relativen Pfad von der Datenbankdatei aus zusammengesetzt wurde. Die grafische Benutzeroberfläche des Betriebssystems entscheidet jetzt darüber, mit welchem Programm die entsprechende Datei zu öffnen ist.

Mit dem Kommando **oShell.execute** werden 3 Parameter übergeben. Als erstes wird eine ausführbare Datei oder der Pfad zu einer Datei aufgeführt, die im System mit einem Programm verbunden sind. Als zweites werden Parameter aufgeführt, mit denen das Programm gestartet werden soll. Als drittes wird über eine Ziffer mitgeteilt, wie mit Fehlermeldungen des Systems bei missglückter Ausführung umzugehen ist. Hier stehen 0 (Standard), 1 (keine Meldung anzeigen) und 2 (nur das Öffnen von absoluten URLs erlauben) zur Verfügung.

Dokumente in die Datenbank einlesen

Beim Einlesen der Dokumente sollten folgende Bedingungen immer im Auge behalten werden:[5]

- Je größer die Dokumente sind, desto schwerfälliger wird die Datenbank. Bei entsprechend großen Dokumenten, vor allem Bildern, ist deshalb eine externe Datenbank der internen Datenbank vorzuziehen.
- In Dokumenten kann ebenso wenig wie in Bildern recherchiert werden. Sie werden als Binärdaten gespeichert und können in einem Feld gespeichert werden, das einem Bildfeld entspricht..

4 Es ist auch möglich, so ein Ereignis durch eine Kombination van z.B. Strg + Maustaste auszulösen. Siehe dazu die Datenbank "Beispiel_Formular_Eingabekontrolle.odb"

5 Die Datenbank «Beispiel_Dokumente_einlesen_auslesen.odb» liegt diesem Handbuch bei.

- Dokumente, die in die Datenbank eingelesen werden, können nur über den Makroweg auch wieder ausgelesen werden. Ein SQL-Weg ist für die interne Datenbank nicht erreichbar.

Die folgenden Makros zum Ein-und Auslesen bauen dabei auf eine Tabelle auf, die neben der Datei im Binärformat eine Beschreibung der Datei und den ursprünglichen Dateinamen enthalten soll. Schließlich wird der Dateiname ja nicht mit abgespeichert und sollte beim Auslesen der Datei aber darauf Schlüsse zulassen, um welchen Dateityp es sich denn handelt. Nur dann kann die Datei auch zum Lesen durch andere Programme einwandfrei erkannt werden.

Die Tabelle enthält die folgenden Felder:

Feldname	***Feldtyp***	***Beschreibung***
ID	Integer	Die ID ist Primärschlüssel dieser Tabelle.
Beschreibung	Text	Beschreibung des Dokumentes, Suchbegriffe ...
Datei	Bild FIREBIRD: BLOB	Nimmt den binären Inhalt des Bildes oder der Datei auf.
DateiName	Text	Soll die Bezeichnung der Datei mit Dateiendung abspeichern. Wichtig für das spätere Auslesen.

Das Formular zum Einlesen und wieder Ausgeben der Dateien sieht so aus:

Solange sich Bilddateien in der Datenbank befinden, können diese Dateien auch in dem graphischen Kontrollfeld angesehen werden. Alle anderen Dateien mit Ausnahme der ersten Seiten von *.pdf-Dateien werden in dem Kontrollfeld nicht sichtbar.

Das folgende Makro für das Einlesen der Dateien wird über **Eigenschaften: Dateiauswahl → Ereignisse → Text modifiziert** ausgelöst.

```
001  SUB DateiEinlesen_mitName(oEvent AS OBJECT)
002     DIM oForm AS OBJECT
003     DIM oFeld AS OBJECT
004     DIM oFeld2 AS OBJECT
005     DIM oFeld3 AS OBJECT
006     DIM oStream AS OBJECT
007     DIM oSimpleFileAccess AS OBJECT
008     DIM stUrl AS STRING
009     DIM stName AS STRING
010     oFeld = oEvent.Source.Model
011     oForm = oFeld.Parent
012     oFeld2 = oForm.getByName("DateiName")
```

```
013     oFeld3 = oForm.getByName("GraphischesFeld")
014     IF oFeld.Text <> "" THEN
015        stUrl = ConvertToUrl(oFeld.Text)
016        ar = split(stUrl,"/")
017        stName = ar(UBound(ar))
018        oFeld2.BoundField.updateString(stName)
019        oSimpleFileAccess = createUnoService("com.sun.star.ucb.SimpleFileAccess")
020        oStream = oSimpleFileAccess.openFileRead(stUrl)
021        oFeld3.BoundField.updateBinaryStream(oStream, oStream.getLength())
022     END IF
023  END SUB
```

Da das Makro über das auslösende Ereignis die Position der anderen Formularfelder ermittelt, muss nicht besonders überprüft werden, ob die Felder nun in einem Formular oder Unterformular liegen. Alle Felder müssen lediglich im gleichen Formular positioniert sein.

Das Feld «DateiName» speichert den Namen der Datei ab, die ausgesucht wird. Bei Bildern muss dieser Name ohne ein zusätzliches Makro händisch eingegeben werden. Hier wird stattdessen der Dateiname aus der URL ermittelt und automatisch beim Einlesen der Datei mit eingefügt.

Das Feld «GraphischesFeld» speichert die Daten in dem gemeinsamen Feld für Bilder und Dateien ab.

Aus dem Dateiauswahlfeld wird über **oFeld.Text** der Pfad komplett mit Dateinamen ausgelesen. Damit die URL-Schreibweise nicht an systemspezifischen Bedingungen scheitert, wird der ausgelesene Text mit **ConvertToUrl** in eine allgemeingültige URL umgewandelt. Die so erstellte allgemeingültige URL wird in ein Array aufgeteilt. Der Trenner ist das /. Letztes Element dieser Pfadangabe ist der Dateiname. **Ubound(ar)** gibt die Nummer für das letzte Element an. Daher kann der Dateiname über **ar(Ubound(ar))** direkt ausgelesen und anschließend als String an das Feld übergeben werden.

Um die Datei selbst einzulesen, muss der **UnoService com.sun.star.ucb.SimpleFileAccess** bemüht werden. Über diesen Service kann der Inhalt der Datei als Datenstrom ausgelesen werden. Anschließend wird der so in dem Objekt **oStream** zwischengespeicherte Inhalt wiederum als Datenstrom in das Feld eingefügt, das mit dem Feld "Datei" der Tabelle verbunden ist. Dabei muss neben dem Objekt **oStream** auch die Länge des Datenstromes als Parameter angegeben werden.

Die Daten sind jetzt wie eine normale Eingabe in die Formularfelder eingefügt. Wird das Formular einfach geschlossen, so sind die Daten noch nicht abgespeichert. Die Speicherung erfolgt erst bei Betätigung des Speicherbuttons in der Navigationsleiste oder automatisch bei der Navigation zum nächsten Datensatz.

Bildnamen ermitteln

Bei dem obigen Verfahren wurde kurz erwähnt, dass der Name der Datei bei der Eingabe über das graphische Kontrollfeld so nicht ermittelt werden kann. Hier jetzt kurz ein Makro zur Ermittlung des Dateinamens, das zum obigen Formular passt. Der Dateiname lässt sich nicht sicher durch ein Ereignis ermitteln, das mit dem grafischen Kontrollfeld direkt verbunden ist. Deswegen wird das Makro über **Formular-Eigenschaften → Ereignisse → Vor der Datensatzaktion** gestartet.

```
001  SUB BildnamenAuslesen(oEvent AS OBJECT)
002     oForm = oEvent.Source
003     IF InStr(oForm.ImplementationName, "ODatabaseForm") THEN
004        oFeld = oForm.getByName("GraphischesFeld")
005        oFeld2 = oForm.getByName("DateiName")
006        IF oFeld.ImageUrl <> "" THEN
007           stUrl = ConvertToUrl(oFeld.ImageUrl)
008           ar = split(stUrl,"/")
009           stName = ar(UBound(ar))
010           oFeld2.BoundField.updateString(stName)
011        END IF
```

```
012     END IF
013 END SUB
```

Vor der Datensatzaktion werden zwei Implementationen mit unterschiedlichem Implementationsnamen ausgeführt. Das Formular ist am einfachsten über die Implementation erreichbar, das in seinem Namen als **ODatabaseForm** bezeichnet wird.

In dem graphischen Kontrollfeld ist die URL der Datenquelle über die **ImageUrl** erreichbar. Diese URL wird ausgelesen, der Dateinamen wie in der vorhergehenden Prozedur «DateiEinlesen_mitName» ausgelesen und in das Feld «DateiName» übertragen.

Bildnamen aus dem Speicher entfernen

Wird nach dem Ablauf des obigen Makros zum nächsten Datensatz gewechselt, so ist der Pfad zu dem ursprünglichen Bild weiter vorhanden. Würde jetzt eine allgemeine Datei über das Dateiauswahlfeld eingelesen, so würde der Name der Datei ohne das folgende Makro einfach durch den Namen der zuletzt eingelesenen Bilddatei überschrieben.

Der Pfad kann leider nicht mit dem vorhergehenden Makro entfernt werden, da das Einlesen der Bilddatei erst beim Abspeichern erfolgt. Eine Entfernung des Pfades vor der Abspeicherung löscht das Bild.

Das Makro wird über **Formular-Eigenschaften → Ereignisse → Nach der Datensatzaktion** gestartet.

```
001 SUB BildnamenZuruecksetzen(oEvent AS OBJECT)
002    oForm = oEvent.Source
003    IF InStr(oForm.ImplementationName, "ODatabaseForm") THEN
004       oFeld = oForm.getByName("GraphischesFeld")
005       IF oFeld.ImageUrl <> "" THEN
006          oFeld.ImageUrl = ""
007       END IF
008    END IF
009 END SUB
```

Es wird, wie in der Prozedur «BildnamenAuslesen», auf das graphische Kontrollfeld zugegriffen. Existiert dort noch ein Eintrag in der **ImageUrl**, dann wird dieser geleert.

Bilder und Dokumente auslesen und anzeigen

Für Dateien wie für die Bilder in Originalgröße gilt, dass der Button Datei mit externem Programm betrachten ausgelöst werden muss. Dann werden die Dateien in das temporäre Verzeichnis ausgelesen und anschließend über das mit der Endung verknüpfte Programm des Betriebssystems angezeigt.

Das Makro wird über **Eigenschaften: Schaltfläche → Ereignisse → Aktion ausführen** gestartet.

```
001 SUB DateiAuslesen_mitName(oEvent AS OBJECT)
002    DIM oForm AS OBJECT
003    DIM oFeld AS OBJECT
004    DIM oFeld2 AS OBJECT
005    DIM oStream AS OBJECT
006    DIM oShell AS OBJECT
007    DIM oPath AS OBJECT
008    DIM oSimpleFileAccess AS OBJECT
009    DIM stName AS STRING
010    DIM stPfad AS STRING
011    DIM stFeld AS STRING
012    oForm = oEvent.Source.Model.Parent
013    oFeld = oForm.getByName("GraphischesFeld")
014    oFeld2 = oForm.getByName("DateiName")
015    stName = oFeld2.Text
016    IF stName = "" THEN
017       stName = "Datei"
018    END IF
```

```
019     oStream = oFeld.BoundField.getBinaryStream
020     oPath = createUnoService("com.sun.star.util.PathSettings")
021     stPfad = oPath.Temp & "/" & stName
022     oSimpleFileAccess = createUnoService("com.sun.star.ucb.SimpleFileAccess")
023     oSimpleFileAccess.writeFile(stPfad, oStream)
024     oShell = createUnoService("com.sun.star.system.SystemShellExecute")
025     stFeld = convertToUrl(stPfad)
026     oShell.execute(stFeld,,0)
027 END SUB
```

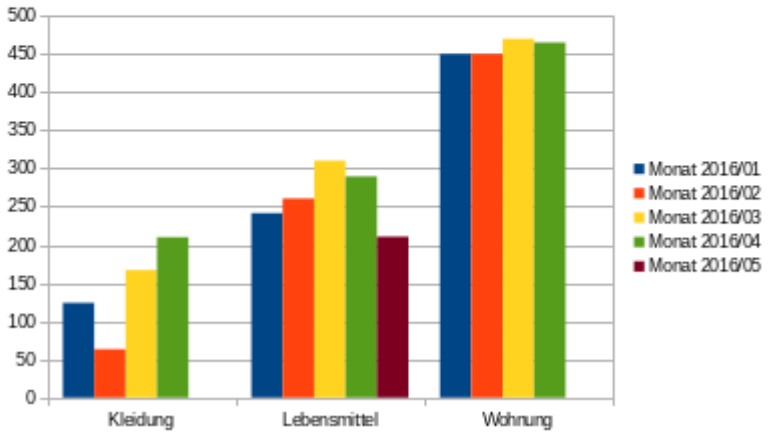
Die Lage der anderen betroffenen Felder im Formular wird abhängig von dem auslösenden Button ermittelt. Fehlt ein Dateiname, so wird der ausgelesenen Datei einfach der Name «Datei» zugeschrieben.

