LibreOffice
The Document Foundation

Base

Kapitel 5
Abfragen

Dokumentationen zu LibreOffice unter de.libreoffice.org

Copyright

Dieses Dokument unterliegt dem Copyright © 2015. Die Beitragenden sind unten aufgeführt. Sie dürfen dieses Dokument unter den Bedingungen der GNU General Public License (*http://www.gnu.org/licenses/gpl.html*), Version 3 oder höher, oder der Creative Commons Attribution License (*http://creativecommons.org/licenses/by/3.0/*), Version 3.0 oder höher, verändern und/oder weitergeben.

Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.

Fast alle Hardware- und Softwarebezeichnungen und weitere Stichworte und sonstige Angaben, die in diesem Buch verwendet werden, sind als eingetragene Marken geschützt.

Da es nicht möglich ist, in allen Fällen zeitnah zu ermitteln, ob ein Markenschutz besteht, wird das Symbol (R) in diesem Buch nicht verwendet.

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 15.08.2022. Basierend auf der LibreOffice Version 7.4.

Inhalt

Allgemeines zu Abfragen

Abfragen an eine Datenbank sind das mächtigste Werkzeug, was uns zur Verfügung steht, um Datenbanken sinnvoll zu nutzen. Sie fassen Daten aus unterschiedlichen Tabellen zusammen, berechnen gegebenenfalls irgendwelche Ergebnisse, filtern einen ganz bestimmten Datensatz aus einer Unmenge an Daten mit hoher Geschwindigkeit heraus. Die großen Internetdatenbanken, die viele täglich nutzen, haben ihren Hauptsinn darin, dass aus der Unmenge an Informationen durch geschickte Wahl der Schlüsselwörter schnell ein brauchbares Ergebnis für den Nutzer geliefert wird – einschließlich natürlich der zu den Suchbegriffen gehörenden Anzeigen, die zum Kauf animieren sollen.

Eingabemöglichkeiten für Abfragen

Die Eingabe von Abfragen kann sowohl in der GUI als auch direkt per SQL erfolgen. In beiden Fällen öffnet sich ein Fenster, das es ermöglicht, die Abfragen auszuführen und gegebenenfalls zu korrigieren.

Abfrageerstellung mit der grafischen Benutzeroberfläche

Die Erstellung von Abfragen mit dem Assistenten wird hier nicht dargestellt, da der Assistent zwar bei wenig Tabellen schnell zu einem Ergebnis führt, für eine größere Datenbank aber wenig zielführend ist. Hier wird stattdessen die direkte Erstellung über **Abfrage in der Entwurfsansicht erstellen** erklärt.

Nach einem Aufruf der Funktion erscheinen zwei Fenster. Ein Fenster stellt die Grundlagen für den Design-Entwurf der Abfrage zur Verfügung, das andere dient dazu, Tabellen, Ansichten oder Abfragen der Abfrage hinzuzufügen.

Da unser einfaches Formular auf der Tabelle "Ausleihe" beruhte, wird zuerst einmal an dieser Tabelle die Grundkonstruktion von Abfragen mit dem Abfrageeditor erklärt.

Aus den zur Verfügung stehenden Tabellen wird die Tabelle "Ausleihe" ausgewählt. Dieses Fenster bietet die Möglichkeit, gleich mehrere Tabellen (und unter diesen auch Ansichten) sowie Abfragen miteinander zu kombinieren. Jede gewünschte Tabelle wird markiert (linke Maustaste) und dann dem grafischen Bereich des Abfrageeditors hinzugefügt.

Sind alle erforderlichen Tabellen ausgewählt, so wird dieses Fenster geschlossen. Gegebenenfalls können später noch mehr Tabellen und Abfragen hinzugefügt werden. Ohne eine einzige

Tabelle lässt sich aber keine Abfrage erstellen, so dass eine Auswahl zu Beginn schon sein muss.

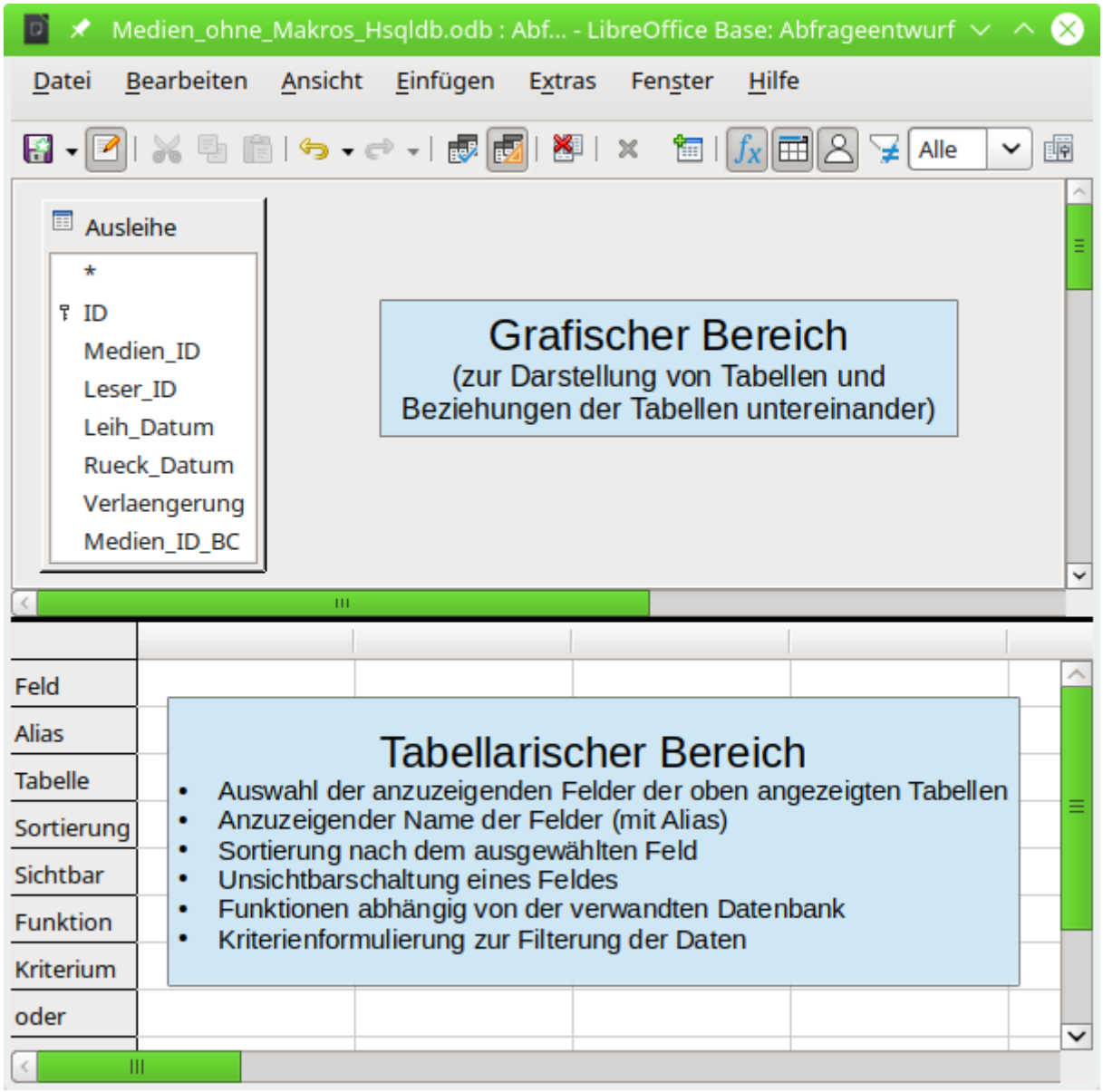


Abbildung 1: Bereiche des Abfrageentwurfs

Abbildung 1 zeigt die grundsätzliche Aufteilung des grafischen Abfrageeditors: Im grafischen Bereich werden die Tabellen angezeigt, die mit der Abfrage zusammen hängen sollen. Hier kann auch ihre Beziehung zueinander in Bezug auf die Abfrage angegeben werden. Im tabellarischen Bereich erfolgt die Auswahl der Felder, die angezeigt werden sollen, sowie Bedingungen, die mit diesen Feldern verbunden sind.

Standardmäßig werden im Abfrageeditor 2 Symbolleisten angezeigt:

Abbildung 2: Symbolleiste «Abfrageentwurf»

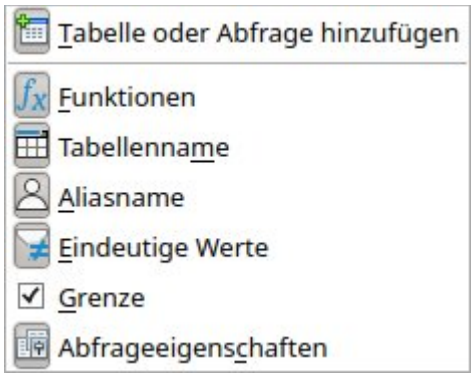
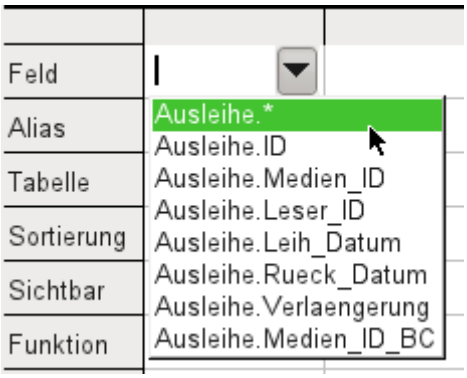


Abbildung 3: Symbolleiste «Design»

