LibreOffice
The Document Foundation

Base

Kapitel 8
Datenbankaufgaben

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 1.3.2013. Basierend auf der LibreOffice Version 4.0.

Anmerkung für Macintosh Nutzer

Einige Tastenbelegungen (Tastenkürzel) und Menüeinträge unterscheiden sich zwischen der Macintosh Version und denen für Windows- und Linux-Rechnern. Die unten stehende Tabelle gibt Ihnen einige grundlegende Hinweise dazu. Eine ausführlichere Aufstellung dazu finden Sie in der Hilfedatei des jeweiligen Moduls.

Windows/Linux	***entspricht am Mac***	***Effekt***
Menü-Auswahl **Extras → Optionen**	**LibreOffice → Einstellungen**	Zugriff auf die Programmoptionen
Rechts-Klick	*Control*+Klick	Öffnen eines Kontextmenüs
Ctrl (Control) oder *Strg* (Steuerung)	⌘ *(Command)*	Tastenkürzel in Verbindung mit anderen Tasten
F5	*Shift*+⌘+*F5*	öffnet den Dokumentnavigator Dialog
F11	⌘+*T*	öffnet den Formatvorlagen Dialog

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 dem kleinsten Zahlentyp, **'TINYINT'**, entsprechen. Die Tabelle **"Filter"** wird sowieso nur einen Datensatz abspeichern.

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:

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

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

```
SELECT * FROM "Medien" WHERE "Titel" = IFNULL( ( SELECT "Filter_1"
FROM "Filter" ), "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.

```
SELECT * , IFNULL( "Titel", '' ) AS "T" FROM "Medien" WHERE "T" =
IFNULL( ( SELECT "Filter_1" FROM "Filter" ), "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:

```
SELECT "Medien".* , IFNULL( "Medien"."Titel", '' ) AS "T" FROM
"Medien" WHERE "T" = IFNULL( ( SELECT "Filter_1" FROM "Filter" ),
"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:

```
SELECT "Medien".* , IFNULL( "Medien"."Titel", '' ) AS "T",
IFNULL( "Medien"."Kategorie", '' ) AS "K" FROM "Medien" WHERE "T" =
IFNULL( ( SELECT "Filter_1" FROM "Filter" ), "T" ) AND "K" = IFNULL( (
SELECT "Filter_2" FROM "Filter" ), "K" )
```

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

```
SELECT DISTINCT "Titel", "Titel" 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.

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 Femdschlüsselfeld, 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 Lis-

tenfeld 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. Zuerst werden die ähnlichen Vorgehensweisen in Tabelle und Formular beschrieben.

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 Labelfeld 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.

```
SELECT * FROM "Medien" WHERE "Titel" = ( SELECT "Suchbegriff" FROM
"Filter" )
```

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:

```
SELECT * FROM "Medien" WHERE "Titel" = IFNULL( ( SELECT "Suchbegriff"
FROM "Filter" ), "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:

```
SELECT * FROM "Medien" WHERE "Titel" LIKE ( SELECT '%' || "Suchbe-
griff" ||'%' FROM "Filter" )
```

oder besser:

```
SELECT * FROM "Medien" WHERE "Titel" LIKE IFNULL( ( SELECT '%' ||
"Suchbegriff" ||'%' FROM "Filter" ), "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 an Möglichkeiten ab:

```
SELECT * FROM "Medien" WHERE
LOWER("Titel") LIKE IFNULL( ( SELECT '%' || LOWER("Suchbegriff") ||'%'
FROM "Filter" ), LOWER("Titel") )
```

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

```
SELECT * FROM "Medien" WHERE
LOWER("Titel") LIKE IFNULL( ( SELECT '%' || LOWER("Suchbegriff") ||'%'
FROM "Filter" ), LOWER("Titel") ) OR
LOWER("Kategorie") LIKE ( SELECT '%' || LOWER("Suchbegriff") ||'%'
FROM "Filter" )
```

Die **'IFNULL'**-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.

Codeschnipsel

Die Codeschnipsel erwachsen aus Anfragen innerhalb von Mailinglisten. Bestimmte Problemstellungen tauchen dort auf, die vielleicht gut als Lösungen 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.

```
SELECT DATEDIFF('yy',"Geburtsdatum",CURDATE()) AS "Alter" FROM "Per-
son"
```

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.

```
SELECT CASEWHEN
( DAYOFYEAR("Geburtsdatum") > DAYOFYEAR(CURDATE()) ,
DATEDIFF ('yy',"Geburtsdatum",CURDATE())-1,
DATEDIFF ('yy',"Geburtsdatum",CURDATE()))
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

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

wird das Ziel erreicht. Solange der Monat des Geburtsdatum 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:

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

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

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.