Der Inhalt des Formularfeldes «GraphischesFeld» entspricht dem Inhalt, der in dem Feld "Datei" der Tabelle liegt. Er wird als Datenstrom ausgelesen. Als Pfad für die ausgelesene Datei wird der Pfad zum temporären Verzeichnis genutzt, der in LibreOffice über **Extras → Optionen → LibreOffice → Pfade** eingestellt werden kann. Soll also die Datei anschließend noch anderweitig verwendet und nicht nur angezeigt werden, so ist sie aus diesem Pfad heraus auch kopierbar. Innerhalb des Makros wird die Datei direkt nach dem erfolgreichen Auslesen mit dem Programm geöffnet, das in der grafischen Benutzeroberfläche des Betriebssystems mit dem Dateityp verbunden ist.

Diagramme in Formulare einbinden

Das Einbinden von Diagrammen in Formulare ist von der GUI her nicht vorgesehen. Mit einem Umweg über Writer, etwas Makrohilfe und allgemein verarbeitbaren Ansichten lässt sich so ein Diagramm dennoch in ein Formular integrieren und darüber hinaus beständig an die momentane Datenlage anpassen.

ID	Kategorie	Datum	Betrag
1	Lebensmittel	15.01.16	241,51 €
2	Lebensmittel	15.02.16	260,75 €
3	Kleidung	15.01.16	124,00 €
4	Wohnung	15.01.16	450,00 €
5	Kleidung	15.02.16	63,20 €
6	Wohnung	15.02.16	450,00 €
7	Lebensmittel	15.03.16	310,23 €
8	Kleidung	15.03.16	167,00 €
9	Wohnung	15.03.16	470,00 €
10	Lebensmittel	15.04.16	289,50 €
11	Kleidung	15.04.16	210,00 €
12	Wohnung	15.04.16	465,00 €
13	Lebensmittel	15.05.16	210,76 €



Als Datenbank für die Diagrammdarstellung wurden beispielhaft Daten aus einer Haushaltsführung und Daten aus einer Temperaturmessung über einen kürzeren Zeitraum verwandt.[6] Während sich die Haushaltsführung für verschiedene Diagrammtypen eignet, ist die Messung eines Temperaturverlaufes auf ein XY-Diagramm zugeschnitten.

6 Die Datenbank «Beispiel_Baseformular_mit_Diagramm» ist den Beispieldatenbanken für dieses Handbuch beigefügt.

Diagramme aus dem Writer importieren

Zuerst wird ein neues Writer-Dokument erstellt: **Datei → Neu → Textdokument**. Anschließend wird über die Symbolleiste oder über **Einfügen → Objekt → Diagramm** ein Diagramm erstellt. Standardmäßig wird hier zuerst einmal ein Säulendiagramm in das Dokument eingefügt.

Das Diagramm wird kopiert und in ein Formular der Datenbank eingefügt. Alle weiteren Einstellungen des Diagramms können auch im Base-Formular über das Kontextmenü vorgenommen werden.

Daten für das Diagramm werden später automatisch eingelesen. Es ist also nicht notwendig, hier irgendeine Voreinstellung vorzunehmen.

Abfragen erstellen und als Ansichten speichern

Für die Darstellung der verschiedenen Diagrammtypen werden Zeilenbeschreibungen (**RowDescriptions**), Spaltenbeschreibungen (**ColumnDescriptions**) und Daten (**Data**) benötigt. Um Probleme bei der Darstellung von XY-Diagrammen zu umgehen, wird zusätzlich eine Information zum Diagrammtyp in den Abfragen dargestellt.

Abfrage für ein Säulendiagramm

Würde von vornherein sichergestellt, dass nur ein Eintrag pro Monat erstellt würde und das Diagramm nur nach allen Einträgen für den Monat aktualisiert werden soll, so könnte für ein **Säulendiagramm** einfach die Tabelle direkt abgefragt werden:

```
001 SELECT "Kategorie", "Datum", "Betrag", 'BC' AS "Type" FROM "Kategorien"
```

Die Kategorien stellen die Zeilenbeschriftungen dar. Die Zeilenbeschriftungen erscheinen unter den Säulen des Diagramms. Für jeden Monat wird eine neue Säule dargestellt. Die Monatsbezeichnungen erscheinen beim Säulendiagramm als Spaltenbeschriftungen rechts vom Diagramm in der Legende. Der Betrag gibt als Dezimalzahl die Höhe der jeweiligen Säule an. Die Information zum Diagrammtypen wird später nur für das XY-Diagramm separat ausgewertet. 'BC' steht hier schlicht für «bar chart» (Säulendiagramm).

Damit auch mehrere Einträge pro Monat angefertigt werden können, sind die Einträge für alle gleichen Kategorien innerhalb eines Monats zusammen zu fassen. Die Darstellung des Monats soll in dem Diagramm z.B. als «Monat 2016/02» erfolgen. Außerdem sollen die Einträge in dem Diagramm auch dann aktualisiert werden, wenn für eine Kategorie eventuell noch gar kein Monatseintrag existiert.

Um all dies zu bewerkstelligen, wird zuerst einmal eine Kombination aller Kategorien mit allen in der Tabelle "Kategorien" vorhandenen Monaten erstellt:

```
001 SELECT
002    "a"."Kategorie",
003    "b"."Monat"
004 FROM "Kategorien" AS "a",
005    ( SELECT 'Monat ' || YEAR( "Datum" ) || '/' || RIGHT( '0'
006       || MONTH( "Datum" ), 2 ) AS "Monat" FROM "Kategorien" ) AS "b"
007 ORDER BY "Kategorie", "Monat"
```

Diese Konstruktion kann allerdings den Datenbestand deutlich aufblähen. Hat die Tabelle "Kategorien" 5 Einträge, so werden bei dieser Abfrage 5 Einträge zu "Kategorie" mit 5 Einträgen zu der Monatsdarstellung kombiniert. Es entstehen zwangsläufig 25 Einträgen, von denen dann gewiss viele Einträge mehrfach vorkommen.

Hinweis

Für Firebird muss statt **`MONTH("Datum")` `EXTRACT(MONTH FROM "Datum")`** usw. eingefügt werden. Firebird kennt die hier verwendeten Kurzformen nicht.

Mit Hilfe des Zusatzes **DISTINCT** kann die Ausgabe der Einträge eingeschränkt werden. Dies sollte möglichst schon bei den Datenquellen für die Abfrage ansetzen, damit nicht unnötig erst Daten geladen und wieder aus dem Speicher entfernt werden. Deshalb wird auf die Tabelle "Kategorien" zweimal mit einer Abfrage und dem Zusatz **DISTINCT** zugegriffen:

```
001 SELECT
002    "a"."Kategorie",
003    "b"."Monat"
004 FROM ( SELECT DISTINCT "Kategorie" FROM "Kategorien" ) AS "a",
005    ( SELECT DISTINCT 'Monat ' || YEAR( "Datum" ) || '/' || RIGHT( '0'
006       || MONTH( "Datum" ), 2 ) AS "Monat" FROM "Kategorien" ) AS "b"
007 ORDER BY "Kategorie", "Monat"
```

Mit Hilfe einer korrelierenden Unterabfrage wird jetzt zu jeder Kombination von "Monat" und "Kategorie" der entsprechende "Betrag" aufsummiert. So werden auch mehrere Einträge im Monat bei der Darstellung im Diagramm berücksichtigt:

```
001 SELECT
002    "a"."Kategorie" AS "RowDescription",
003    "b"."Monat" AS "ColumnDescription",
004    ( SELECT SUM( "Betrag" ) FROM "Kategorien" WHERE 'Monat ' ||
005       YEAR( "Datum" ) || '/' || RIGHT( '0' || MONTH( "Datum" ), 2 ) =
006       "b"."Monat" AND "Kategorie" = "a"."Kategorie" ) AS "Data",
007    'BC' AS "Type"
008 FROM ( SELECT DISTINCT "Kategorie" FROM "Kategorien" ) AS "a",
009    ( SELECT DISTINCT 'Monat ' || YEAR( "Datum" ) || '/' || RIGHT( '0'
010       || MONTH( "Datum" ), 2 ) AS "Monat" FROM "Kategorien" ) AS "b"
011 ORDER BY "Kategorie", "Monat"
```

Diese Abfrage wird abgespeichert, mit der rechten Maustaste angeklickt und über **Abfrage → Als Ansicht erstellen** unter der Bezeichnung «Chart_Kategorie» als Ansicht gespeichert.

Abfrage für ein Kreisdiagramm

Für die Darstellung eines **Kreisdiagramms** ist es erforderlich, dass statt der einzelnen Monatsdarstellungen die Summe über alle Monate erstellt wird. Hier wird der Einfachheit halber direkt auf die eben erstellte Ansicht zugegriffen:

```
001 SELECT
002    "RowDescription",
003    '' AS "ColumnDescription",
004    SUM( "Data" ) AS "Data",
005    'CC' AS "Type"
006 FROM "Chart_Kategorie"
007 GROUP BY "RowDescription"
```

Nach der "Kategorie", hier unter dem Alias "RowDescription", wird die Abfrage gruppiert. Für das Feld "Data" wird die Summe des Feldes "Data" aus der Ansicht "Chart_Kategorie", bezogen auf diese Gruppierung, erstellt.

Eine Spaltenbeschreibung ist nicht erforderlich und wird mit einem leeren Text versehen. Als Typ des Diagramms wird 'CC' für «circle chart» (Kreisdiagramm) angegeben.

Die Abfrage wird abgespeichert, mit der rechten Maustaste angeklickt und über **Abfrage → Als Ansicht erstellen** unter der Bezeichnung «Chart_Kategorie_Circle» als Ansicht gespeichert.

Abfrage für ein XY-Diagramm

Daten für ein **XY-Diagramm** liegen in der Regel in anderer Weise vor als für Säulendiagramme oder Kreisdiagramme. Statt einen Wert in einer Zeile zu speichern, werden hier Wertepaare gespeichert. Damit dennoch auf die Daten mit der gleichen Makrokonstruktion zugegriffen werden kann, ist eine besondere Abfragekonstruktion vorgesehen, die wieder in der ersten Spalte

die Zeilenbeschreibung, in der zweiten Spalte die Spaltenbeschreibung, in der dritten Spalte die Daten und in der vierten Spalte den Diagrammtyp wiedergeben kann.

```
001 SELECT
002    "Zeit" AS "RowDescription",
003    '' AS "ColumnDescription",
004    "Zeit" AS "Data",
005    'XY' AS "Type"
006 FROM "Temperaturverlauf"
007 UNION
008 SELECT
009    "Zeit" AS "RowDescription",
010    'Temperatur [°C]' AS "ColumnDescription",
011    "Temperatur" AS "Data",
012    'XY' AS "Type"
013 FROM "Temperaturverlauf"
014 ORDER BY "RowDescription" ASC, "ColumnDescription" ASC
```

Zuerst werden alle Zeiten ausgelesen. Das Ergebnis dieser Abfrage wird mit der Folgeabfrage über **UNION** kombiniert, die jetzt alle Temperaturdaten ausliest. Leider funktioniert diese Kombination nicht wie gewünscht, da die Zeit einen anderen Datentypen hat als die Temperatur. Es kommt noch hinzu, dass mit dem Datentypen für die Zeit leider auch keine kontinuierliche Darstellung in einem XY-Diagramm möglich ist. Aus der Zeit muss durch Umformung ein Dezimalwert erstellt werden. Dies erfolgt, indem der Tag als Grundmaß angesehen wird. Stunden, Minuten und Sekunden werden als Bruchteile des Tages errechnet und addiert.

```
001 SELECT
002    HOUR( "Zeit" ) / 24.00000 + MINUTE( "Zeit" ) / 1440.00000 +
003       SECOND( "Zeit" ) / 86400.00000 AS "RowDescription",
004    '' AS "ColumnDescription",
005    HOUR( "Zeit" ) / 24.00000 + MINUTE( "Zeit" ) / 1440.00000 +
006       SECOND( "Zeit" ) / 86400.00000 AS "Data",
007    'XY' AS "Type"
008 FROM "Temperaturverlauf"
009 UNION
010 SELECT
011    HOUR( "Zeit" ) / 24.00000 + MINUTE( "Zeit" ) / 1440.00000 +
012       SECOND( "Zeit" ) / 86400.00000 AS "RowDescription",
013    'Temperatur [°C]' AS "ColumnDescription",
014    "Temperatur" AS "Data",
015    'XY' AS "Type"
016 FROM "Temperaturverlauf"
017 ORDER BY "RowDescription" ASC, "ColumnDescription" ASC
```

Hinweis

Für Firebird muss statt **HOUR("Zeit") / 24.00000 + MINUTE("Zeit") / 1440.00000 +SECOND("Zeit") / 86400.00000** einfach **("Zeit" – TIME '00:00')/86400.00000** eingefügt werden. Firebird kennt die hier verwendeten Kurzformen nicht, kann aber Datumswerte und Zeitwerte voneinander subtrahieren.

Die Zeilenbeschriftung "RowDescription" wird für das XY-Diagramm gar nicht benötigt. Hier stehen schließlich die Daten der X-Achse. Diese Spalte wird über die zu einer Dezimalzahl umformatierten Inhalte dazu genutzt, die Zeilen in der korrekten Reihenfolge zu sortieren.

Das Diagramm stellt lediglich die Werte der Y-Achse als 'Temperatur [°C]' dar. Deswegen wird für die Zeilen, die jetzt die Zeitangaben als Daten wiedergeben, eine leere Spaltenbeschriftung "ColumnDescription" ausgegeben.

Der gesamte Inhalt wird nach den Zeiten in "RowDescription" und den Einträgen in "ColumnDescription" sortiert, so dass die Werte für X- und Y-Achse direkt aufeinander folgen und durch gleiche "RowDescription"-Werte einander zugeordnet werden können.