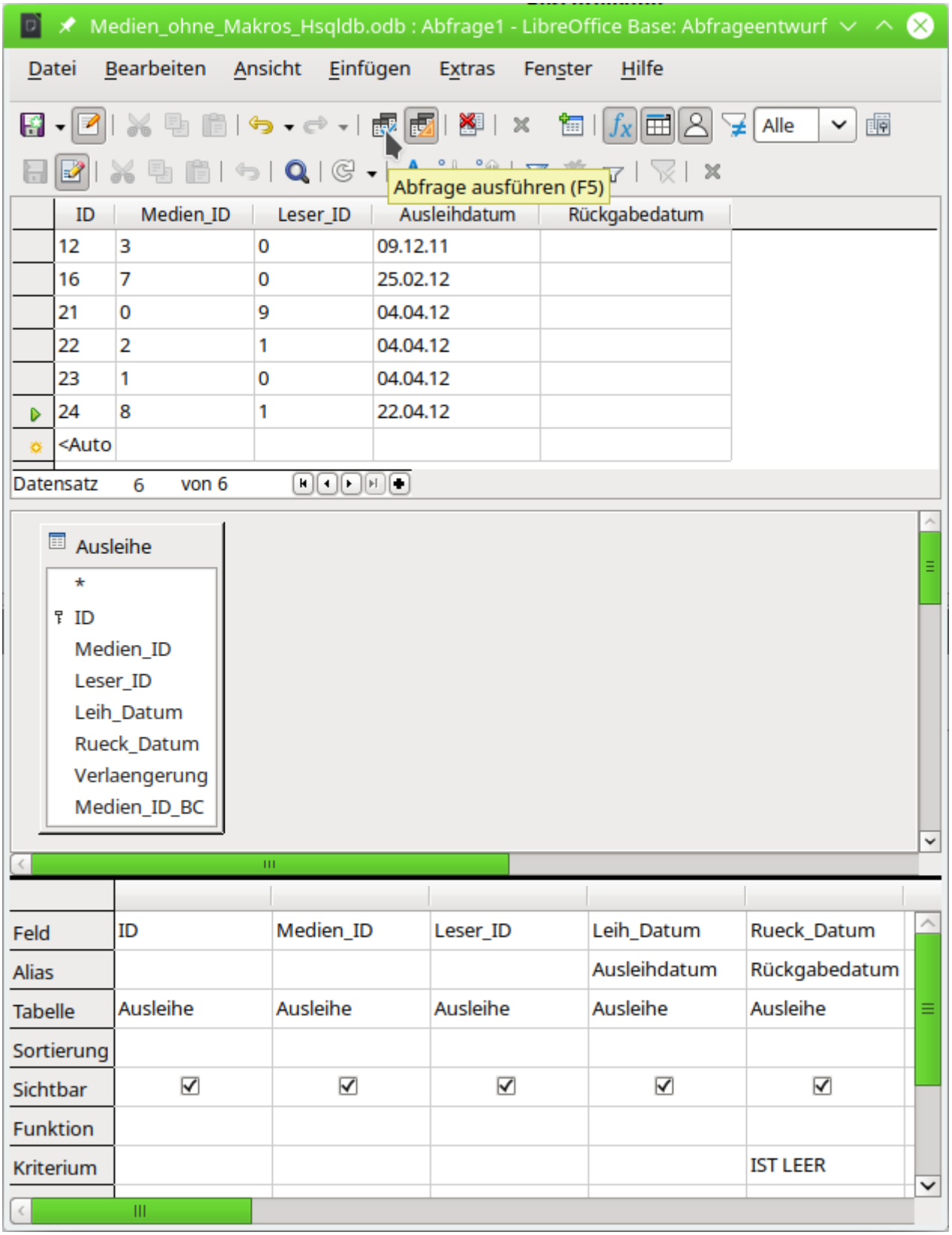
Ein Klick mit der Maustaste auf das Feld der ersten Spalte im tabellarischen Bereich öffnet die Feldauswahl:



Hier stehen jetzt alle Felder der Tabelle "Ausleihe" zur Verfügung. Die Schreibweise ist: **Tabellenname.Feldname** – deshalb beginnen alle Feldbezeichnungen hier mit dem Begriff "Ausleihe.".

Eine besondere Bedeutung hat die markierte Feldbezeichnung **Ausleihe.*** . Hier wird mit einem Klick jeder Feldname der zugrundeliegenden Tabelle in der Abfrage wiedergegeben. Wenn allein diese Feldbezeichnung mit dem Jokerzeichen «*» für alle Felder gewählt wird, unterscheidet sich die Abfrage nicht von der Tabelle.

Tipp	Sollen schnell alle möglichen Felder aus Tabellen in die Abfrage übertragen werden, so kann auch direkt im grafischen Bereich auf die Tabellenübersicht geklickt werden. Mit einem Doppelklick auf ein Feld wird das Feld im tabellarischen Bereich an der nächsten freien Position eingefügt.



Es werden die ersten fünf Felder der Tabelle Ausleihe ausgewählt. Abfragen können auch im Design-Modus immer wieder testweise ausgeführt werden. Dann erscheint oberhalb der grafischen Ansicht der Tabelle eine tabellarische Übersicht über die Daten. Die testweise Ausführung von Abfragen ist vor dem Abspeichern immer sinnvoll, damit für den Nutzer klar ist, ob die Abfrage auch wirklich das erreicht, was sie erreichen soll. Manchmal wird durch einen Denkfehler ausgeschlossen, dass eine Abfrage überhaupt je Daten ausgeben kann. Ein anderes Mal

kann es passieren, dass plötzlich genau die Datensätze angezeigt werden, die ausgeschlossen werden sollten.

Grundsätzlich lässt sich eine Abfrage, die eine Fehlerrückmeldung bei der zugrundeliegenden Datenbank produziert, gar nicht erst abspeichern.

	ID	Medien_ID
▶	12	3
	16	7
	23	1
	22	2
	24	8
	21	0
*	<Auto	

Abbildung 4: Abfrage bearbeitbar

	Medien_ID	Leser
▶	3	0
	7	0
	1	0
	2	1
	8	1
	0	9

Abbildung 5: Abfrage nicht bearbeitbar

Besonderes Augenmerk sollte in dem obigen Test einmal auf die erste Spalte des dargestellten Abfrageergebnisses geworfen werden. Auf der linken Seite der Tabelle erscheint immer der Datensatzmarkierer, der hier auf den ersten Datensatz als aktivem Datensatz hinweist. Während in *Abbildung 4* aber das erste Feld des ersten Datensatzes grün markiert ist, zeigt das erste Feld in *Abbildung 5* nur eine gestrichelte Umrandung an. Die grüne Markierung deutet bereits an, dass hier im Feld selbst etwas geändert werden kann. Die Datensätze sind also änderbar. In *Abbildung 4* ist außerdem eine zusätzliche Zeile zur Eingabe neuer Daten vorhanden, in der für das Feld "ID" schon *<AutoWert>* vorgemerkt ist. Auch hier also sichtbar, dass Neueingaben möglich sind.

Grundsätzlich sind dann keine Neueingaben möglich, wenn der Primärschlüssel der abgefragten Tabelle nicht in der Abfrage enthalten ist.

Feld	ID	Medien_ID	Leser_ID	Leih_Datum	Rueck_Datum
Alias				Ausleihdatum	Rückgabedatum

Den Feldern "Leih_Datum" und "Rueck_Datum" wurde ein Aliasname zugewiesen. Sie wurden damit nicht umbenannt, sondern unter diesem Namen für den Nutzer der Abfrage sichtbar gemacht.

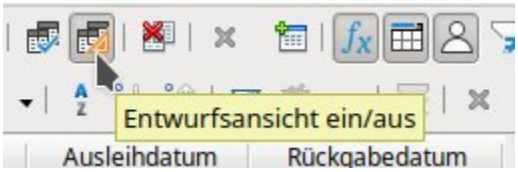
	ID	Medien_ID	Leser_ID	Ausleihdatum	Rückgabedatum
▶	12	3	0	09.12.11	

Entsprechend ist in der Tabellenansicht der Alias statt der eigentlichen Feldbezeichnung zu sehen.

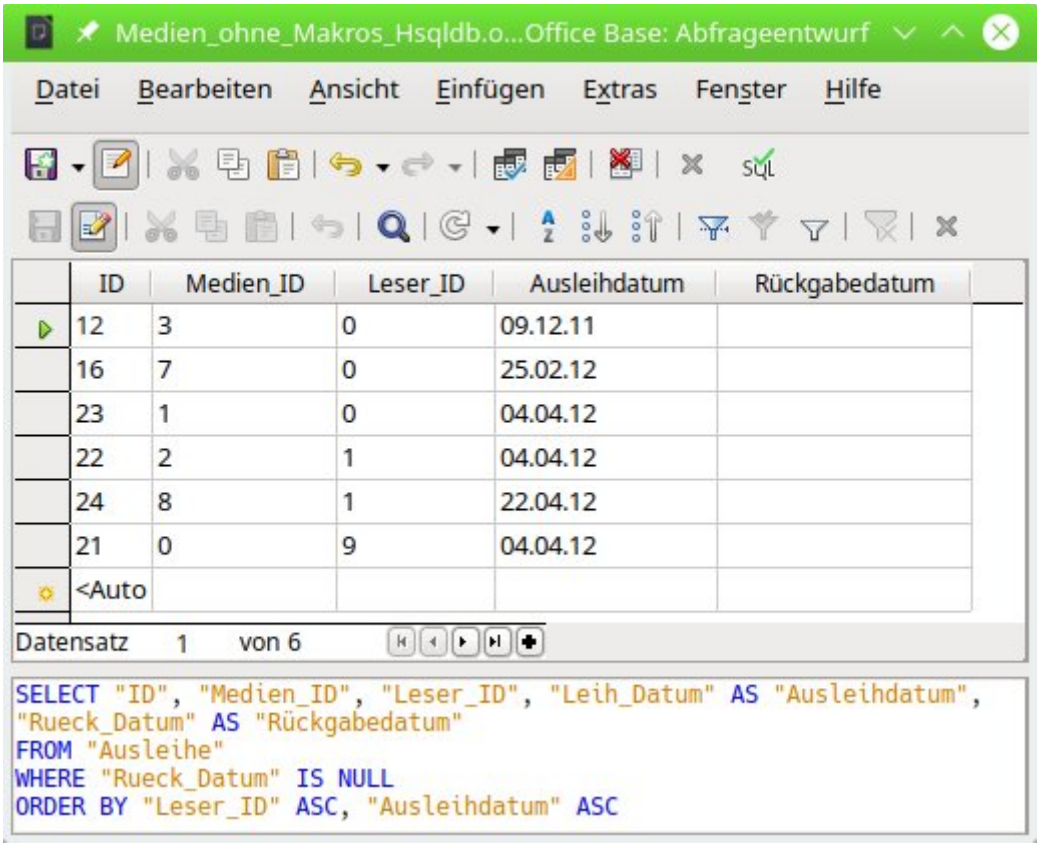
Rueck_Datum
Rückgabedatum
Ausleihe
☑
IST LEER

Dem Feld "Rueck_Datum" wurde nicht nur ein Alias, sondern auch ein Kriterium zugewiesen, nach dem nur die Datensätze angezeigt werden sollen, bei denen das Feld "Rueck_Datum" leer ist. Die Angabe erfolgt hier in deutscher Sprache, wird dann aber in der eigentlichen Abfrage in SQL übersetzt.