```
SELECT "ID", "Betrag", ( SELECT SUM( "Betrag" ) FROM "Kasse" WHERE
"ID" <= "a"."ID" ) AS "Saldo" 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 automa-

tisch erzeugten Primärschlüssel "ID" an. Kontostände werden aber eigentlich täglich ermittelt. Es muss also eine Datumsabfrage her.

```
SELECT "ID", "Datum", "Betrag", ( SELECT SUM( "Betrag" ) FROM "Kasse"
WHERE "Datum" <= "a"."Datum" ) AS "Saldo" 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.

```
SELECT "ID", "Datum", "Betrag", "Konto_ID",
( SELECT "Konto" FROM "Konto" WHERE "ID" = "a"."Konto_ID" ) AS "Konto-
name",
( SELECT SUM( "Betrag" ) FROM "Kasse" WHERE "Datum" <= "a"."Datum" AND
"Konto_ID" = "a"."Konto_ID" ) AS "Saldo",
( SELECT SUM( "Betrag" ) FROM "Kasse" WHERE "Datum" <= "a"."Datum" )
AS "Saldo_gesamt"
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 erscheint. 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 hochzählende 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.

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

Auch eine Nummerierung für entsprechende Gruppierungen ist möglich:

```
SELECT "ID", ( SELECT COUNT( "ID" ) FROM "Tabelle" WHERE "ID" <=
"a"."ID" AND "RechnungsID" = "a"."RechnungsID" ) AS "lfdNr." 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. Dies kann dann 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:

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

```
SELECT "ID", ( SELECT COUNT( "ID" ) + 1 FROM "Tabelle" WHERE "Zeit" <
"a"."Zeit" ) AS "Platz",
CASE WHEN
( SELECT COUNT( "ID" ) + 1 FROM "Tabelle" WHERE "Zeit" = "a"."Zeit" )
= 1
THEN ( SELECT COUNT( "ID" ) + 1 FROM "Tabelle" WHERE "Zeit" <
"a"."Zeit" )
ELSE (SELECT ( SELECT COUNT( "ID" ) + 1 FROM "Tabelle" WHERE "Zeit" <
"a"."Zeit" ) + COUNT( "ID" ) FROM "Tabelle" WHERE "Zeit" = "a"."Zeit"
"ID" < "a"."ID"
END
AS "Zeilennummer" 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:

```
SELECT "ID", "Name", ( SELECT COUNT( "ID" ) FROM "Tabelle" WHERE "ID"
<= "a"."ID" ) AS "lfdNr." 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:

```
SELECT "Name" AS "Name_1", ( SELECT "Name" FROM "Ansicht1" WHERE
"lfdNr." = 2 ) AS "Name_2", ( SELECT "Name" FROM "Ansicht1" WHERE
"lfdNr." = 3 ) AS "Name_3" 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:

```
SELECT "ID", "Name", "Nachname", ( SELECT COUNT( "ID" ) FROM "Tabelle"
WHERE "ID" <= "a"."ID" AND "Nachname" = "a"."Nachname") AS "lfdNr."
FROM "Tabelle" AS "a"
```

Jetzt kann über die erstellte Ansicht eine entsprechende Familienzusammenstellung erfolgen:

```
SELECT "Nachname", "Name" AS "Name_1", ( SELECT "Name" FROM "Ansicht1"
WHERE "lfdNr." = 2  AND "Nachname" = "a"."Nachname") AS "Name_2",
( SELECT "Name" FROM "Ansicht1" WHERE "lfdNr." = 3  AND "Nachname" =
"a"."Nachname") AS "Name_3" FROM "Ansicht1" AS "a" 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:

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

Dies erzeugt nachher:

```
Vorname Nachname
Straße
Ort
```

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.

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.

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:

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

```
SELECT "Nachname",
( SELECT "Vorname" FROM "Ansicht_Gruppe" WHERE "Nachname" = "a"."Nach-
name" AND "GruppenNr" = 1 ) ||
IFNULL( ( SELECT ', ' || "Vorname" FROM "Ansicht_Gruppe" WHERE "Nach-
name" = "a"."Nachname" AND "GruppenNr" = 2 ), '' ) ||
IFNULL( ( SELECT ', ' || "Vorname" FROM "Ansicht_Gruppe" WHERE "Nach-
name" = "a"."Nachname" AND "GruppenNr" = 3 ), '' ) ||
IFNULL( ( SELECT ', ' || "Vorname" FROM "Ansicht_Gruppe" WHERE "Nach-
name" = "a"."Nachname" AND "GruppenNr" = 4 ), '' ) ||
IFNULL( ( SELECT ', ' || "Vorname" FROM "Ansicht_Gruppe" WHERE "Nach-
name" = "a"."Nachname" AND "GruppenNr" = 5 ), '' )
AS "Vornamen"
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.