Diagramme über ein Makro anpassen

Um dem Makro mitteilen zu können, welche Ansicht in dem jeweiligen Diagramm dargestellt werden soll, wird in den Formularen ein **verstecktes Steuerelement** eingebaut. Über den Formularnavigator wird das Formular, das als Auslöser des Diagramms genutzt werden soll, mit der rechten Maustaste angeklickt. Im Kontextmenü wird **Neu → Verstecktes Steuerelement** aufgerufen.

Das Element wird über das Makro mit dem Namen 'Chart' gesucht. Entsprechend wird das versteckte Steuerelement erst einmal umbenannt: **rechte Maustaste → Kontextmenü → Umbenennen**. Anschließend wird in dem Steuerelement über r**echte Maustaste → Kontextmenü → Eigenschaften → Allgemein → Zusatzinformation** der Name für die Ansicht vermerkt.

Jetzt ist nur noch notwendig, das folgende Makro mit den Ereignissen des Formulars **Beim Laden** bzw. **Nach der Datensatzaktion** zu verknüpfen.

```
001 SUB ChangeData(oEvent AS OBJECT)
002    DIM oDiag AS OBJECT
003    DIM oDatasource AS OBJECT
004    DIM oConnection AS OBJECT
005    DIM oSQL_Command AS OBJECT
006    DIM oResult AS OBJECT
007    DIM oForm AS OBJECT
008    DIM oHiddenControl AS OBJECT
009    DIM stSql AS STRING
010    DIM stRow AS STRING
011    DIM stType AS STRING
012    DIM i AS INTEGER
013    DIM k AS INTEGER
014    DIM x AS INTEGER
015    DIM n AS INTEGER
016    DIM aNewData(0)
017    DIM aNewRowDescription(0)
018    DIM aTmp() AS DOUBLE
019    DIM aNewColumnDescription()
020    DIM aType()
021    DIM arView()
022    DIM arDiag()
023    oForm = oEvent.Source
024    oHiddenControl = oForm.getByName("Chart")
025    arView = Split(oHiddenControl.Tag,",")
026    arDiag = Split(oHiddenControl.HiddenValue,",")
027    FOR n = LBound(arView()) TO UBound(arView())
028       stView = oHiddenControl.Tag
029       stSql = "SELECT * FROM """+arView(n)+""""
030       oDatasource = ThisComponent.Parent.CurrentController
031       IF NOT (oDatasource.isConnected()) THEN
032          oDatasource.connect()
033       END IF
034       oConnection = oDatasource.ActiveConnection()
035       oSQL_Command = oConnection.createStatement()
036       oResult = oSQL_Command.executeQuery(stSql)
037       i = 0
038       k = 0
039       x = 0
040       WHILE oResult.next
041          stRow = oResult.getString(1)
042          stType = oResult.getString(4)
043          IF aNewRowDescription(i) = stRow THEN
044             ReDim Preserve aNewColumnDescription(k)
```

```
045             ReDim Preserve aTmp(k)
046          ELSE
047             IF x > 0 THEN
048                i = i + 1
049             ELSE
050                x = 1
051             END IF
052             ReDim Preserve aNewRowDescription(i)
053             ReDim Preserve aNewData(i)
054             k = 0
055             ReDim Preserve aNewColumnDescription(k)
056             ReDim aTmp(k)
057          END IF
058          aNewRowDescription(i) = stRow
059          aNewColumnDescription(k) = oResult.getString(2)
060          aTmp(k) = oResult.getDouble(3)
061          aNewData(i) = aTmp()
062          k = k + 1
063       WEND
064       oDiag = thisComponent.EmbeddedObjects.getByName(arDiag(n))
065       oXCOEO = oDiag.ExtendedControlOverEmbeddedObject
066       oXCOEO.changeState(4)
067       IF stType <> "XY" THEN
068          oDiag.model.Data.setData(aNewData)
069       END IF
070       oDiag.model.DataProvider.setData(aNewData)
071       oDiag.model.DataProvider.setRowDescriptions(aNewRowDescription)
072       oDiag.model.DataProvider.setColumnDescriptions(aNewColumnDescription)
073       oDiag.Component.setmodified(true)
074       oDiag.Component.update()
075       oXCOEO.changeState(1)
076    NEXT
077 END SUB
```

Nach der Deklaration der Variablen wird zuerst aus den Zusatzinformationen (**Tag**) des versteckten Kontrollfeldes der Name für die Ansicht ausgelesen. Da hier mehrere Namen für mehrere Diagramme gespeichert werden können werden die Namen durch Komma getrennt und über **Split** in ein Array eingelesen. Gleiches gilt für die Namen der Diagramme, die im **HiddenValue** in der entsprechenden Reihenfolge eingetragen sind. Anschließend läuft eine Schleife über das erste Array ab, das ja genau so viele Elemente enthält wie das zweite Array. Der erforderliche SQL-Code zum Auslesen der gesamten Ansicht wird formuliert, die Verbindung zur Datenbank, falls erforderlich, hergestellt und der SQL-Code an die Datenbank weitergeleitet. In **oResult** wird das Ergebnis der Abfrage gespeichert.

Vor dem Start der Schleife zur Ermittlung des Inhaltes aus **oResult** werden noch einige Zahlenvariablen auf '0' gesetzt, die im Laufe der Schleife verändert werden sollen.

Innerhalb der Schleife werden die Inhalte der verschiedenen Spalten der Ansicht ausgelesen. Die Inhalte werden in unterschiedliche Arrays zur Weitergabe an den DataProvider abgespeichert. Hier werden wieder Daten (**Data**), Zeilenbeschriftungen (**RowDescriptions**) und Spaltenbeschriftungen (**ColumnDescriptions**) unterschieden.

Mit **oResult.getString(1)** wird das Ergebnis zum jeweiligen Datensatz aus der ersten Spalte als Text ausgelesen. Hier handelt es sich um Zeilenbeschriftungen, die eben einfach Text sind. Solange die Abfrage nacheinander gleiche Zeilenbeschriftungen liefert, werden alle weitere Inhalte dem gleichen Datenbestand zugeordnet. Um dies zu gewährleisten, erfolgt zuerst eine Abfrage, ob der entsprechende Eintrag von **stRow** bereits als letzter Eintrag von **aNewRowDescriptions**() vorhanden ist.

Ist dies nicht der Fall, wie z.B. direkt beim Einlesen des ersten Datensatzes, dann erfolgt das Vorgehen, das unter **ELSE** beschrieben ist. Nur wenn der Zähler **x** größer als '0' ist, wird der Zähler für **i** heraufgesetzt. Dies soll vermeiden, dass der erste Eintrag des zu erzeugenden Arrays später leer ist. Für alle späteren Eintritte in diese Schleife wird allerdings dann **x** auf '1'

gesetzt, so dass **i** heraufgesetzt wird und die Arrays mit einem weiteren Datensatz beschrieben werden können.

Redim Preserve sichert den bisherigen Inhalt eines Arrays und eröffnet gleichzeitig die Möglichkeit, einen zusätzlichen Eintrag ans Ende des Arrays anzufügen.

aNewData und **aNewRowDescriptions** werden immer zusammen abgespeichert. Deswegen haben die Arrays den gemeinsamen Zähler **i**. **aNewColumnDescriptions** sowie die in einem temporären Array **aTmp** gespeicherten Dateninhalte werden bei jedem neuen Durchgang durch die **WHILE**-**NEXT**–Schleife mit einem zusätzlichen Datensatz versehen. Auch sie haben deshalb einen gemeinsamen Zähler, hier **k**. Dieser Zähler wird bei jeder Änderung von **aNewRowDescriptions** neu mit '0' gestartet. Dabei wird das temporäre Array nicht gesichert, da es zwischenzeitig in dem Array **aNewData** abgespeichert wurde.

Die Inhalte für das Array **aTmp** werden immer als Dezimalzahlen aus der Abfrage ausgelesen. Daher der Eintrag **oResult.getDouble(3)**.

Damit ein Diagramm im Formular kontinuierlich geändert werden kann, muss es zuerst einmal aktiviert werden. Das Diagramm selbst ist dem Formular sonst völlig unbekannt. Das Diagramm wird über den Namen ausgesucht. Anschließend wird der Controller für das eingebettete Diagramm angesprochen.Zuerst wird mit '4' die UI aktiviert. Die möglichen Parameter sind unter **com.sun.star.embed.EmbedStates**: LOADED = 0, RUNNING = 1, ACTIVE = 2, INPLACE_ACTIVE = 3, UI_ACTIVE = 4.

Nachdem alle Daten ausgelesen wurden, werden diese in den **DataProvider** übertragen. Mit **oDiag.model.Data.setData(aNewData)** werden nicht nur die Daten neu geschrieben, sondern z.B. auch die Anzahl der Säulen in einem Spaltendiagramm aktualisiert. Dieser Eintrag führt allerdings bei einem XY-Diagramm dazu, dass hier die Grundstruktur des Diagramms zerstört und der erste Dateneintrag nicht als X-Achsenwert gesehen wird. Deshalb ist dieser Eintrag für XY-Diagramme ausgeschlossen.

Nach der Änderung wird der Status des Diagramms auf '1' gesetzt, da ansonsten das Diagramm die ganze Zeit im Bearbeitungsmodus erscheint.

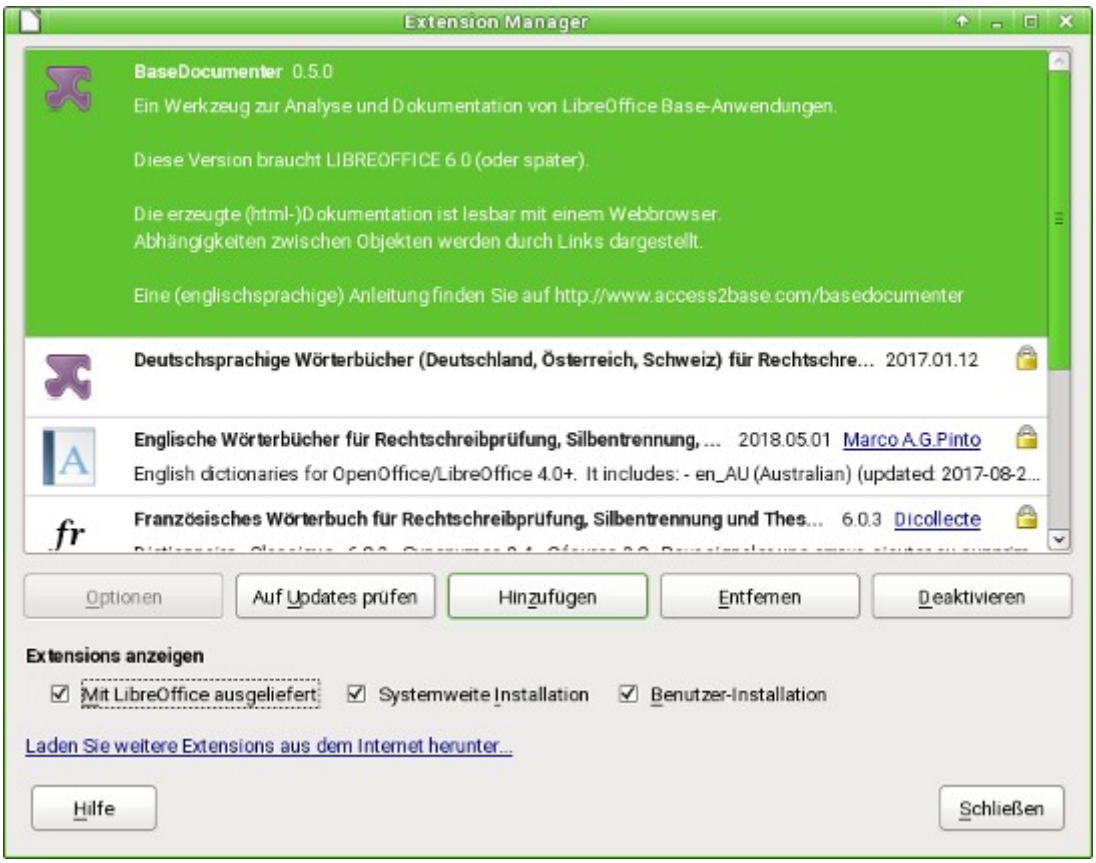
Übersicht über die Datenbank: BaseDocumenter – Extension

Wem geht es nicht oft so, dass irgendwann der Überblick über die ganzen Details einer Datenbank nützlich wären. Hier hilft die Erweiterung «BaseDocumenter»[7] von Jean-Pierre Ludure, die eine Übersicht in Form von HTML-Dateien erzeugt. Diese Erweiterung erscheint dann als zusätzliches Menü in jedem Base-Fenster.

Hinweis

Diese Erweiterung funktioniert zur Zeit (LO 7.2) nur mit der internen HSQLDB, nicht mit der internen ~~FIREBIRD~~ Datenbank.