Durch dieses Ausschlusskriterium werden nur die Datensätze von Medien angezeigt, die ein Medium enthalten, das noch nicht zurückgegeben wurde.



Um die SQL-Sprache besser kennen zu lernen, empfiehlt es sich immer wieder einmal vom Entwurfs-Modus aus in den SQL-Darstellungsmodus zu wechseln.

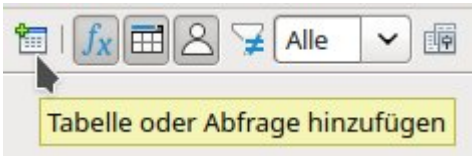


Hier wurde die durch die Auswahlen erstellte SQL-Formulierung sichtbar gemacht. Für die bessere Übersicht ist die Ansicht mit Zeilenumbrüchen versehen worden. Leider speichert der Editor diese Zeilenumbrüche standardmäßig nicht mit ab, so dass beim nächsten Aufruf die Abfrage wieder komplett als eine durchgängige Zeile mit Umbruch am Fensterrand wiedergegeben wird.

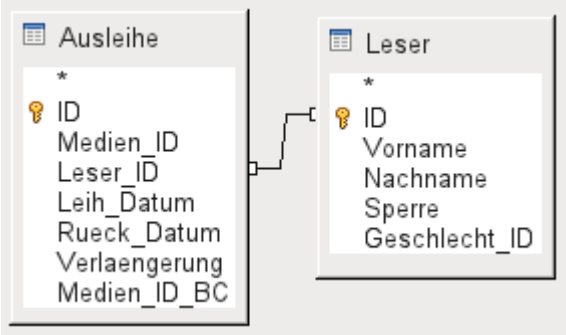
Über **SELECT** wird die Auswahl gestartet. Mit **AS** können die Aliasbezeichnungen eingeführt werden. **FROM** zeigt auf die Tabellenquelle der Abfrage. **WHERE** gibt die Bedingung für die Abfrage wieder, hier also, dass der Inhalt des Feldes "Rueck_Datum" leer ist (**IS NULL**). Mit **ORDER BY** wird die Sortierung definiert, und zwar als aufsteigend (**ASC** – ascending) für die beiden Felder "Leser_ID" und "Ausleihdatum". Diese Sortierung zeigt auch, dass die Zuweisung eines Alias das Feld "Leih_Datum" auch in der Abfrage selbst mit dem Alias ansprechbar macht.

Bisher sind die Felder "Medien_ID" und "Leser_ID" nur als Zahlenfelder sichtbar. Welchen Namen der Leser hat, bleibt unklar. Um dies in einer Abfrage anzuzeigen, muss die Tabelle Leser eingebunden werden. Um die folgende Ansicht zu erhalten, muss in den Design-Modus

zurückgeschaltet werden. Danach kann dann eine neue Tabelle in der Design-Ansicht hinzugefügt werden.



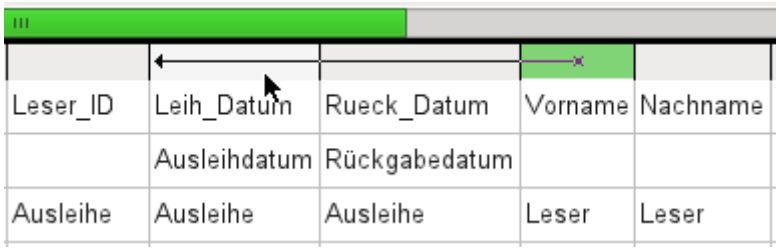
Hier können im Nachhinein weitere Tabellen oder Abfragen in der grafischen Benutzeroberfläche sichtbar gemacht werden. Sind bei der Erstellung der Tabellen Beziehungen geklärt worden (siehe Kapitel «Beziehungen zwischen Tabellen allgemein»), dann werden die Tabellen entsprechend direkt miteinander verbunden angezeigt:



Fehlt die Verbindung, so kann hier durch ein Ziehen mit der Maus von "Ausleihe"."Leser_ID" zu "Leser"."ID" eine direkte Verknüpfung erstellt werden.

Hinweis	Eine Verbindung von Tabellen geht zur Zeit nur, wenn es sich um die interne Datenbank oder ein relationales Datenbanksystem handelt. Tabellen einer Tabellenkalkulation z. B. lassen sich so nicht miteinander verbinden. Sie müssen dazu in eine interne Datenbank importiert werden. Um die Verbindung der Tabellen herzustellen, reicht ein Import ohne zusätzliche Erstellung eines Primärschlüssels aus.

Jetzt können im tabellarischen Bereich auch die Felder der Tabelle "Leser" ausgewählt werden. Die Felder werden dabei erst einmal am Schluss der Abfrage angeordnet.



Mit der Maus kann in dem tabellarischen Bereich des Editors die Lage der Felder korrigiert werden. Hier wird z. B. gerade das Feld "Vorname" direkt vor das Feld "Leih_Datum" verlegt.

	ID	Medien_ID	Leser_ID	Vorname	Nachname	Ausleihdatum	Rückgabedatum
▶	12	3	0	Bert	Lederstrumpf	09.12.11	
	16	7	0	Bert	Lederstrumpf	25.02.12	
	23	1	0	Bert	Lederstrumpf	04.04.12	
	22	2	1	Heinrich	Müller	04.04.12	
	24	8	1	Heinrich	Müller	22.04.12	
	21	0	9	Terence	Nobody	04.04.12	

Datensatz 1 von 6

Die Namen wurden jetzt sichtbar gemacht. Die "Leser_ID" ist eigentlich überflüssig. Auch ist die Sortierung nach "Nachname" und "Vorname" eigentlich sinnvoller als nach der "Leser_ID".

Diese Abfrage eignet sich nicht mehr für Base als Abfrage mit Eingabemöglichkeit, da zu der neu hinzugekommenen Tabelle "Leser" der Primärschlüssel fehlt. Erst wenn auch dieser Primärschlüssel eingebaut wird, ist die Abfrage wieder editierbar – allerdings komplett editierbar, so dass auch die Namen der Leser geändert werden können. Die Möglichkeit der Editierbarkeit ist also sehr vorsichtig zu nutzen, gegebenenfalls über ein Formular einzuschränken.

Selbst wenn die Abfrage weiter editierbar ist, lässt sie sich nicht so komfortabel nutzen wie ein Formular mit Listenfeldern, die zwar die Lesernamen anzeigen, aber die "Leser_ID" an die Tabelle weitergeben. Listenfelder lassen sich in eine Abfrage nicht einfügen. Sie sind den Formularen vorbehalten.

```
SELECT "Ausleihe"."ID", "Ausleihe"."Medien_ID", "Ausleihe"."Leser_ID",
"Leser"."Vorname", "Leser"."Nachname", "Ausleihe"."Leih_Datum" AS
"Ausleihdatum", "Ausleihe"."Rueck_Datum" AS "Rückgabedatum"
FROM "Ausleihe", "Leser"
WHERE "Ausleihe"."Leser_ID" = "Leser"."ID" AND
"Ausleihe"."Rueck_Datum" IS NULL
ORDER BY "Ausleihe"."Leser_ID" ASC, "Ausleihdatum" ASC
```

Wird jetzt auf die SQL-Ansicht umgeschaltet, so zeigt sich, dass alle Felder mit einer Doppelbezeichnung gekennzeichnet sind: **"Tabellenname"."Feldname"**. Dies ist notwendig, damit der Datenbank klar wird, aus welcher Tabelle die jeweiligen Feldinhalte stammen. Schließlich können Felder in unterschiedlichen Tabellen ohne weiteres den gleichen Feldnamen tragen. Bei den bisherigen Tabellenkonstruktionen trifft dies z. B. immer auf das Feld "ID" zu.

Hinweis

Folgende Abfrage funktioniert auch ohne Tabellenname vor den Feldnamen:

```
001 SELECT
002    "Ware"."ID",
003    "Abgang"."Anzahl",
004    "Ware"."Preis"
005 FROM "Ware",
006    "Abgang"
007 WHERE "Abgang"."Ware_ID" = "Ware"."ID"
```

Hierbei wird "ID" aus der Tabelle gezogen, die als erste in der **FROM**-Definition steht. Auch die Tabellendefinition in der **WHERE**-Formulierung wäre überflüssig, da "Ware_ID" nur einmal in der Tabelle "Abgang" vorkommt und die ID aus der Tabelle "Ware" genommen wird (Position der Tabelle). Die folgende Abfrage liefert also das gleiche Ergebnis:

```
001 SELECT
002    "ID",
003    "Anzahl",
004    "Preis"
005 FROM "Ware",
006    "Abgang"
007 WHERE "Ware_ID" = "ID"
```

Mit ~~Firebird~~ ist diese Abfrage nicht möglich, da sowohl die Tabelle "Ware" als auch die Tabelle "Abgang" ein Feld "ID" besitzen. Es erscheint stattdessen die Fehlermeldung «*Ambiguous field name between table Ware and table Abgang *ID»

Wird einem Feld in der Abfrage ein Alias zugewiesen, so kann es z. B. in der Sortierung mit diesem Alias ohne einen Tabellennamen angesprochen werden. Dies gilt nicht für ~~Firebird~~.