7 Siehe: *http://www.access2base.com/basedocumenter/basedocumenter.html*

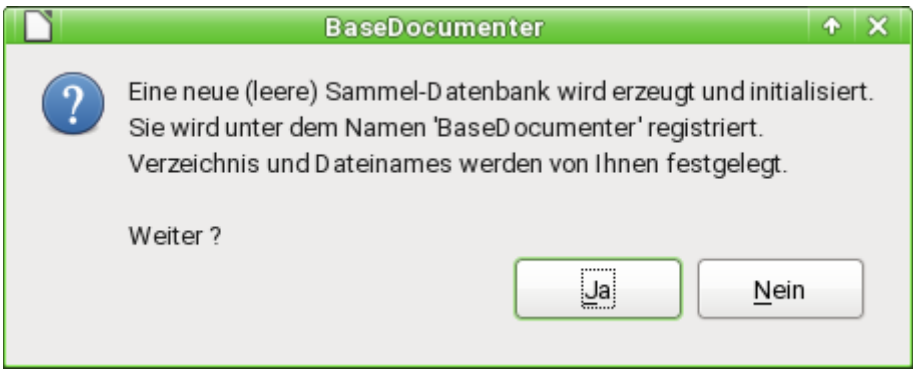


Zuerst wird über **Extras → Extension Manager** der BaseDocumenter hinzugefügt. Der BaseDocumenter benötigt mindestens eine LO-Version 6.0.

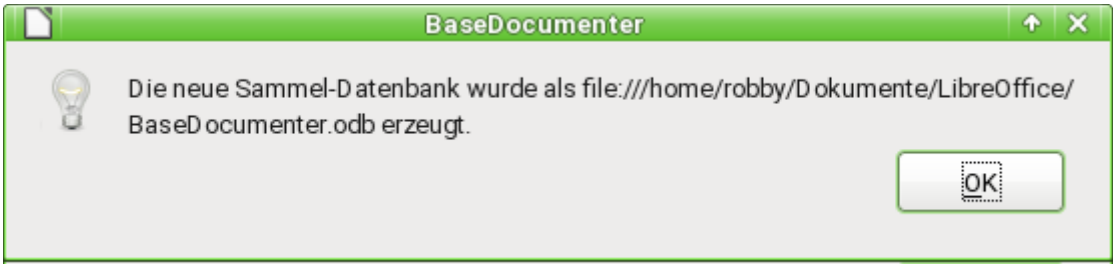
Die abzuspeichernden Informationen für die Datenbanken benötigen ein Verzeichnis, in dem sie abgelegt werden können. Dieses Verzeichnis muss zuerst erstellt werden.

In diesem Verzeichnis sollte außerdem css-Dateien («Cascading Style Sheets»), Javascripte, Logos und Vorlagen für die Darstellung der HTML-Dateien abgespeichert werden. Ein erstes Beispiel steht hier zum Download bereit: *http://www.access2base.com/basedocumenter/_download/Templates.zip* . Diese *.zip-Datei muss entpackt und anschließend komplett in das erstellte Verzeichnis kopiert werden.

BaseDocumenter → Neue Sammel-DB erstellt eine Datenbank, in der die notwendigen Informationen gespeichert werden. Diese Datenbank hat für den normalen Nutzer keine weitere Bedeutung. Sie braucht anschließend auch nicht mit **Öffne Sammel-DB** aufgesucht zu werden.

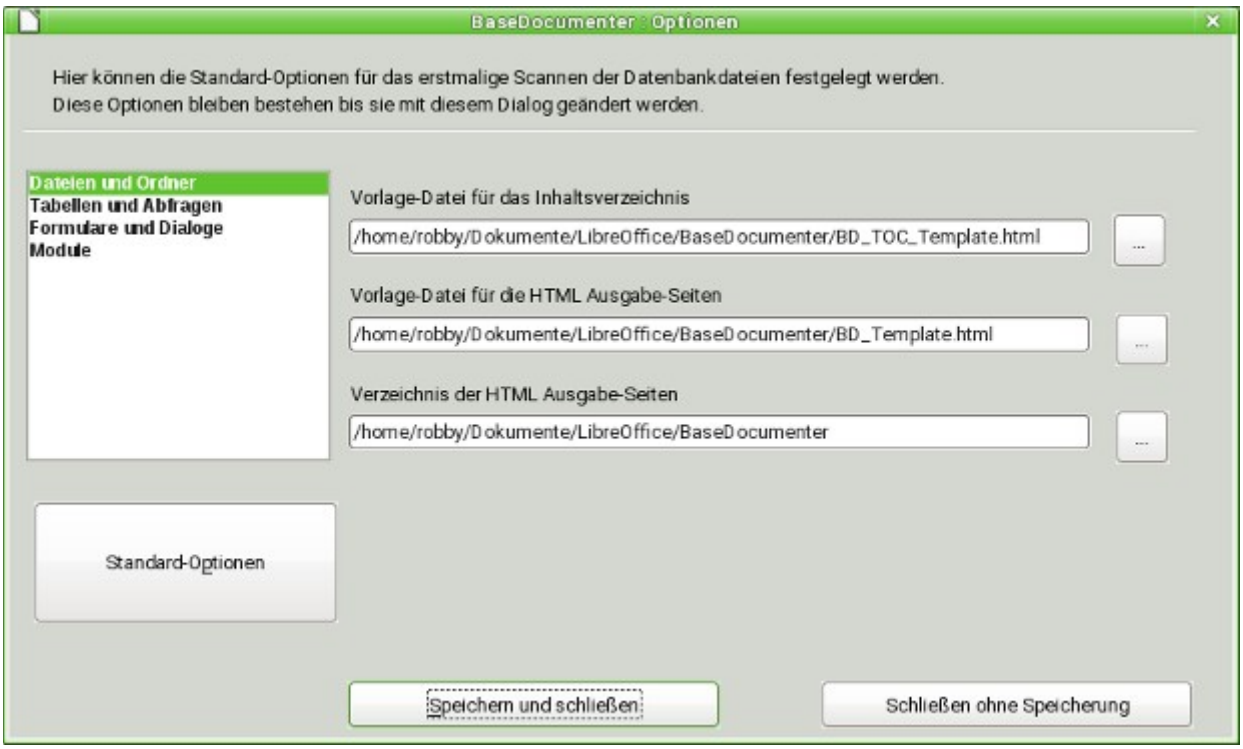


Die Sammeldatenbank wird erstellt und als «BaseDocumenter» in **Extras → Optionen → LibreOffice Base → Datenbanken** registriert.



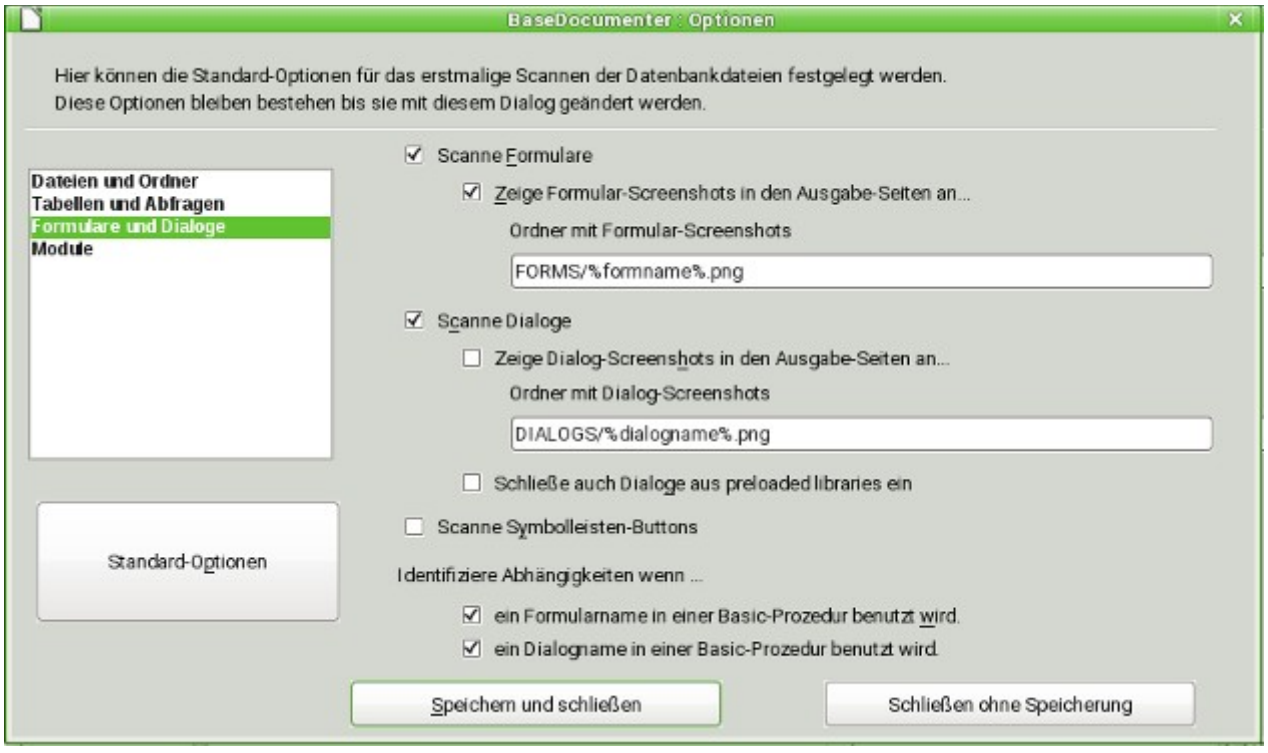
Die in dieser Sammeldatenbank erstellten Tabellen dürfen nicht geändert werden. Es könnten aber sehr wohl z.B. Abfragen hinzugefügt werden. Für den Normalgebrauch ist dies aber nicht notwendig.

BaseDocumenter → Optionen dient zuerst einmal zur Einstellung der Pfade für die gerade abgespeicherten Vorlagedateien und die abzuspeichernden Informationen der Datenbank. In diesem Dialog kann auch detailliert ausgewählt werden, welche Elemente in den HTML-Dateien dargestellt werden sollen. Hier ist es auch möglich, Screenshots von Formularen und eventuell erstellten Dialogen anfertigen zu lassen.



Die Pfade werden auf das vorher erstellte Verzeichnis und die darin befindlichen Vorlagen für das Inhaltsverzeichnis und der HTML-Ausgabe der einzelnen Dateien eingestellt.

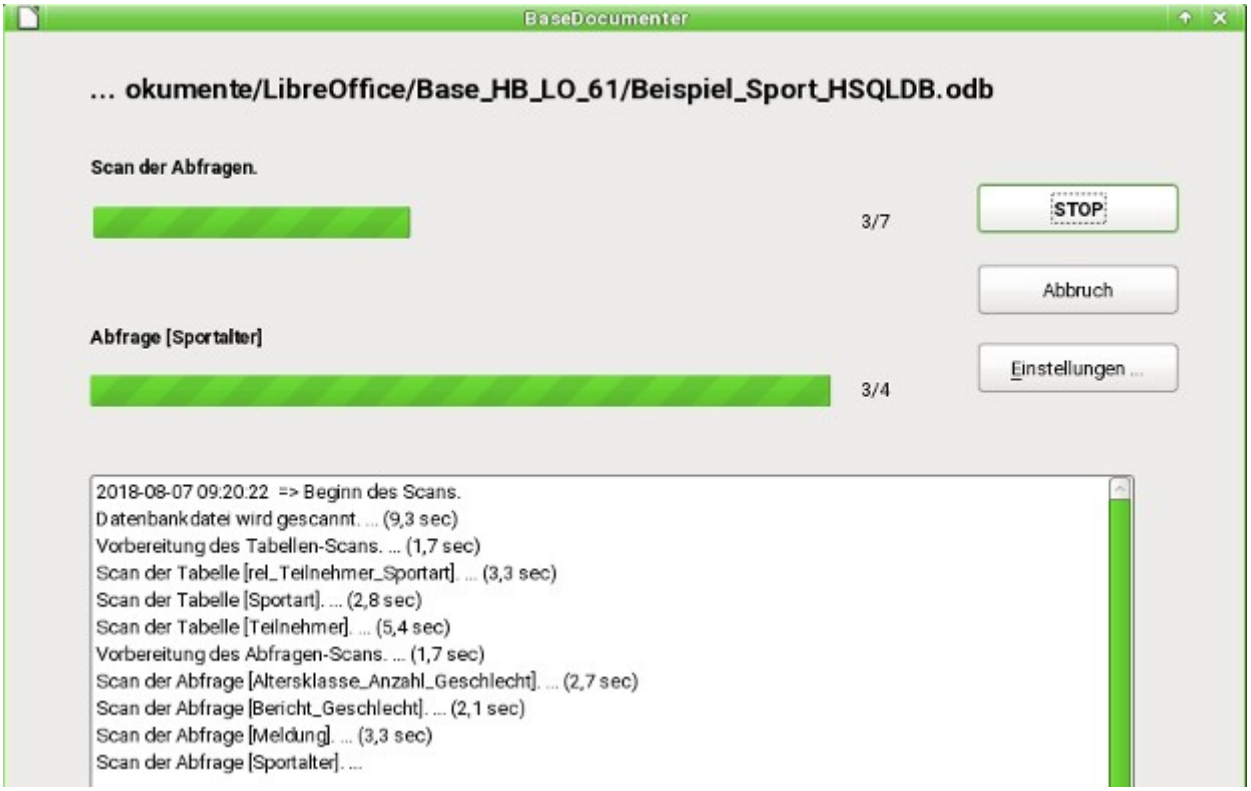
Ohne die Angabe eines Verzeichnisses für die Ausgabe-Seiten lässt sich ein Export nicht starten. Ohne die Vorlage-Dateien sind die erzeugten HTML-Dateien sehr unübersichtlich.



Bei den Formularen und Dialogen lässt sich die Einbindung von Screenshots einstellen. Die Screenshots selbst müssen allerdings durch den Nutzer angefertigt werden. Sie werden für Formulare in dem Unterordner **FORMS/«Formularname».png** abgelegt.

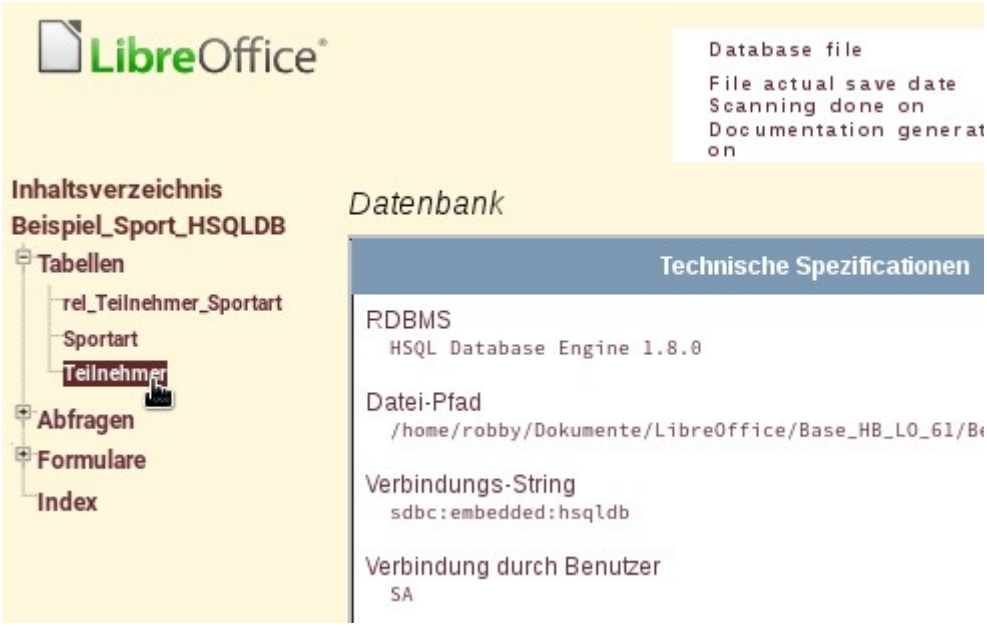
Wenn alle oben genannten Einstellungen erledigt sind kann die erste Datenbank dokumentiert werden. Der Aufruf erfolgt hier immer direkt in der geöffneten Datenbank, bei der für die Dokumentation die Ausführung von Makros unterbunden werden sollte.

Ein recht großer Dialog erscheint, von dem aus der Scan der Datei gestartet wird.



Nach dem Scan liegt in dem vorher erstellten Verzeichnis für die HTML-Dateien eine Datei «TOC.html». Diese Datei kann über den Browser aufgerufen werden.

Sind die Formatdateien installiert, so erscheint ein Inhaltsverzeichnis, das in dem obigen Bild erst einmal nur den Scan für die aktuelle Datei aufweist. Direkt über das Inhaltsverzeichnis sind hier u.a. die Tabelle (3), Abfragen (4) und Formulare(1) zu erreichen.



Wird der Name der Datenbank angeklickt, so erscheint zuerst eine Übersicht über die Datenbank mit technischen Spezifikationen usw. Daneben ist bei installierten Formatdateien ein Auswahlmenü zu sehen, mit dem z.B. durch die Tabellen navigiert werden kann.

Der BaseDocumenter zeigt sämtliche Beziehungen der Tabellen zu Abfragen und Formularen auf. Hier kann genau recherchiert werden, an welcher Stelle die Tabelle in der Datenbank benötigt wird. Das aktuelle Beispiel ist über *https://www.familiegrosskopf.de/robert/base_documenter/TOC.html* komplett einzusehen.

Codeschnipsel

Die Codeschnipsel erwachsen aus Anfragen innerhalb von Mailinglisten. Bestimmte Problemstellungen tauchen dort auf, deren Lösungen vielleicht gut innerhalb der eigenen Datenbankentwürfe genutzt werden können.

Aktuelles Alter ermitteln

Aus einem Datum soll mittels Abfrage das aktuelle Alter ermittelt werden. Siehe hierzu die Funktionen im Anhang zu diesem Base-Handbuch.

```
001 SELECT DATEDIFF('yy',"Geburtsdatum",CURDATE()) AS "Alter" FROM "Person"
```

Hinweis

Für Firebird muss der Code entsprechend angepasst werden. **CURDATE()** als Kurzform ist hier unbekannt. Es muss schon **CURRENT_DATE** sein. Und statt **'yy'** muss an dieser Stelle ein String ohne Hochkommata angegeben werden: **year**.
Auch die Kurzform **DAYOFYEAR** kennt Firebird nicht. Hier muss wieder **EXTRACT(YEARDAY FROM "Geburtsdatum")** usw. geschrieben werden.

Die Abfrage gibt das Alter als Jahresdifferenz aus. Das Alter eines Kindes, das am 31.12.2011 geboren ist, wird am 1.1.2012 mit 1 Jahr angegeben. Es muss also die Lage des Tages im Jahr berücksichtigt werden. Dies ist mit der Funktion **'DAYOFYEAR()'** ermittelbar. Mittels einer Funktion wird der Vergleich durchgeführt.

```
001 SELECT CASEWHEN
002 ( DAYOFYEAR("Geburtsdatum") > DAYOFYEAR(CURDATE()) ,
003 DATEDIFF ('yy',"Geburtsdatum",CURDATE())-1,
004 DATEDIFF ('yy',"Geburtsdatum",CURDATE()))
005 AS "Alter" FROM "Person"
```

Jetzt wird das aktuelle Alter in Jahren ausgegeben.

Über **'CASEWHEN'** könnte dann auch in einem weiteren Feld der Text **'Heute Geburtstag'** ausgegeben werden, wenn **DAYOFYEAR("Geburtsdatum") = DAYOFYEAR(CURDATE())**.

Spitzfindig könnte jetzt der Einwand kommen: «Wie steht es mit Schaltjahren?». Für Personen, die nach dem 28. Februar geboren wurden, kann es zu Abweichungen um einen Tag kommen. Für den Hausgebrauch nicht weiter schlimm, aber wo bleibt der Ehrgeiz, es möglichst doch genau zu machen?

Mit

```
001 SELECT CASEWHEN (
002 (MONTH("Geburtsdatum") > MONTH(CURDATE())) OR
003    ((MONTH("Geburtsdatum") = MONTH(CURDATE()))
004    AND (DAY("Geburtsdatum") > DAY(CURDATE()))) ,
005 DATEDIFF('yy',"Geburtsdatum",CURDATE())-1,
006 DATEDIFF('yy',"Geburtsdatum",CURDATE()))
007 AS "Alter" FROM "Person"
```

wird das Ziel erreicht. Solange der Monat des Geburtsdatums größer ist als der aktuelle Monat wird auf jeden Fall von der Jahresdifferenz 1 Jahr abgezogen. Ebenfalls 1 Jahr abgezogen wird, wenn zwar der Monat gleich ist, der Tag im Monat des Geburtsdatums aber größer ist als der Tag im aktuellen Monat. Leider ist diese Eingabe für die GUI nicht verständlich. Erst **'SQL-Kommando direkt ausführen'** lässt die Abfrage erfolgreich absetzen. So ist also unsere Abfrage nicht mehr editierbar. Die Abfrage soll aber weiter editierbar sein; also gilt es die GUI zu überlisten:

```
001 SELECT CASE
002 WHEN MONTH("Geburtsdatum") > MONTH(CURDATE())
003 THEN DATEDIFF('yy',"Geburtsdatum",CURDATE())-1
004 WHEN (MONTH("Geburtsdatum") = MONTH(CURDATE())
005    AND DAY("Geburtsdatum") > DAY(CURDATE()))
006 THEN DATEDIFF('yy',"Geburtsdatum",CURDATE())-1
007 ELSE DATEDIFF('yy',"Geburtsdatum",CURDATE())
008 END
009 AS "Alter" FROM "Person"
```

Auf diese Formulierung reagiert die GUI nicht mit einer Fehlermeldung. Das Alter wird jetzt auch in Schaltjahren genau ausgegeben und die Abfrage bleibt editierbar.

Geburtstage in den nächsten Tagen anzeigen

Mit Hilfe von einigen kleinen Berechnungskniffen lässt sich auch aus einer Tabelle z.B. auslesen, wer in den nächsten 8 Tagen Geburtstag hat.

```
001 SELECT *
002 FROM "Tabelle"
003 WHERE
004    DAYOFYEAR("Datum") BETWEEN DAYOFYEAR(CURDATE()) AND
005       DAYOFYEAR(CURDATE()) + 7
006    OR DAYOFYEAR("Datum") < 7 -
007       DAYOFYEAR(CAST(YEAR(CURDATE())||'-12-31' AS DATE)) +
008       DAYOFYEAR(CURDATE())
```

Die Abfrage zeigt alle Datensätze an, deren Datumseintrag zwischen dem momentanen Jahrestag und den folgenden 7 Tagen liegt.

Damit auch am Jahresende entsprechend 8 Tage angezeigt werden müssen die Anfangstage des Jahres grundsätzlich auch überprüft werden. Diese Überprüfung findet aber nur für die Jahrestage statt, die höchstens den Wert 7 abzüglich des Jahrestages des letzten Datumswertes des aktuellen Jahres (meist 365) zuzüglich des Jahrestages des aktuellen Datumswertes. Liegt das aktuelle Datum mehr als 7 Tage vom Jahresende entfernt, so ist der Gesamtwert also < 1. Kein Eintrag in der Tabelle hat so ein Datum. Diese Teilbedingung wird dann nicht erfüllt.

In der oben stehenden Formulierung werden bei Schaltjahren noch Unstimmigkeiten hervortreten, da sich dann ab dem 29.2. die Jahrestagzählung verschiebt. Der Code, um dieses zu berücksichtigen, gestaltet sich um einiges umfangreicher:

```
001 SELECT *
002 FROM "Tabelle"
003 WHERE
004       CASE
005          WHEN
006             DAYOFYEAR(CAST(YEAR("Datum")||'-12-31' AS DATE)) = 366
007             AND DAYOFYEAR("Datum") > 60 THEN DAYOFYEAR("Datum") - 1
008          ELSE
009             DAYOFYEAR("Datum")
010       END
011    BETWEEN
012       CASE
013          WHEN
014             DAYOFYEAR(CAST(YEAR(CURDATE())||'-12-31' AS DATE)) = 366
015             AND DAYOFYEAR(CURDATE()) > 60 THEN DAYOFYEAR(CURDATE()) - 1
016          ELSE
017             DAYOFYEAR(CURDATE())
018       END
019    AND
020       CASE
021          WHEN
022             DAYOFYEAR(CAST(YEAR(CURDATE())||'-12-31' AS DATE)) = 366
023             AND DAYOFYEAR(CURDATE()) > 60 THEN DAYOFYEAR(CURDATE()) + 6
024          ELSE
025             DAYOFYEAR(CURDATE()) + 7
026       END
027    OR DAYOFYEAR("Datum") < 7 -
028       DAYOFYEAR(CAST(YEAR(CURDATE())||'-12-31' AS DATE)) +
029       DAYOFYEAR(CURDATE())
```

Schaltjahre sind daran zu erkennen, dass das gesamte Jahr 366 und nicht 365 Tage hat. Dies wird für die entsprechenden Unterscheidungen genutzt.

Zum einen muss jeder Datumswert darauf überprüft werden, ob er in einem Schaltjahr liegt und außerdem für die korrekte Zählung der 60. Tag im Jahr ist (31 Tage im Januar und 29 Tage im Februar). Für diesen Fall werden alle darauffolgende DAYOFYEAR – Werte für das Datum um 1 herabgestuft. Dann trifft ein 1.3. eines Schaltjahres wieder exakt auf einen 1.3. eines normalen Jahres.

Zum anderen muss auch beim aktuellen Jahr (CURDATE()) überprüft werden, ob es sich um ein Schaltjahr handelt. Auch hier wird entsprechend die Tageszahl um 1 herabgestuft.

Auch eine Anzeige von Terminen für die nächsten 8 Tage ist so noch nicht einwandfrei möglich, da das Jahr in der Abfrage noch ausgeklammert ist. Dies wäre aber über eine einfache zusätzliche Bedingung **`YEAR("Datum") = YEAR(CURDATE())`** für das aktuelle bzw. **`YEAR("Datum") = YEAR(CURDATE()) + 1`** für das zukünftige Jahr zu erfüllen.

Tage zu Datumswerten addieren

Bei der Ausleihe von Medien möchte vielleicht der Entleiher genau wissen, an welchem Tag denn die Rückgabe des Mediums erfolgen soll. Leider bietet die interne HSQLDB dazu nicht die Funktion **`DATEADD()`** an, wie dies in vielen der externen Datenbanken und auch bei Firebird der Fall ist.

Hinweis

Für Firebird ist diese Vorgehensweise zum Addieren von Tagen zu Datumswerten überflüssig. Mit

```
001  SELECT
002     "Leih_Datum",
003     DATEADD(DAY, 14, "LeihDatum") AS "Rueckgabedatum"
004  FROM "Ausleihe"
```

werden zum Ausleihdatum 14 Tage addiert. Der Befehl funktioniert allerdings nur bei direkter Ausführung des SQL-Kommandos.