Hinweis

Der Alias in einer Abfrage kann bei der HSQLDB zur Beziehungsdefinition, zum Sortieren usw. genutzt werden. In Firebird muss hingegen der ursprüngliche Ausdruck an der entsprechenden Stelle stehen.

Sortierungen werden in der grafischen Benutzeroberfläche nach der Reihenfolge der Felder in der Tabellenansicht vorgenommen. Sollte stattdessen zuerst nach "Ausleihdatum" und dann nach "Ausleihe"."Leser_ID" sortiert werden, so kann dies erzeugt werden, indem

- die Reihenfolge der Felder im tabellarischen Bereich der grafischen Benutzeroberfläche geändert wird,
- ein zweites Feld eingefügt wird, das auf unsichtbar geschaltet ist und nur die Sortierung gewährleisten soll (wird allerdings beim Editor nur vorübergehend angenommen, wenn kein Alias definiert wurde) oder
- der Text für die **ORDER BY** – Anweisung im SQL-Editor entsprechend umgestellt wird.

Die Sortierung aus der SQL-Ansicht wird in die GUI korrekt übernommen und entsprechend mit nicht sichtbaren Feldern in der grafischen Benutzeroberfläche angezeigt.

Funktionen in der Abfrage

Mittels Funktionen lässt sich aus Abfragen auch mehr ersehen als nur ein gefilterter Blick auf die Daten einer oder mehrerer Tabellen. In der folgenden Abfrage wird, abhängig von der "Leser_ID", gezählt, wie viele Medien ausgeliehen wurden.

Feld	ID	Leser_ID	Rueck_Datum
Alias	Anzahl		
Tabelle	Ausleihe	Ausleihe	Ausleihe
Sortierung			
Sichtbar	☑	☑	☐
Funktion	Anzahl	Gruppierung	
Kriterium			IST LEER

Für die "ID" der Tabelle "Ausleihe" wird die Funktion **Anzahl** ausgewählt. Prinzipiell ist hier egal, welches Feld einer Tabelle gewählt wurde. Die einzige Bedingung: *Das Feld darf nicht in irgendwelchen Datensätzen leer sein*. Aus diesem Grunde ist der Primärschlüssel, der ja nie leer ist, immer eine geeignete Wahl. Gezählt werden die Felder, die einen Inhalt enthalten, der von **NULL** verschieden ist.

Hinweis	Sobald für ein Feld eine andere Funktion als die Gruppierung gewählt wird, sollte das Feld einen **Aliasnamen** erhalten. Ansonsten kann es bei der Weiterverarbeitung z.B. in Berichten, verschachtelten Abfragen oder auch Ansichten zu Problemen kommen.

Für die "Leser_ID", die ja Rückschlüsse auf den Leser zulässt, wird als Funktion die **Gruppierung** gewählt. Dadurch werden die Datensätze nach der "Leser_ID" zusammengefasst. So zählt denn die Anweisung die Datensätze, die zu jeder "Leser_ID" passen.

Als Kriterium ist wie in den vorhergehenden Beispielen das "Rueck_Datum" auf **IST LEER** gesetzt.

	Anzahl	Leser_ID
	3	0
	1	9
▶	2	1

Datensatz 3 von 3

```
SELECT COUNT( "ID" ) AS "Anzahl", "Leser_ID"
FROM "Ausleihe"
WHERE "Rueck_Datum" IS NULL
GROUP BY "Leser_ID"
```

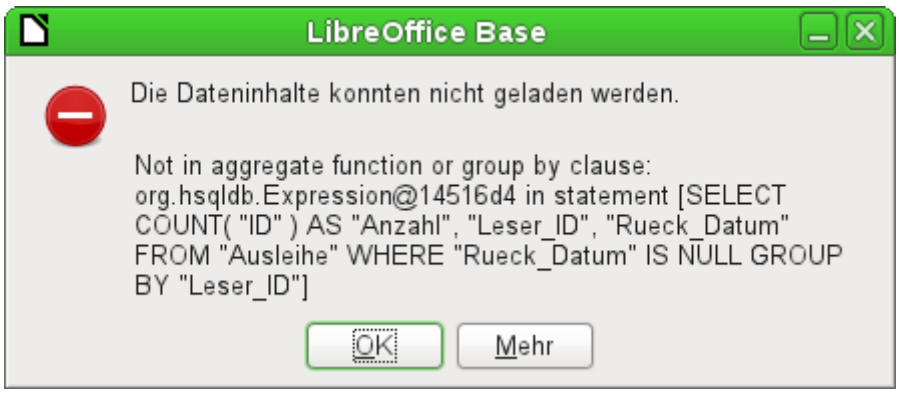
Die Abfrage zeigt im Ergebnis, dass z. B. "Leser_ID" '0' insgesamt noch 3 Medien entliehen hat. Wäre die Funktion **Anzahl** statt der "ID" dem "Rueck_Datum" zugewiesen worden, so würden für alle "Leser_ID" jeweils '0' entliehene Medien dargestellt, da ja "Rueck_Datum" als **LEER** vordefiniert ist.

Die entsprechende Formulierung für den SQL-Code ist oben wieder abgebildet. Aus dem Begriff **Anzahl** der deutschen GUI wird **COUNT()**. Aus dem Begriff **Gruppierung** wird der Zusatz **GROUP BY**.

Insgesamt stehen über die grafische Benutzeroberfläche folgende Funktionen zur Verfügung, die ihre Entsprechung zu Funktionen in der zugrundeliegenden HSQLDB haben:

Eine Erläuterung zu den Funktionen ist in dem folgenden Kapitel *Abfrageerweiterungen im SQL-Modus* nachzulesen. Die Funktionen «Sammeln», «Vereinigung» und «Durchschnitt» funktionieren mit keiner der eingebauten Datenbanken. Sie sind Standardfunktionen von Oracle.

Wird einem Feld in einer Abfrage eine Sammelfunktion hinzugefügt, so müssen alle anderen Felder der Abfrage auch mit Funktionen versehen sein, sofern die Felder sichtbar sein sollen. Dies liegt daran, dass in einem Datensatz nicht plötzlich zwischendurch Felder mehrere Datensätze abbilden können. Wird dies nicht beachtet, so erscheint die folgende Fehlermeldung:



Etwas frei übersetzt: Der folgenden Ausdruck enthält ein Feld ohne eine der Sammelfunktionen oder eine Gruppierung.

Danach wird die gesamte Abfrage aufgelistet, leider ohne das Feld konkret zu benennen. Hier wurde einfach das Feld "Rueck_Datum" als sichtbar hinzugefügt. Dieses Feld hat keine Funktion zugewiesen bekommen und ist auch nicht in der Gruppierung enthalten.

Die über den Button Mehr erreichbaren Informationen sind für den Normalnutzer einer Datenbank nicht aufhellender. Hier wird lediglich zusätzlich noch der SQL-Fehlercode aufgeführt.

Innerhalb der GUI können auch die Grundrechenarten sowie weitere Funktionen angewandt werden.

Feld	ID	Medien_ID	Leser_ID	Datum	Anzahl("Mahnung"."Datum") * 2	Rueck_Datum
Alias				Mahnzahl	Mahnbetrag	
Tabelle	Ausleihe	Ausleihe	Ausleihe	Mahnung		Ausleihe
Sortierung						
Sichtbar	☑	☑	☑	☑	☑	☐
Funktion	Gruppierung	Gruppierung	Gruppierung	Anzahl		
Kriterium						IST LEER

Hier wurden die Tabelle "Ausleihe" und die Tabelle "Mahnung" zusammen abgefragt. Aus der Zahl der Datumseinträge in der Tabelle "Mahnung" wird auf die Anzahl der Mahnungen geschlossen. Als Mahnbetrag wird in der Abfrage 2,- € festgelegt. Statt der Feldauswahl wird in das Feld einfach geschrieben: **`Anzahl(Mahnung.Datum)*2`** . Die grafische Benutzeroberfläche setzt anschließend die Anführungsstriche und wandelt den Begriff Anzahl in den entsprechenden SQL-Begriff um.

Vorsicht

Werden in der grafischen Benutzeroberfläche Zahlen mit Nachkommastellen eingegeben, so ist auf jeden Fall darauf zu achten, dass statt eines Kommas ein Punkt der Trenner für Dezimalzahlen in SQL ist. Kommata sind hingegen die Trenner der Felder. Deshalb werden einfach neue Abfragefelder gegründet, die die Nachkommastellen ausgeben.

	ID	Medien_ID	Leser_ID	Mahnzahl	Mahnbetrag
▶	12	3	0	2	4
	16	7	0	1	2
	23	1	0	1	2

Datensatz 1 von 3

```
SELECT "Ausleihe"."ID", "Ausleihe"."Medien_ID", "Ausleihe"."Leser_ID",
COUNT( "Mahnung"."Datum" ) AS "Mahnzahl",
COUNT( "Mahnung"."Datum" ) * 2 AS "Mahnbetrag"
FROM "Mahnung", "Ausleihe"
WHERE "Mahnung"."Ausleihe_ID" = "Ausleihe"."ID"
AND "Ausleihe"."Rueck_Datum" IS NULL
GROUP BY "Ausleihe"."ID", "Ausleihe"."Medien_ID", "Ausleihe"."Leser_ID"
```

Die Abfrage ermittelt jetzt für jedes noch entliehene Medium anhand der herausgegebenen Mahnungen und der zusätzlich eingefügten Multiplikation die Mahngebühren. Die folgende Abfragekonstruktion hilft weiter, wenn die Gebühren für jeden Leser berechnet werden sollen:

Feld	Leser_ID	Datum	Anzahl("Mahnung"."Datum") * 2	Rueck_Datum
Alias		Mahnzahl	Mahnbetrag	
Tabelle	Ausleihe	Mahnung		Ausleihe
Sortierung				
Sichtbar	☑	☑	☑	☐
Funktion	Gruppierung	Anzahl		
Kriterium				IST LEER

Die Felder "Ausleihe"."ID" und "Ausleihe"."Medien_ID" wurden entfernt. Sie erzeugten in der vorherigen Abfrage über die Gruppierung für jedes Medium einen separaten Datensatz. Jetzt wird nur noch nach den Lesern gruppiert. Das Abfrageergebnis sieht dann so aus:

	Leser_ID	Mahnzahl	Mahnbetrag
▶	0	4	8

Statt die Medien für "Leser_ID" '0' separat aufzulisten werden alle Felder aus "Mahnung"."Datum" zusammengezählt und die Summe von 8,- € als Mahngebühr ermittelt.