Für die interne HSQLDB lässt sich für einen begrenzten Zeitraum der folgende Umweg nutzen:

Zuerst wird eine Tabelle mit dem Datumsverlauf über den gewünschten Zeitraum erstellt. Hierzu wird einfach Calc geöffnet, in das Feld A1 die Bezeichnung "ID" und in das Feld B1 die Bezeichnung "Datum" geschrieben. In Feld A2 wird eine 1 eingetragen, in Feld B2 das gewünschte Startdatum, z.B. 1.1.15. A2 und B2 werden markiert und weiter nach unten gezogen, so dass daraus eine fortlaufende Nummer und ein fortlaufendes Datum entstehen.

Anschließend wird diese Tabelle mit den Spaltenüberschriften zusammen markiert und in Base eingefügt: **rechte Maustaste → Einfügen → Tabellenname → Datum**. Bei den Optionen wird **Definition und Daten** sowie **Erste Zeile als Spaltennamen verwenden** angeklickt. Alle Spalten werden übernommen. Anschließend ist nur noch darauf zu achten, dass bei den Typformatierungen das Feld "ID" dem Typ **`Integer [INTEGER]`** zugeordnet werden soll und dem Datumsfeld auch der Typ **`Datum [DATE]`** zugeordnet wird. Ein Primärschlüssel ist nicht erforderlich, da die Daten später nicht geändert werden sollen. Dadurch, dass eben der Primärschlüssel nicht definiert wird, wird die Tabelle gleichzeitig schreibgeschützt.

Tipp

Auch über Abfragetechniken ist die Erstellung so einer Übersicht möglich. Diese Übersicht kann, wenn eine Filtertabelle genutzt wird, im Startdatum und ihrem Umfang an Datumswerten sogar gesteuert werden:

```
001  SELECT DISTINCT CAST
002     ( "Y"."Nr" + (SELECT "Jahr" FROM "Filter" WHERE "ID" =
003        True) - 1 || '-' ||
004     CASEWHEN( "M"."Nr" < 10, '0' || "M"."Nr", '' || "M"."Nr" )
005     || '-' ||
006     CASEWHEN( "D"."Nr" < 10, '0' || "D"."Nr", '' || "D"."Nr" )
007     AS DATE ) AS "Datum"
008  FROM "NrBis31" AS "D", "NrBis31" AS "M", "NrBis31" AS "Y"
009  WHERE "Y"."Nr" <= (SELECT "Jahre" FROM "Filter" WHERE "ID" =
010     True)
011  AND "M"."Nr" <= 12 AND "D"."Nr" <= 31
```

Diese Ansicht greift auf eine Tabelle zu, die lediglich die Nummern von 1 bis 31 erfasst und schreibgeschützt ist. In einer weiteren Filtertabelle wird das Startjahr und der Umfang in Jahren festgelegt, den die Ansicht umfassen soll. Das Datum wird dadurch zusammengestellt, dass ein Datumsausdruck aus Jahr, Monat und Tag als Text erstellt und anschließend in ein Datum umgewandelt wird. Die HSQLDB akzeptiert alle Tage bis zum 31. eines Monats, auch z.B. den 31.02.2015. Aus dem 31.02.2015 wird allerdings entsprechend der 3.03.2015 wiedergegeben. Deshalb muss bei der Erstellung der Ansicht über **DISTINCT** ausgeschlossen werden, dass Datumswerte doppelt vorkommen. Hierauf greift die folgende Ansicht zu:

```
001  SELECT "a"."Datum",
002     (SELECT COUNT(*) FROM "Ansicht_Datum" WHERE "Datum" <=
003        "a"."Datum") AS "lfdNr"
004  FROM "Ansicht_Datum" AS "a"
```

Über eine *Zeilennummerierung* wird den Datumswerten eine Nummer hinzugefügt. Da aus Ansichten sowieso keine Daten gelöscht werden können, muss hier natürlich nicht extra ein Schreibschutz bemüht werden.

Mit einer Abfrage kann jetzt zu einem bestimmten Datum ermittelt werden, wie z.B. das Datum in 14 Tagen lauten wird:

```
001 SELECT "a"."Leih_Datum",
002    (SELECT "Datum" FROM "Datum" WHERE "ID" =
003       (SELECT "ID" FROM "Datum" WHERE "Datum" = "a"."Leih_Datum")+14)
004       AS "Rueckgabedatum"
005 FROM "Ausleihe" AS "a"
```

In der ersten Spalte wird das Ausleihdatum ermittelt. Auf diese Spalte greift eine korrelierende Unterabfrage zu, die wieder in zwei Abfragen geschachtelt ist. Es wird über **SELECT "ID" FROM "Datum"** der Wert des Feldes "ID" ermittelt, der dem Ausleihdatum entspricht. Zu diesem Wert wird 14 addiert. Dieser Wert wird dem Feld "ID" bei der äußeren Unterabfrage zugewiesen. Zu dieser neuen "ID" wird nachgesehen, welches Datum im Datumsfeld des Datensatzes steht.

Leider wird bei der Anzeige in der Abfrage der Datumstyp nicht automatisch erkannt, so dass hier entsprechend formatiert werden muss. In einem Formular ist die entsprechende Anzeige aber dauerhaft speicherbar, so dass auch bei jeder Abfrage entsprechend ein Datumswert ausgegeben wird.

Eine Direktvariante zur Ermittlung des Datumswertes ist sogar auf kürzerem Wege möglich. Hierbei wird der Startwert genutzt, ab dem die interne Tageszählung von Base beginnt:

```
001 SELECT "Leih_Datum",
002    DATEDIFF( 'dd', '1899-12-30', "Leih_Datum" ) + 14 AS "Rueckgabedatum"
003 FROM "Tabelle"
```

Der ermittelte Zahlenwert kann im Formular als Datum über ein formatiertes Feld dargestellt werden. Es ist allerdings nur mit großem Aufwand möglich, ihn einfach für weiteren Abfragecode in ein SQL-Datum zu übertragen.

Zeiten zu Zeitstempeln addieren

In MySQL gibt es die Funktion **TIMESTAMPADD**(). Eine vergleichbare Funktion existiert in der HSQLDB nicht. Aber auch hier kann über den internen Zahlenwert, den so ein Zeitstempel einnimmt, mittels eines formatierten Feldes im Formular auch die Addition oder Subtraktion einer Zeit dargestellt werden.

Hinweis

Für Firebird ist diese Vorgehensweise zum Addieren von Zeiten zu Zeitstempeln überflüssig. Mit

```
001  SELECT "Zeitstempel",
002     DATEADD( MINUTE, 14, "Zeitstempel" )
003  FROM "Tabelle"
```

werden zum Zeitstempel 14 Minuten addiert. Der Befehl funktioniert allerdings nur bei direkter Ausführung des SQL-Kommandos.

Im Gegensatz zur Addition von Tagen zu einem Datum tritt bei den Zeiten allerdings ein Problem auf, das anfangs vielleicht gar nicht auffällt:

```
001 SELECT "DatumZeit"
002    DATEDIFF( 'ss', '1899-12-30', "DatumZeit" ) / 86400.0000000000 + 36/24
003       AS "DatumZeit+36Stunden"
004 FROM "Tabelle"
```

Für die neue Zeitberechnung wird der Unterschied zur Startzeit '0' des Systems genommen. Das ist, wie bei der Datumsberechnung bereits angewandt, der Datumsstempel vom 30.12.1899.

Hinweis

Das Nulldatum am 30.12.1899 ist vermutlich deshalb entstanden, weil das Jahr 1900 im Gegensatz zur üblichen 4-Jahres-Rechnung kein Schaltjahr war. So ist der Tag '1' der internen Rechnung auf den 31.12.1899 vorverlegt worden und eben nicht der 1.1.1900.

Der Unterschied wird hier allerdings in Sekunden ausgedrückt. Die interne Zahl rechnet allerdings die Tage als Stellen vor dem Komma, die Stunden, Minuten und Sekunden als Stellen nach dem Komma. Da in einen Tag 60*60*24 Sekunden passen, muss die entsprechend ermittelte Sekundenzahl durch 86400 geteilt werden, um schließlich vor dem Komma mit den entsprechenden Tagen und hinter dem Komma mit den Bruchteilen rechnen zu können. Damit die interne Hsqldb überhaupt Nachkommastellen bei dieser Berechnung ausgibt, müssen diese auch in der Rechnung vorkommen. Statt durch 86400 zu teilen, wurde deshalb durch 86400,0000000000 geteilt. Dezimalstellen erscheinen in der Abfrage durch einen Punkt getrennt. Das Ergebnis hat letztlich also 10 Dezimalstellen hinter dem Komma.

Zu diesem Ergebnis wird die Stundenzahl als Bruchteil eines Tages hinzugezählt. Die berechnete Ziffer lässt sich bei entsprechender Formatierung in der Anfrage darstellen. Dort wird die Formatierung aber leider nicht gespeichert. Innerhalb eines Formulars mit formatierbarem Feld oder innerhalb eines Berichts kann sie aber dauerhaft mit entsprechender Formatierung übernommen werden.

Sollen Minuten oder Sekunden addiert werden, so ist hier auch immer darauf zu achten, dass die Angaben der Minuten und Sekunden als Tagesbruchteile angegeben werden.

Liegt das Datum in den Monaten November, Dezember, Januar usw., dann fällt bei der Rechnung erst einmal nichts auf. Die Darstellung ist stimmig, zum Zeitstempel von 20.01.2015 13:00:00 eine Zeit von 36 Stunden addiert ergibt den neuen (nur dargestellten) Stempel 22.01.2015 01:00:00. Anders verhält es sich bei 20.04.2015 13:00:00. Da wird anschließend der 22.04.2015 00:00:00 ausgegeben. Das liegt an der Sommerzeit, die der Berechnung hier in die Quere kommt. Die gerade bei der Zeitumstellung «verlorene» oder

«gewonnene» Stunde lässt sich bei einem Übergang nicht berücksichtigen. Innerhalb einer Zeitzone kann allerdings auf verschiedene Weise eine entsprechende Berechnung mit «korrektem» Ergebnis durchgeführt werden. Hier die dafür einfachere Variante:

```
001 SELECT "DatumZeit"
002    DATEDIFF( 'dd', '1899-12-30', "DatumZeit" ) +
003       HOUR( "DatumZeit" ) / 24.0000000000 +
004       MINUTE( "DatumZeit" ) / 1440.0000000000 +
005       SECOND( "DatumZeit" ) / 86400.0000000000 +
006       36/24
007          AS "DatumZeit+36Stunden"
008 FROM "Tabelle"
```

Statt die Stunden, Minuten und Sekunden aus dem Ursprungsdatum zu beziehen, werden sie hier im Verhältnis zum aktuellen Datum dargestellt. Am 20.05.2015 steht die Uhr auf 13:00 Uhr, würde aber ohne die Sommerzeit 12:00 Uhr anzeigen. Die Funktion **HOUR** berücksichtigt die Sommerzeit und gibt 13 Stunden als Stundenanteil aus. Damit kann dann der Stundenanteil korrekt zum Tagesanteil addiert werden. Auf gleiche Art und Weise geschieht dies mit Hilfe des Minutenanteils und des Sekundenanteils. Anschließend werden die zu addierenden Stunden wieder als Tagesbruchteil addiert und das Ganze mit Hilfe der Zellenformatierung als berechneter Zeitstempel ausgegeben.

Zwei Dinge sollten aber bei diesen Berechnungen nie aus den Augen verloren werden:

1. Bei Übergängen von Winterzeit zu Sommerzeit lassen sich die Stundenwerte nicht korrekt berechnen. Hier könnte mit Hilfe einer Zusatztabelle nachgeholfen werden, die Beginn und Ende der Sommerzeiten aufnimmt und dann mit einer Stundenkorrektur einspringt. Ein etwas aufwändiges Verfahren.
2. Die Ausgabe der Zeitangabe ist nur über formatierbare Felder zu erreichen. Das Ergebnis ist ein Dezimalwert, kein Zeitstempelwert, der z.B. wiederum in einer Datenbank direkt gespeichert werden könnte. Hier müsste entweder innerhalb des Formular kopiert werden oder über aufwändige Abfragen aus dem Dezimalwert zu einem Zeitstempelwert portiert werden. Der Knackpunkt bei dieser Portierung ist der Datumswert, da es eben Schaltjahre und Monate mit unterschiedlicher Tagesanzahl gibt.