Beziehungsdefinition in der Abfrage

Werden Daten in Tabellen oder einem Formular gesucht, so beschränkt sich die Suche in der Regel auf eine Tabelle bzw. auf ein Formular. Selbst der Weg von einem Hauptformular zu einem Unterformular ist für die eingebauten Suchfunktionen nicht gangbar. Da bietet es sich dann an, zu durchsuchende Daten mit einer Abfrage zusammenzufassen.

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

Datensatz 1 von 9

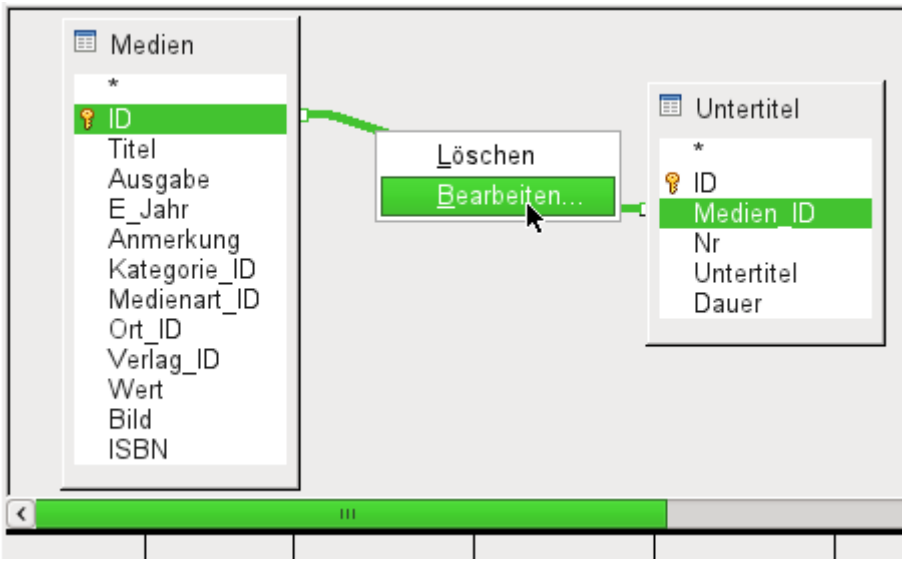
Feld	Titel
Alias	
Tabelle	Medien
Sortierung	
Sichtbar	☑ ☐

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

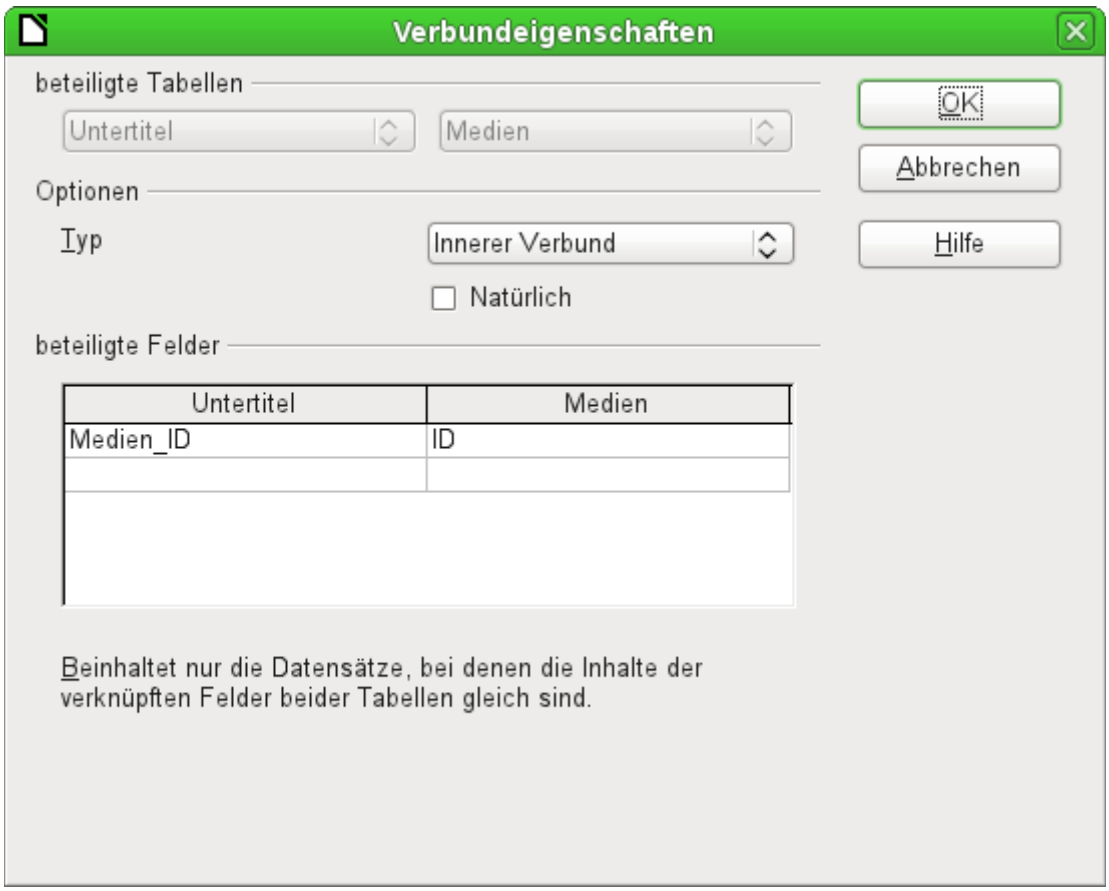
Datensatz 1 von 8

Feld	Titel	Untertitel
Alias		
Tabelle	Medien	Untertitel
Sortierung		
Sichtbar	☑	☑ ☐

Die einfache Abfrage an die "Titel" aus der Tabelle "Medien" zeigt den eingegebenen Testbestand dieser Tabelle mit 9 Datensätzen an. Wird jedoch die Tabelle "Untertitel" mit in die Abfrage aufgenommen, so reduziert sich der Datenbestand aus der Tabelle "Medien" auf lediglich 2 "Titel". Nur für diese beiden "Titel" gibt es auch "Untertitel" in der Tabelle "Untertitel". Für alle anderen "Titel" existieren keine "Untertitel". Dies entspricht der Verknüpfungsbedingung, dass nur die Datensätze angezeigt werden sollen, bei denen in der Tabelle "Untertitel" das Feld "Medien_ID" gleich dem Feld "ID" aus der Tabelle "Medien" ist. Alle anderen Datensätze werden ausgeschlossen.



Die Verknüpfungsbedingung muss zum Bearbeiten geöffnet werden, damit alle gewünschten Datensätze angezeigt werden. Es handelt sich hier **nicht** um die Verknüpfung von Tabellen im *Relationenentwurf,* **sondern** um die Verknüpfung in einer *Abfrage*.



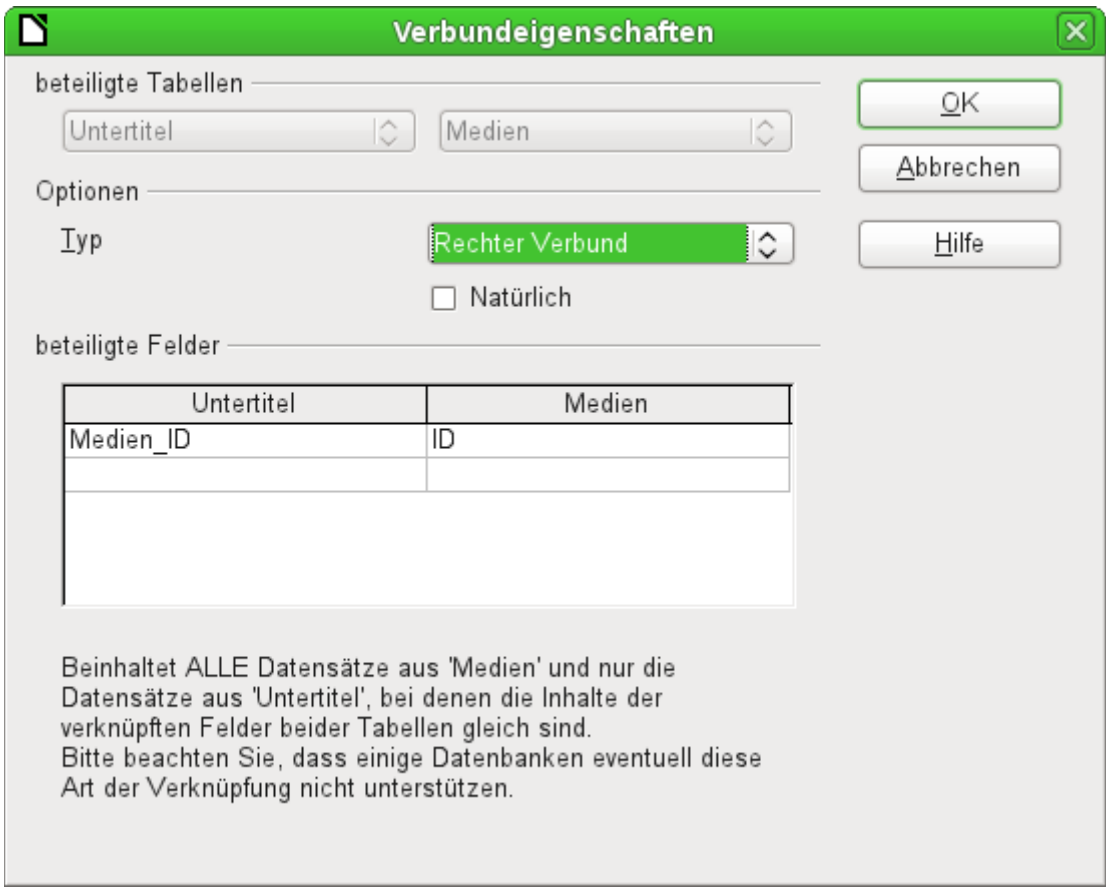
Standardmäßig steht die Verknüpfung als **Innerer Verbund** zur Verfügung. Das Fenster gibt darüber Aufschluss, wie diese Form der Verknüpfung sich auswirkt.

Als beteiligte Tabellen werden die beiden vorher ausgewählten Tabellen gelistet. Sie sind hier nicht wählbar. Die beteiligten Felder der beiden Tabellen werden aus der Tabellendefinition ausgelesen. Ist eine Beziehung in der Tabellendefinition nicht vorgegeben, so kann sie hier für die Abfrage erstellt werden. Eine saubere Datenbankplanung mit der HSQLDB sieht aber so aus, dass auch an diesen Feldern nichts zu verstellen ist.

Wichtigste Einstellung ist die Option des *Verbundes*. Hier können Verknüpfungen so gewählt werden, dass alle Datensätze von der Tabelle "Untertitel" und nur die Datensätze aus "Medien" gewählt werden, die in der Tabelle "Untertitel" auch "Untertitel" verzeichnet haben.

Umgekehrt kann gewählt werden, dass auf jeden Fall alle Datensätze aus der Tabelle "Medien" angezeigt werden – unabhängig davon, ob für sie auch "Untertitel existieren.

Die Option **Natürlich** setzt voraus, dass die zu verknüpfenden Felder in den Tabellen gleich lauten. Auch von dieser Einstellung ist Abstand zu nehmen, wenn bereits zu Beginn bei der Datenbankplanung die Beziehungen definiert wurden.



Für den **Typ → Rechter Verbund** zeigt die Beschreibung an, dass aus der Tabelle "Medien" (die in der Abbildung *rechts* angezeigte Tabelle) auf jeden Fall alle Datensätze angezeigt werden. Da es keine "Untertitel" gibt, die nicht in "Medien" mit einem "Titel" verzeichnet sind, sehr wohl aber "Titel" in "Medien", die nicht mit einem "Untertitel" versehen sind, ist dies also die richtige Wahl.

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

Datensatz 1 von 15

Nach Bestätigung des *rechten Verbundes* sieht das Abfrageergebnis aus wie gewünscht. "Titel" und "Untertitel" werden komplett zusammen in einer Abfrage angezeigt. Natürlich kommen jetzt "Titel" wie in der vorhergehenden Verknüpfung mehrmals vor. Solange allerdings Suchtreffer nicht gezählt werden, könnte diese Abfrage im weiteren Verlauf als Grundlage für

eine Suchfunktion dienen. Siehe hierzu die Codeschnipsel in diesem Kapitel, im Kapitel «Makros» («Suchen von Datensätzen») und im Kapitel «DB-Aufgaben komplett» («Datensuche»).

Abfrageeigenschaften definieren

Mit der Version 4.1 von LibreOffice ist es möglich, in dem Abfrageeditor zusätzliche Abfrageeigenschaften zu definieren.

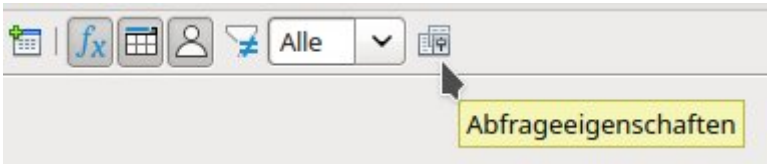
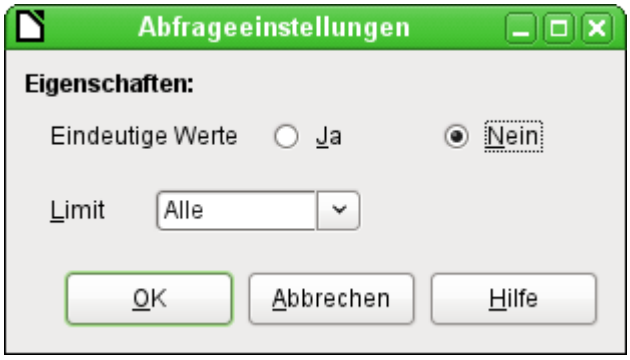


Abbildung 6: Aufruf der Abfrageeigenschaften im Abfrageeditor (ab LO 4.1)

Neben dem Button zum Aufruf der Abfrageeigenschaften befindet sich noch ein Kombinationsfeld, mit dem die Anzahl der anzuzeigenden Datensätze reguliert werden kann, sowie ein Button Eindeutige Werte. Diese Funktionen sind zusätzlich noch einmal in dem folgenden Dialog untergebracht:



alternativ:

Mit der Einstellung **Eindeutige Werte** wird beeinflusst, ob gleichlautende Datensätze in den Abfragen unterdrückt werden sollen.

Vorname	Nachname	Rueck_Datum
Lisa	Gerd	
Lisa	Gerd	
Lisa	Gerd	
Bert	Lederstrumpf	
Bert	Lederstrumpf	
Bert	Lederstrumpf	
Bert	Lederstrumpf	
Bert	Lederstrumpf	
Heinrich	Müller	
Heinrich	Müller	
Terence	Nobody	

In einer Abfrage soll ermittelt werden, welche Leser und Leserinnen noch Medien entliehen haben. Die Namen werden angezeigt, wenn das Rückgabedatum leer ist. Allerdings werden die Namen mehrmals angezeigt, wenn ein Leser oder eine Leserin noch mehrere Medien entliehen hat.

	Vorname	Nachname	Rueck_Datum
▶	Lisa	Gerd	
	Bert	Lederstrumpf	
	Heinrich	Müller	
	Terence	Nobody	

Wird **Eindeutige Werte** ausgewählt, so verschwinden die Datensätze mit gleichem Inhalt.

Die Abfrage sind dann so aus:

```
SELECT DISTINCT
"Leser"."Vorname", "Leser"."Nachname", "Ausleihe"."Rueck_Datum"
FROM "Ausleihe", "Leser"
WHERE "Ausleihe"."Leser_ID" = "Leser"."ID" AND "Ausleihe"."Rueck_Datum" IS NULL
ORDER BY "Leser"."Nachname" ASC
```

Der ursprünglichen Abfrage

```
001 SELECT "Leser"."Vorname", "Leser"."Nachname" …
```

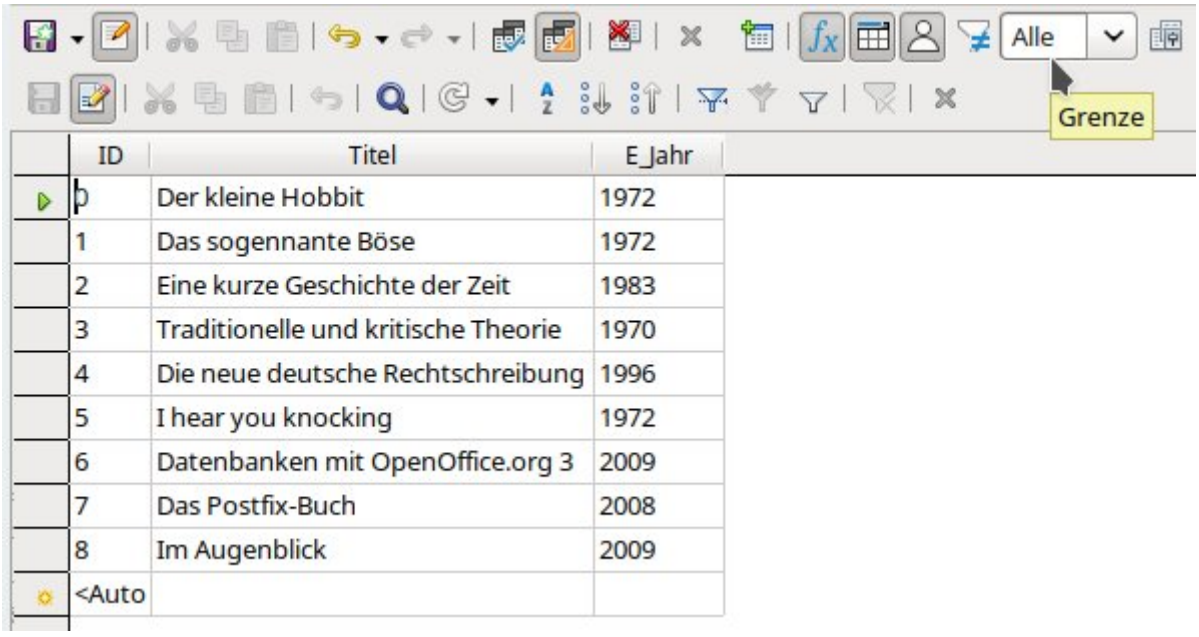
wird ein **DISTINCT** hinzugefügt:

```
001 SELECT DISTINCT "Leser"."Vorname", "Leser"."Nachname" …
```

Damit werden alle gleichlautenden Zeilen unterdrückt.

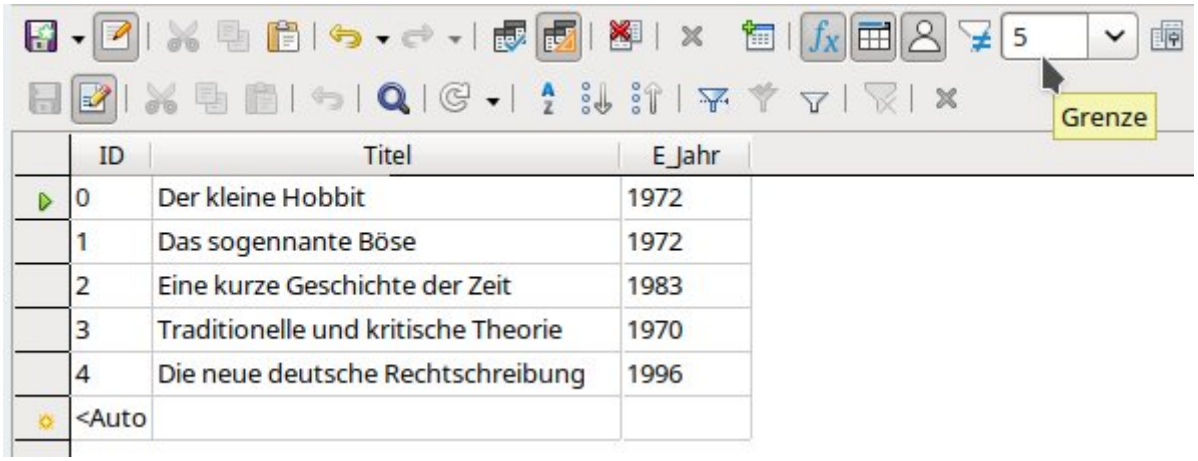
Die Auswahl eindeutiger Werte war auch in den Vorversionen möglich. Allerdings musste hier von der Design-Ansicht in die SQL-Ansicht umgeschaltet werden, um den Begriff **DISTINCT** einzufügen. Diese Eigenschaft ist also ohne Probleme abwärtskompatibel zu den Vorversionen von LO.