Laufenden Kontostand nach Kategorien ermitteln

Statt eines Haushaltsbuches wird eine Datenbank im PC die leidigen Aufsummierungen von Ausgaben für Lebensmittel, Kleidung, Mobilität usw. erleichtern. Ein Großteil dieser Angaben sollte natürlich auf Anhieb in der Datenbank sichtbar sein. Dabei wird in dem Beispiel davon ausgegangen, dass Einnahmen und Ausgaben in einem Feld "Betrag" mit Vorzeichen abgespeichert werden. Prinzipiell lässt sich das Ganze natürlich auf getrennte Felder und eine Summierung hierüber erweitern.

```
001 SELECT "ID", "Betrag",
002    ( SELECT SUM( "Betrag" ) FROM "Kasse" WHERE "ID" <= "a"."ID" )
          AS "Saldo"
003 FROM "Kasse" AS "a" ORDER BY "ID" ASC
```

Mit dieser Abfrage wird bei jedem neuen Datensatz direkt ausgerechnet, welcher Kontostand jetzt erreicht wurde. Dabei bleibt die Abfrage editierbar, da das Feld "Saldo" durch eine korrelierende Unterabfrage erstellt wurde. Die Abfrage gibt den Kontostand in Abhängigkeit von dem automatisch erzeugten Primärschlüssel "ID" an. Kontostände werden aber eigentlich täglich ermittelt. Es muss also eine Datumsabfrage her.

```
001 SELECT "ID", "Datum", "Betrag",
002    ( SELECT SUM( "Betrag" ) FROM "Kasse" WHERE "Datum" <= "a"."Datum"
003       ) AS "Saldo"
004 FROM "Kasse" AS "a" ORDER BY "Datum", "ID" ASC
```

Die Ausgabe erfolgt jetzt nach dem Datum sortiert und nach dem Datum summiert. Bleibt noch die Kategorie, nach der entsprechende Salden für die einzelnen Kategorien dargestellt werden sollen.

```
001 SELECT "ID", "Datum", "Betrag", "Konto_ID",
002    ( SELECT "Konto" FROM "Konto" WHERE "ID" = "a"."Konto_ID" )
003       AS "Kontoname",
004    ( SELECT SUM( "Betrag" ) FROM "Kasse" WHERE "Datum" <= "a"."Datum"
005       AND "Konto_ID" = "a"."Konto_ID" ) AS "Saldo",
006    ( SELECT SUM( "Betrag" ) FROM "Kasse" WHERE "Datum" <= "a"."Datum"
007       ) AS "Saldo_gesamt"
008 FROM "Kasse" AS "a" ORDER BY "Datum", "ID" ASC
```

Hiermit wird eine editierbare Abfrage erzeugt, in der neben den Eingabefeldern (Datum, Betrag, Konto_ID) der Kontoname, der jeweilige Kontostand und der Saldo insgesamt erscheinen. Da sich die korrelierenden Unterabfragen teilweise auch auf vorhergehende Eingaben stützen ("Datum" <= "a"."Datum") werden nur Neueingaben reibungslos dargestellt. Änderungen eines vorhergehenden Datensatzes machen sich zuerst einmal nur in diesem Datensatz bemerkbar. Die Abfrage muss aktualisiert werden, damit auch die davon abhängigen späteren Berechnungen neu durchgeführt werden.

Zeilennummerierung

Automatisch hoch zählende Felder sind etwas feines. Nur sagen sie nicht unbedingt etwas darüber aus, wie viele Datensätze denn nun in der Datenbank oder dem Abfrageergebnis wirklich vorhanden sind. Häufig werden Datensätze gelöscht und manch ein User versucht verzweifelt dahinter zu kommen, welche Nummer denn nun nicht mehr vorhanden ist, damit die laufenden Nummern wieder stimmen.

```
001 SELECT "ID",
002    ( SELECT COUNT( "ID" ) FROM "Tabelle" WHERE "ID" <= "a"."ID" )
003       AS "lfdNr."
004 FROM "Tabelle" AS "a"
```

Das Feld "ID" wird ausgelesen, im zweiten Feld wird durch eine korrelierende Unterabfrage festgestellt, wie viele Feldinhalte von "ID" kleiner oder gleich dem aktuellen Feldinhalt sind. Daraus wird dann die laufende Zeilennummer erstellt.

Hinweis

Mit Firebird ist es möglich, hier eine eingebaute Funktion zu nutzen.

```
001 SELECT "ID",
002    ROW_NUMBER() OVER (ORDER BY "ID") AS "lfdNr."
003 FROM "Tabelle" AS "a"
```

Wird einer Abfrage eine Bedingung hinzugefügt oder beruht eine Abfrage aus einer Verbindung von mehreren Tabellen, so ist diese Bedingung bzw. die Verbindung von mehreren Tabellen auch in der korrelierenden Unterabfrage hinzuzufügen. Andernfalls stimmt die Zählung nicht:

```
001 SELECT "Name",
002    ( SELECT COUNT( "ID" ) FROM "Name" WHERE "GebDat" > '1995-12-31'
          AND "ID" <= "a"."ID" ) AS "lfdNr."
003 FROM "Name" AS "a" WHERE "GebDat" > '1995-12-31'
```

Die gesamte Bedingung der äußeren Abfrage muss in der korrelierenden Unterabfrage wiederholt werden. Hinzu kommt in der korrelierenden Unterabfrage noch die Bedingung, mit der sich die korrelierende Unterabfrage auf den aktuellen Datensatz bezieht.

Auch eine Nummerierung für entsprechende Gruppierungen ist möglich:

```
001 SELECT "ID",
002    ( SELECT COUNT( "ID" ) FROM "Tabelle" WHERE "ID" <= "a"."ID"
```

```
          AND "RechnungsID" = "a"."RechnungsID" ) AS "lfdNr."
003 FROM "Tabelle" AS "a"
```

Hier gibt es in einer Tabelle verschiedene Rechnungsnummern ("RechnungsID"). Für jede Rechnungsnummer wird separat die Anzahl der Felder "ID" aufsteigend nach der Sortierung des Feldes "ID" wiedergegeben. Das erzeugt für jede Rechnung die Nummerierung von 1 an aufwärts.

Soll die aktuelle Sortierung der Abfrage mit der Zeilennummer übereinstimmen, so ist die Art der Sortierung entsprechend abzubilden. Dabei muss die Sortierung allerdings einen eindeutigen Wert ergeben. Sonst liegen 2 Nummern auf dem gleichen Wert.

```
001 SELECT "Name",
002    ( SELECT COUNT( "ID" ) FROM "Tabelle" WHERE "Name" <= "a"."Name" )
003       AS "lfdNr."
004 FROM "Tabelle" AS "a"
005 ORDER BY "Name"
```

Nur wenn ein eindeutigen Index für das Feld "Name" definiert wurde, ist hier eine eindeutige Sortierung und auch eine klare Nummerierung zu erwarten. Tauchen aber zwei gleiche Namen auf, so erhalten beide die gleiche Nummer.

```
001 SELECT "Name",
002    ( SELECT COUNT( "ID" ) FROM "Tabelle"
003       WHERE "Name"||"ID" <= "a"."Name"||"a"."ID" ) AS "lfdNr."
004 FROM "Tabelle" AS "a"
005 ORDER BY "Name"||"ID"
```

Hier wird der Inhalt des Feldes "Name" mit dem Inhalt des Feldes "ID" zusammengefügt. Die Sortierung ist jetzt eindeutig, die laufende Nummer entsprechend auch. Bei genauerer Betrachtung fällt allerdings auf, dass die Sortierung der "ID" natürlich jetzt nicht der Zahlensortierung folgt. Stattdessen wird z.B. die Zahl '9' nach der Zahl '10' einsortiert, da 9 größer als 1 ist. Wer dem noch abhelfen will muss entsprechend mit Leerstellen oder Nullen auffüllen:

```
001 SELECT "Name",
002    ( SELECT COUNT( "ID" ) FROM "Tabelle"
003       WHERE "Name"||RIGHT('000000000'||"ID",10) <= "a"."Name"||
004       RIGHT('000000000'||"a"."ID",10) ) AS "lfdNr."
005 FROM "Tabelle" AS "a"
006 ORDER BY "Name"||RIGHT('000000000'||"ID",10)
```

Das Feld "ID" ist in diesem Beispiel ein INTEGER-Feld. INTEGER-Zahlen haben maximal 10 Stellen. Der Wert des Feldes wird mit 9 führenden Nullen aufgefüllt. Anschließend werden die 10 rechts stehenden Zeichen für die weitere Verarbeitung genutzt. '0000000009' ist nun kleiner als '0000000010'. Die Sortierung ist eindeutig und erfolgt bei gleichem Namen schließlich in gewohnter Reihenfolge der Schlüsselnummerierung.

Sollen Zeilen für eine Abfrage nummeriert werden, die sich nicht nur auf eine Tabelle bezieht oder mit einer zusätzlichen Bedingung versehen ist, so muss in der korrelierenden Unterabfrage die komplette Bedingung mit allen betroffenen Tabellen enthalten sein. Nur dann kann die Unterabfrage die gleichen Daten verarbeiten wie die äußere Abfrage.

Gleiche Werte können natürlich genutzt werden, wenn z.B. die Platzierung bei einem Wettkampf wiedergegeben werden soll, da hier gleiche Wettkampfergebnisse auch zu einer gleichen Platzierung führen. Damit die Platzierung allerdings so wiedergegeben wird, dass bei einer gleichen Platzierung der nachfolgende Wert ausgelassen wird, ist die Abfrage etwas anders zu konstruieren:

```
001 SELECT "ID",
002    ( SELECT COUNT( "ID" ) + 1 FROM "Tabelle" WHERE "Zeit" < "a"."Zeit" )
003       AS "Platz"
004 FROM "Tabelle" AS "a"
```

Es werden alle Einträge ausgewertet, die in dem Feld "Zeit" einen kleineren Eintrag haben. Damit werden alle Sportler erfasst, die vor dem aktuellen Sportler ins Ziel gekommen sind. Zu diesem Wert wird der Wert 1 addiert. Damit ist der Platz des aktuellen Sportlers bestimmt. Ist

dieser zeitgleich mit einem anderen Sportler, so ist auch der Platz gleich. Damit sind Platzierungen wie 1. Platz, 2. Platz, 2. Platz, 4. Platz usw. möglich.

Schwieriger wird es, wenn neben der Platzierung auch eine Zeilennummerierung erfolgen soll. Dies kann z.B. sinnvoll sein, um mehrere Datensätze in einer Zeile zusammen zu fassen.

```
001 SELECT "ID",
002    ( SELECT COUNT( "ID" ) + 1 FROM "Tabelle" WHERE "Zeit" < "a"."Zeit" )
003       AS "Platz",
004    CASE WHEN
005       ( SELECT COUNT( "ID" ) + 1 FROM "Tabelle"
006       WHERE "Zeit" = "a"."Zeit" ) = 1
007    THEN
008       ( SELECT COUNT( "ID" ) + 1 FROM "Tabelle"
009       WHERE "Zeit" < "a"."Zeit" )
010    ELSE (SELECT
011       (SELECT COUNT( "ID" ) + 1 FROM "Tabelle" WHERE "Zeit" < "a"."Zeit")
012       + COUNT( "ID" ) FROM "Tabelle"
013       WHERE "Zeit" = "a"."Zeit" "ID" < "a"."ID" )
       END AS "Zeilennummer"
014 FROM "Tabelle" AS "a"
```

Die zweite Spalte gibt weiterhin die Platzierung wieder. In der 3. Spalte wird zuerst nachgefragt, ob auch wirklich nur eine Person mit der gleichen Zeit durchs Ziel gekommen ist. Wenn dies erfüllt ist, wird die Platzierung auf jeden Fall direkt als Zeilennummer übernommen. Wenn dies nicht erfüllt ist, wird zu der Platzierung ein weiterer Wert addiert. Bei gleicher Zeit (**`"Zeit" = "a"."Zeit"`**) wird dann mindestens 1 addiert, wenn es eine weitere Person mit dem Primärschlüssel ID gibt, deren Primärschlüssel kleiner ist als der aktuelle Primärschlüssel des aktuellen Datensatzes (**`"ID" < "a"."ID"`**). Diese Abfrage gibt also solange identische Werte zur Platzierung heraus, wie keine zweite Person mit der gleichen Zeit existiert. Existiert eine zweite Person mit der gleichen Zeit, so wird nach der ID entschieden, welche Person die geringere Zeilennummer enthält.

Diese Zeilensortierung entspricht übrigens der, die die Datenbanken anwenden. Wird z.B. eine Reihe Datensätze nach dem Namen sortiert, so erfolgt die Sortierung bei gleichen Datensätzen nicht nach dem Zufallsprinzip, sondern aufsteigend nach dem Primärschlüssel, der ja schließlich eindeutig ist. Es lässt sich auf diese Weise also über die Nummerierung eine Sortierung der Datensätze abbilden.

Die Zeilennummerierung ist auch eine gute Voraussetzung, um einzelne Datensätze als einen Datensatz zusammen zu fassen. Wird eine Abfrage zur Zeilennummerierung als Ansicht erstellt, so kann darauf mit einer weiteren Abfrage problemlos zugegriffen werden. Als einfaches Beispiel hier noch einmal die erste Abfrage zur Nummerierung, nur um ein Feld ergänzt:

```
001 SELECT "ID", "Name",
002    ( SELECT COUNT( "ID" ) FROM "Tabelle" WHERE "ID"   <= "a"."ID" )
003       AS "lfdNr."
004 FROM "Tabelle" AS "a"
```

Aus dieser Abfrage wird jetzt die Ansicht '*Ansicht1*' erstellt. Die Abfrage, mit der z.B. die ersten 3 Namen zusammen in einer Zeile erscheinen können, lautet:

```
001 SELECT "Name" AS "Name_1",
002    ( SELECT "Name" FROM "Ansicht1" WHERE "lfdNr." = 2 ) AS "Name_2",
003    ( SELECT "Name" FROM "Ansicht1" WHERE "lfdNr." = 3 ) AS "Name_3"
004 FROM "Ansicht1" WHERE "lfdNr." = 1
```

Auf diese Art und Weise können mehrere Datensätze nebeneinander als Felder dargestellt werden. Allerdings läuft diese Nummerierung einfach vom ersten bis zum letzten Datensatz durch.