Mit der Einstellung **Grenze** (SQL: **Limit**) wird beeinflusst, wie viele Datensätze in der Abfrage angezeigt werden sollen. Es wird also nur eine begrenzte Zahl an Datensätzen wieder gegeben.



	ID	Titel	E_Jahr
▶	0	Der kleine Hobbit	1972
	1	Das sogennante Böse	1972
	2	Eine kurze Geschichte der Zeit	1983
	3	Traditionelle und kritische Theorie	1970
	4	Die neue deutsche Rechtschreibung	1996
	5	I hear you knocking	1972
	6	Datenbanken mit OpenOffice.org 3	2009
	7	Das Postfix-Buch	2008
	8	Im Augenblick	2009
✱	<Auto		

Alle Datensätze der Tabelle "Medien" werden angezeigt. Die Abfrage ist editierbar, da auch der Primärschlüssel enthalten ist.



ID	Titel	E_Jahr
0	Der kleine Hobbit	1972
1	Das sogennante Böse	1972
2	Eine kurze Geschichte der Zeit	1983
3	Traditionelle und kritische Theorie	1970
4	Die neue deutsche Rechtschreibung	1996
<Auto		

Nur die ersten fünf Datensätze werden angezeigt (ID 0 bis 4). Eine Sortierung wurde nicht vorgewählt. Die Standardsortierung ist hier die nach dem Primärschlüssel, sofern nichts anderes festgelegt wurde. Die Abfrage ist trotz der Begrenzung weiterhin editierbar. Dies unterscheidet die Eingabe im grafischen Modus von der, die in früheren Versionen nur mit dem direkten SQL-Modus erreichbar ist.

```
SELECT "ID", "Titel", "E_Jahr" FROM "Medien" LIMIT 5
```

Der ursprünglichen Abfrage wurde lediglich «LIMIT 5» hinzugefügt. Die entsprechende Größe des Limits kann beliebig festgelegt werden.

Vorsicht

Die Einstellung des Limits durch die grafische Benutzeroberfläche ist nicht abwärtskompatibel. In allen LO-Versionen vor der Version 4.1 konnte ein Limit nur im direkten SQL-Modus eingegeben werden. Dort erforderte das Limit eine Sortierung (**`ORDER BY …`**) oder eine Bedingung (**`WHERE …`**).

Ohne eine entsprechend eingestellte Sortierung ist es auch nicht möglich, aus einem mit der GUI erstellten Limit eine Ansicht zu erstellen.

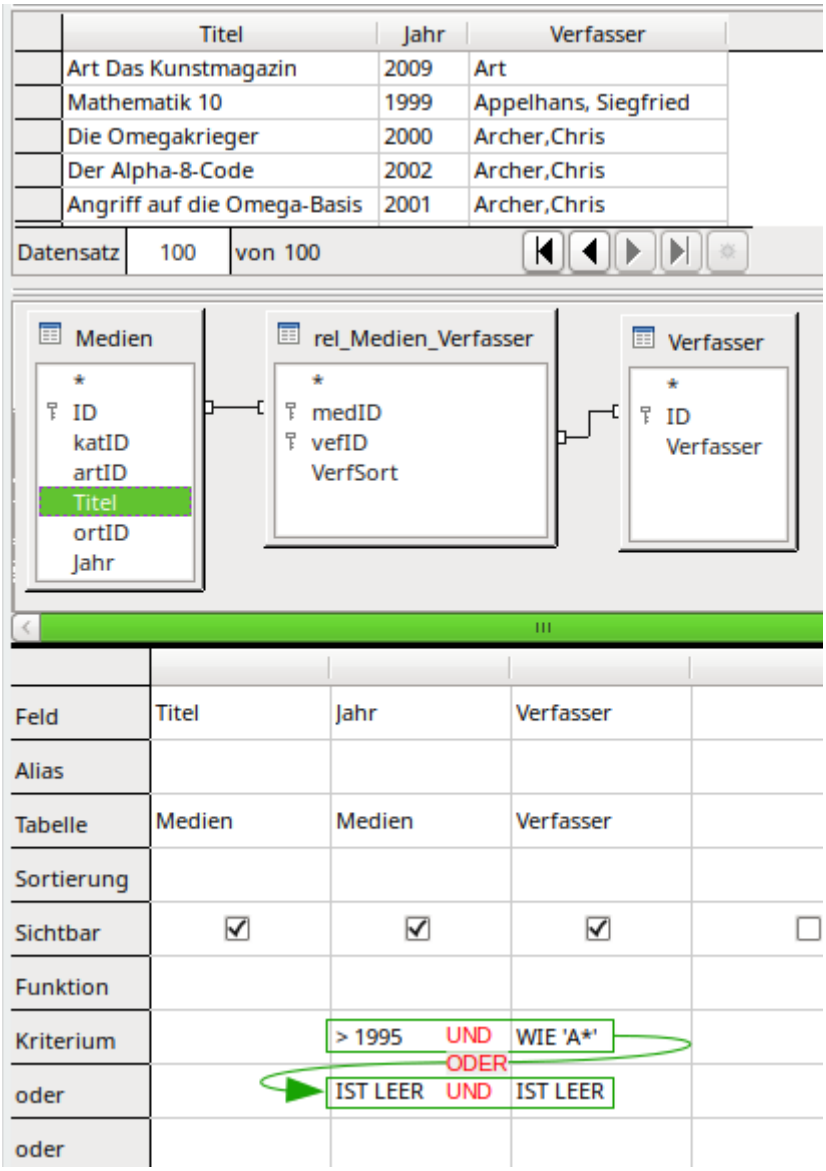
Abfragen nach Filterkriterien durchsuchen

Die Abfrage-GUI bietet eine einfache Möglichkeit, bestimmte Kriterien für die Datensuche festzulegen. Diese Kriterien werden unterhalb der Funktionen zusammengestellt.

Kriterien, die in einer Zeile nebeneinander stehen, werden mit **`UND`** verbunden. Es gelten die Eingaben der Zeile alle zusammen. Alle Eingaben in einer Zeile müssen also erfüllt sein, damit die Daten angezeigt werden. Diese Einträge in einer Zeile werden immer zuerst gelesen.

Kriterien, die untereinander stehen werden mit **`ODER`** verbunden. Es gilt entweder die Angabe in der einen Zeile oder die Angabe in der anderen Zeile.

Diese Einstellungen über die GUI sind in der Praxis manchmal nicht so leicht zu durchschauen. Deswegen hier ein Beispiel anhand von zwei Datenfeldern, das deutlich machen soll, wie die GUI-Konstruktion der Kriterien funktioniert:

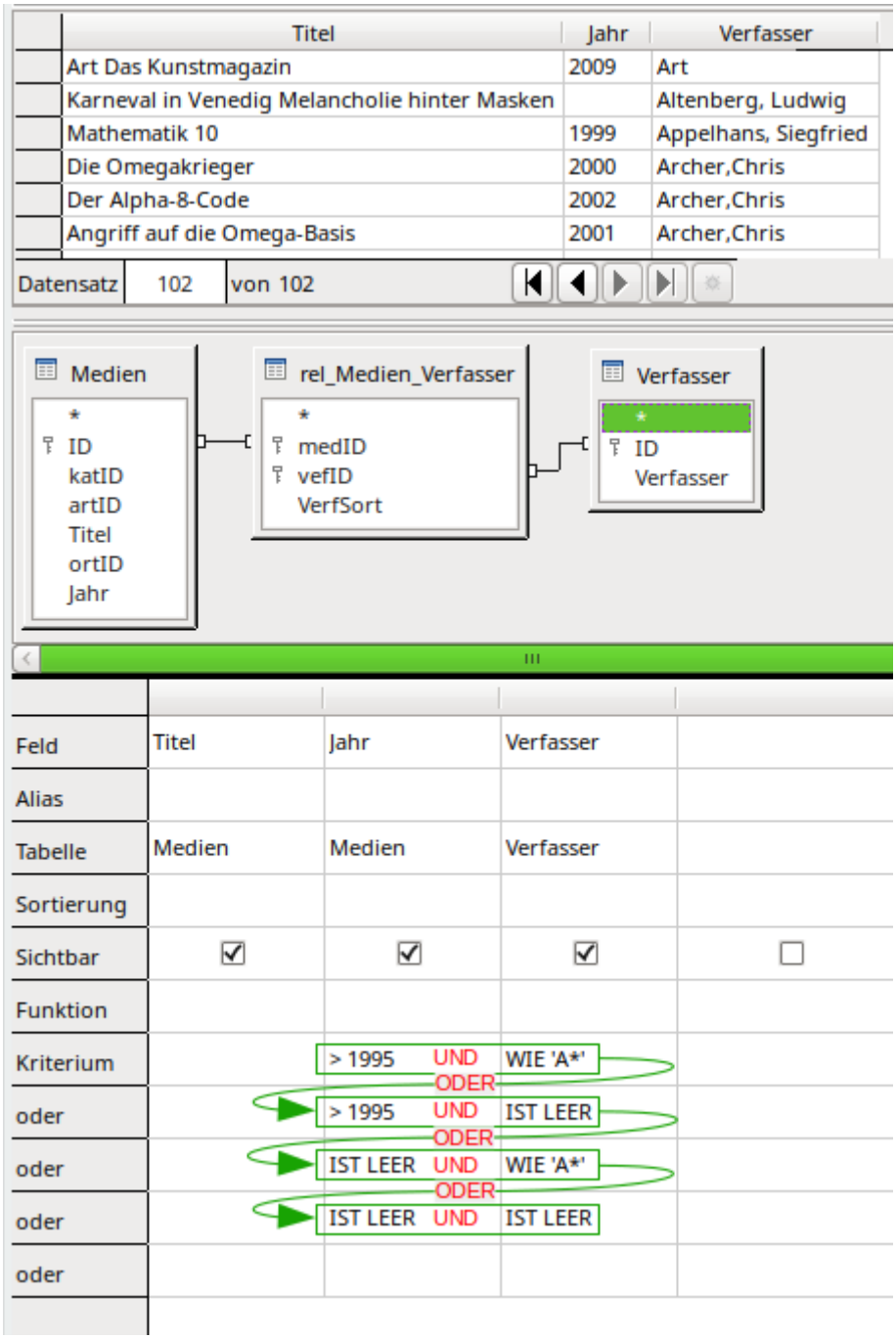


Aus den verbundenen Tabellen sollen die Datensätze angezeigt werden, bei denen das Jahr > 1995 ist oder der Eintrag für das Jahr leer ist. Außerdem sollen nur Verfasser mit 'A' beginnend angezeigt werden oder Datensätze, bei denen der Eintrag im Feld "Verfasser" leer ist.