Sollen alle Personen zu einem Nachnamen zugeordnet werden, so ließe sich das folgendermaßen realisieren:

```
001 SELECT "ID", "Name", "Nachname",
```

```
002    ( SELECT COUNT( "ID" ) FROM "Tabelle" WHERE "ID" <= "a"."ID"
003       AND "Nachname" = "a"."Nachname") AS "lfdNr."
004 FROM "Tabelle" AS "a"
```

Jetzt kann über die erstellte Ansicht eine entsprechende Familienzusammenstellung erfolgen:

```
001 SELECT "Nachname", "Name" AS "Name_1",
002    ( SELECT "Name" FROM "Ansicht1" WHERE "lfdNr." = 2
003       AND "Nachname" = "a"."Nachname") AS "Name_2",
004    ( SELECT "Name" FROM "Ansicht1" WHERE "lfdNr." = 3
005       AND "Nachname" = "a"."Nachname") AS "Name_3"
006 FROM "Ansicht1" AS "a"
007 WHERE "lfdNr." = 1
```

In einem Adressbuch ließen sich so alle Personen einer Familie ("Nachnamen") zusammenfassen, damit jede Adresse nur einmal für ein Anschreiben berücksichtigt würde, aber alle Personen, an die das Anschreiben gehen soll, aufgeführt würden.

Da es sich um keine fortwährende Schleifenfunktion handelt, ist hier allerdings Vorsicht geboten. Schließlich wird die Grenze der parallel als Felder angezeigten Datensätze durch die Abfrage im obigen Beispiel z.B. auf 3 begrenzt. Diese Grenze wurde willkürlich gesetzt. Weitere Namen tauchen nicht auf, auch wenn die Nummerierung der "*lfdNr.*" größer als 3 ist.

In seltenen Fällen ist so eine Grenze aber auch klar nachvollziehbar. Soll z.B. ein Kalender erstellt werden, so können mit diesem Verfahren die Zeilen die Wochen des Jahres darstellen, die Spalten die Wochentage. Da im ursprünglichen Kalender nur das Datum über den Inhalt entscheidet, werden durch die Zeilennummerierung immer die Tage einer Woche durchnummeriert und nach Wochen im Jahr als Datensatz später ausgegeben. Spalte 1 gibt dann Montag wieder, Spalte 2 Dienstag usw. Die Unterabfrage endet also jeweils bei der "*lfdNr.*" = *7*. Damit lassen sich dann im Bericht alle sieben Wochentage nebeneinander anzeigen und eine entsprechende Kalenderübersicht erstellen.

Zeilenumbruch durch eine Abfrage erreichen

Manchmal ist es sinnvoll, durch eine Abfrage verschiedene Felder zusammenzufassen und mit einem Zeilenumbruch zu trennen. So ist es z.B. einfacher eine Adresse in einen Bericht komplett einzulesen.

Der Zeilenumbruch innerhalb einer Abfrage erfolgt durch **`Char(13)`**. Beispiel:

```
001 SELECT "Vorname"||' '||"Nachname"||Char(13)||"Straße"||Char(13)||"Ort"
002 FROM "Tabelle"
```

Dies erzeugt nachher:

```
Vorname Nachname
Straße
Ort
```

Hinweis

Für Firebird muss statt **`CHAR(13)`** **`ASCII_CHAR(13)`** eingefügt werden. Firebird kennt die hier verwendete Kurzform nicht.

Mit so einer Abfrage, zusammen mit einer Nummerierung jeweils bis zur Nummer 3, lassen sich auch dreispaltige Etikettendrucke von Adressetiketten über Berichte realisieren. Eine Nummerierung ist in diesem Zusammenhang nötig, damit drei Adressen nebeneinander in einem Datensatz erscheinen. Nur so sind sie auch nebeneinander im Bericht einlesbar.

Je nach Betriebssystem kann es auch notwendig sein, zusätzlich zu Char(13) auch Char(10) in den Code mit aufzunehmen:

```
001 SELECT "Vorname"||' '||"Nachname"||Char(13)||Char(10)||
002    "Straße"||Char(13)||Char(10)||"Ort"
```

```
003 FROM "Tabelle"
```

Solche Zeilenumbrüche werden allerdings nicht in der Abfrage angezeigt. Die Steuerzeichen werden in einem VARCHAR-Feld nicht umgesetzt. Entsprechend rückt der Text dort ohne Leerzeichen aneinander. Soll der Zeilenumbruch in der Abfrage angezeigt werden, so muss der zusammengefügte Text von VARCHAR nach LONGVARCHAR umgewandelt werden:

```
001 SELECT CAST("Vorname"||' '||"Nachname"||Char(13)||Char(10)||
002    "Straße"||Char(13)||Char(10)||"Ort" AS LONGVARCHAR) AS "Adresse"
003 FROM "Tabelle"
```

Damit wird dann die Adresse auch in einer Abfrage mehrzeilig unter dem Alias "Adresse" dargestellt.

Gruppieren und Zusammenfassen

Für andere Datenbanken, auch neuere Versionen der HSQLDB, ist der Befehl **'Group_Concat()'** verfügbar. Mit ihm können einzelne Felder einer Datensatzgruppe zusammengefasst werden. So ist es z.B. möglich, in einer Tabelle Vornamen und Nachnamen zu speichern und anschließend die Daten so darzustellen, dass in einem Feld die Nachnamen als Familiennamen erscheinen und in dem 2. Feld alle Vornamen hintereinander, durch z.B. Komma getrennt, aufgeführt werden.

Hinweis

Für Firebird ist diese Vorgehensweise zum Gruppieren überflüssig. Mit

```
001 SELECT "Nachname", LIST( "Vorname", ', ' ) AS "Vornamen"
002 FROM "Tabelle" GROUP BY "Nachname"
```

werden die Vornamen, gruppiert nach den Nachnamen, zusammengefasst.

Dieses Beispiel entspricht in vielen Teilen dem der Zeilennummerierung. Die Gruppierung zu einem gemeinsamen Feld stellt hier eine Ergänzung dar.

Nachname	***Vorname***
Müller	Karin
Schneider	Gerd
Müller	Egon
Schneider	Volker
Müller	Monika
Müller	Rita

Wird nach der Abfrage zu:

Nachname	***Vornamen***
Müller	Karin, Egon, Monika, Rita
Schneider	Gerd, Volker

Dieses Verfahren kann in Grenzen auch in der HSQLDB nachgestellt werden. Das folgende Beispiel bezieht sich auf eine Tabelle "Name" mit den Feldern "ID", "Vorname" und "Nachname". Folgende Abfrage wird zuerst an die Tabelle gestellt und als Ansicht "Ansicht_Gruppe" gespeichert:

```
001 SELECT "Nachname", "Vorname",
002    ( SELECT COUNT( "ID" ) FROM "Name" WHERE "ID" <= "a"."ID"
003       AND "Nachname" = "a"."Nachname" ) AS "GruppenNr"
004 FROM "Name" AS "a"
```

Im Kapitel «Abfragen» ist nachzulesen, wie diese Abfrage über die «Korrelierte Unterabfrage» auf Feldinhalte in der gleichen Abfragezeile zugreift. Dadurch wird eine aufsteigende Nummerierung, gruppiert nach den "*Nachnamen*", erzeugt. Diese Nummerierung wird in der folgenden Abfrage benötigt, so dass in dem Beispiel maximal 5 Vornamen aufgeführt werden.

```
001 SELECT DISTINCT "Nachname",
002    ( SELECT "Vorname" FROM "Ansicht_Gruppe"
003       WHERE "Nachname" = "a"."Nachname" AND "GruppenNr" = 1 ) ||
004       COALESCE( ( SELECT ', ' || "Vorname" FROM "Ansicht_Gruppe"
005       WHERE "Nachname" = "a"."Nachname" AND "GruppenNr" = 2 ), '' ) ||
006       COALESCE( ( SELECT ', ' || "Vorname" FROM "Ansicht_Gruppe"
007       WHERE "Nachname" = "a"."Nachname" AND "GruppenNr" = 3 ), '' ) ||
008       COALESCE( ( SELECT ', ' || "Vorname" FROM "Ansicht_Gruppe"
009       WHERE "Nachname" = "a"."Nachname" AND "GruppenNr" = 4 ), '' ) ||
010       COALESCE( ( SELECT ', ' || "Vorname" FROM "Ansicht_Gruppe"
011       WHERE "Nachname" = "a"."Nachname" AND "GruppenNr" = 5 ), '' )
012 AS "Vornamen"
013 FROM "Ansicht_Gruppe" AS "a"
```

Durch Unterabfragen werden nacheinander die Vornamen zu Gruppenmitglied 1, 2 usw. abgefragt und zusammengefasst. Ab der 2. Unterabfrage muss abgesichert werden, dass **'NULL'**-Werte nicht die Zusammenfassung auf **'NULL'** setzen. Deshalb wird bei einem Ergebnis von **'NULL'** stattdessen **''** angezeigt.

Mehrere Werte in einem Feld speichern

Sollen mit dem Datensatz einer Tabelle mehrere Werte der gleichen Art verbunden werden, so wird dies normalerweise durch eine n:m-Beziehung über 3 Tabellen gelöst. Bei einer bereits bestehenden Datenbank und einem nicht zu großen Umfang der Werte können aber auch mehrere Werte in einem Feld abgespeichert und wieder abgerufen werden. Dies soll hier an einem Terminkalender dargestellt werden.[8]

Neben den üblichen Feldern für einen Terminkalender erscheint in der Tabelle "Termine" ein Feld "Personen". Aus diesem Feld "Personen" soll erkennbar sein, welche Personen von einem Termin betroffen sind. Jeder Person wird dabei ein bestimmter Wert zugeordnet, der gleichzeitig einem bestimmten Bit-Wert entspricht.

Person	***Wert (Integer)***	***Wert (Bit)***
Müller	1	0000 0001
Schneider	2	0000 0010
Meier	4	0000 0100
Schulze	8	0000 1000
Achenbach	16	0001 0000
Sorge	32	0010 0000

Wird in dem Feld "Personen" '1' abgespeichert, so betrifft der Termin nur die Person '1' - 'Müller'. Wird '3' eingetragen, so sind die Personen '1' - 'Müller' und '2' - 'Schneider' von dem Termin betroffen. In das Feld "Personen" wird also immer die Summe der Integer-Werte eingetragen. Über die folgende Abfrage kann dann ermittelt werden, welche von den Personen betroffen ist:

```
001 SELECT "Termine".*,
002    BITAND( "Personen", 1 ) "Müller",
003    BITAND( "Personen", 2 ) "Schneider",
004    BITAND( "Personen", 4 ) "Meier",
005    BITAND( "Personen", 8 ) "Schulze",
```

8 Die Datenbank «Beispiel_Arrayfeld.odb» ist den Beispieldatenbanken für dieses Handbuch beigefügt.

```
006    BITAND( "Personen", 16 ) "Achenbach",
007    BITAND( "Personen", 32 ) "Sorge"
008 FROM "Termine"
```

Während bei der HSQLDB die Funktion den Namen **BITAND** trägt, wird die gleiche Berechnung unter FIREBIRD mit **BIN_AND** durchgeführt. Die Abfrage ergibt die jeweiligen Personenwerte oder stattdessen 0, wenn der entsprechende Wert nicht in der abgespeicherten Integer-Zahl enthalten ist.

In einem Formular lassen sich diese unterschiedlichen Werte am besten mit Markierfeldern auslesen. Den Markierfeldern wird dabei unter **Eigenschaften → Allgemein → Titel** der jeweilige Name der Person gegeben, unter **Eigenschaften → Daten → Referenzwert (ein)** der Integer-Wert zugeschrieben. Bei **Referenzwert (aus)** kann '0' stehen. In dem Formular sind dann die jeweiligen Felder markiert, wenn das Formular mit der Abfrage und die Felder mit dem entsprechenden Namen verbunden sind.

Das Ändern der Werte soll jetzt natürlich auch über das Formular möglich sein. Dafür muss ein bisschen mit einem Makro nachgeholfen werden:

```
001 SUB PersonenSpeichern(oEvent AS OBJECT)
002    DIM aFields()
003    DIM oForm AS OBJECT
004    DIM inValue AS INTEGER
005    DIM i AS INTEGER
006    oForm = oEvent.Source.Model.Parent
007    aFields = Array("Check1","Check2","Check3","Check4","Check5","Check6")
008    inValue = 0
009    FOR i = LBound(aFields()) TO UBound(aFields())
010       IF oForm.getByName(aFields(i)).State = 1 THEN
011          inValue = inValue + CInt(oForm.getByName(aFields(i)).refValue)
012       END IF
013    NEXT
014    oForm.UpdateInt(oForm.findColumn("Personen"),inValue)
015 END SUB
```

Das Formular wird aus dem auslösenden Ereignis (**Eigenschaften → Ereignisse → Status geändert**) ausgelesen. Sämtliche Markierfelder, die zusammen mit ihrem Wert in dem Feld "Personen" vertreten sein sollen, sind in dem Array aufgelistet. Aus jedem Feld, bei dem der Status auf «ausgewählt» steht (**State = 1**) wird der Referenzwert ausgelesen. Dieser Referenzwert ist als Text gespeichert, so dass er für das Makro mit **Cint** in einen Integer-Wert umgewandelt werden muss. Alle Werte werden addiert und dann in das Feld "Personen" der dem Formular zugrundeliegenden Abfrage übertragen.

Damit lassen sich beliebige Personenzusammenstellungen in einem Feld speichern und auch wieder auslesen. Über Abfragen lassen sich alle Termine zusammenstellen, die eine Person hat, ohne dass dabei auch noch die anderen Termine stören, von denen die Person gar nicht betroffen ist.