Das in der obigen Einstellung der GUI entwickelte Schema für die Kriterien erfüllt diese Bedingungen leider *nicht*:
Das erste Kriterium ist für die Abfrage-GUI, dass das Jahr > 1995 sein muss **UND** "Verfasser" mit 'A' beginnen soll.
Das zweite Kriterium sagt aus, dass das Feld "Jahr" **UND** das Feld "Verfasser" leer sein sollen.

Damit gilt also: Entweder enthalten beide die geforderten Daten **ODER** beide sind leer, aber nicht, dass auch nur ein Feld die erforderlichen Inhalte hat und das andere leer ist. Die Kriterien werden zuerst in der Zeile und dann in der Spalte ausgelesen.

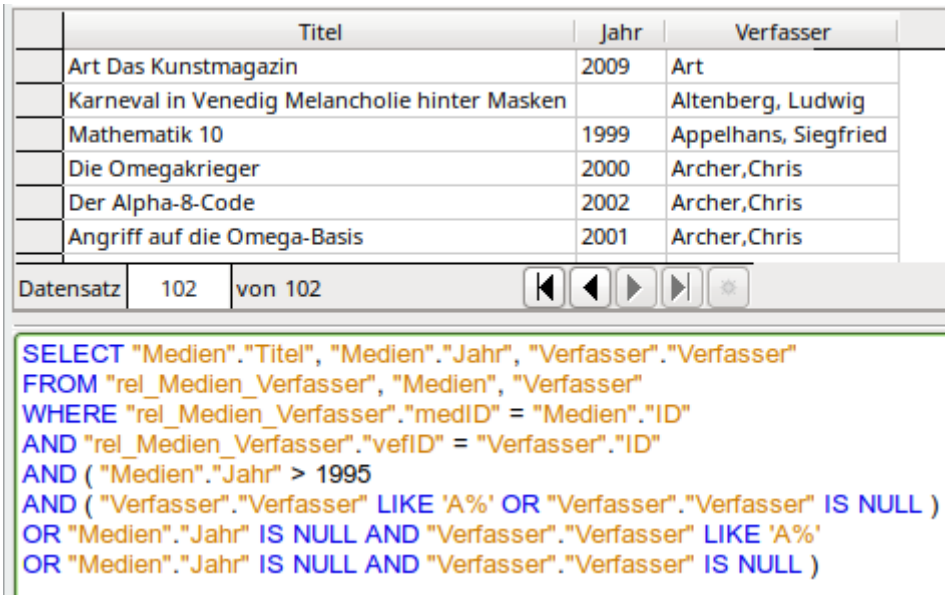
In der obigen Abfrage ist so kein Datensatz enthalten, bei dem z.B. der Eintrag für das Jahr fehlt. Die Anzahl der Datensätze ergibt hier 100.



Hier ist die Anzahl der Datensätze auf 102 angestiegen. Die Abfrage hat jetzt auch Felder entdeckt, bei denen das Jahr leer geblieben ist, aber ein Eintrag im Feld "Verfasser" vorhanden war. Hier im Beispiel das Buch «Karneval ...», das im vorherigen Screenshot nicht auftaucht.

Alle notwendigen Kombinationen stehen in der GUI:

- "Jahr" **UND** "Verfasser" haben beide einen gewünschten Inhalt, sind beide nicht leer **ODER**
- "Jahr" hat einen Inhalt **UND** "Verfasser" ist leer **ODER**
- "Jahr" ist leer **UND** "Verfasser" hat einen gewünschten Inhalt **ODER**
- "Jahr" **UND** "Verfasser" sind beide leer.

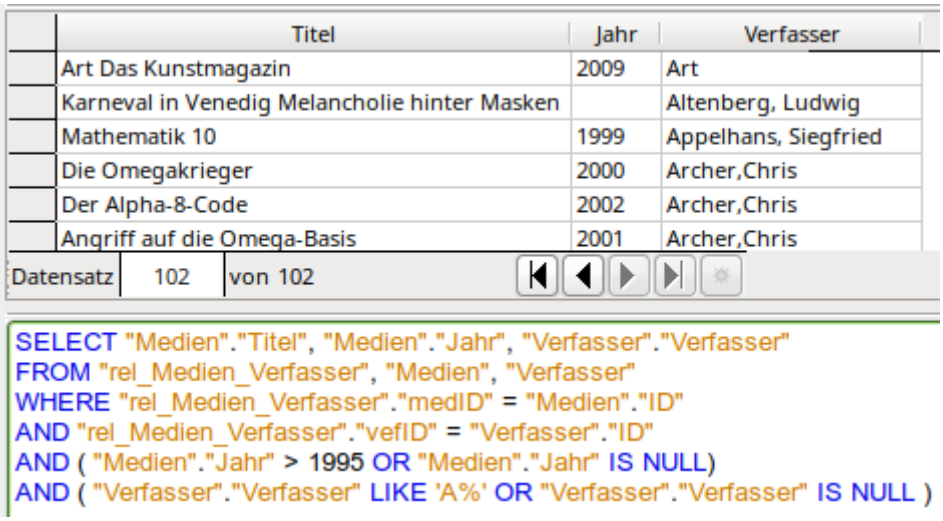


Titel	Jahr	Verfasser
Art Das Kunstmagazin	2009	Art
Karneval in Venedig Melancholie hinter Masken		Altenberg, Ludwig
Mathematik 10	1999	Appelhans, Siegfried
Die Omegakrieger	2000	Archer,Chris
Der Alpha-8-Code	2002	Archer,Chris
Angriff auf die Omega-Basis	2001	Archer,Chris

Datensatz 102 von 102

```
SELECT "Medien"."Titel", "Medien"."Jahr", "Verfasser"."Verfasser"
FROM "rel_Medien_Verfasser", "Medien", "Verfasser"
WHERE "rel_Medien_Verfasser"."medID" = "Medien"."ID"
AND "rel_Medien_Verfasser"."vefID" = "Verfasser"."ID"
AND ( "Medien"."Jahr" > 1995
AND ( "Verfasser"."Verfasser" LIKE 'A%' OR "Verfasser"."Verfasser" IS NULL )
OR "Medien"."Jahr" IS NULL AND "Verfasser"."Verfasser" LIKE 'A%'
OR "Medien"."Jahr" IS NULL AND "Verfasser"."Verfasser" IS NULL )
```

Der durch die GUI erzeugte Code sieht leider reichlich unübersichtlich aus. Der Eintrag von "Jahr" > 1995 wird über **AND** sowohl mit dem Anfangsbuchstaben des Verfassers als auch mit dem leeren Feld von "Verfasser" über **OR** verbunden. Damit werden die beiden ersten Zeilen des Kriteriums erledigt. Die anschließenden Zeilen werden dann nacheinander mit **OR** abgearbeitet.



Titel	Jahr	Verfasser
Art Das Kunstmagazin	2009	Art
Karneval in Venedig Melancholie hinter Masken		Altenberg, Ludwig
Mathematik 10	1999	Appelhans, Siegfried
Die Omegakrieger	2000	Archer,Chris
Der Alpha-8-Code	2002	Archer,Chris
Angriff auf die Omega-Basis	2001	Archer,Chris

Datensatz 102 von 102

```
SELECT "Medien"."Titel", "Medien"."Jahr", "Verfasser"."Verfasser"
FROM "rel_Medien_Verfasser", "Medien", "Verfasser"
WHERE "rel_Medien_Verfasser"."medID" = "Medien"."ID"
AND "rel_Medien_Verfasser"."vefID" = "Verfasser"."ID"
AND ( "Medien"."Jahr" > 1995 OR "Medien"."Jahr" IS NULL)
AND ( "Verfasser"."Verfasser" LIKE 'A%' OR "Verfasser"."Verfasser" IS NULL )
```

Schöner wäre es, wenn die GUI einen Code wie den obigen erzeugen könnte. Die Abfrage-GUI kann diesen Code so aber nicht erstellen, da zuerst die Zeile mit **UND** verbunden werden und anschließend die nächsten Zeilen mit **OR** angehängt werden. Hier fehlt der GUI schlicht die Möglichkeit eine Anweisung zu geben, dass zuerst die Spalten und dann die Zeilen abgehandelt werden sollen. Stattdessen werden bereits bei zwei Feldern insgesamt 4 von 5 in der GUI zur Verfügung stehenden Spalten beschrieben. Solche einfacheren Formulierungen sollten daher später zum Bearbeiten nur noch über das Kontextmenü mit **In SQL-Ansicht bearbeiten...** geöffnet werden. Sonst wird der Code auf den GUI-Code geändert.

Abfragen nachträglich ändern

Soll eine einmal erstelle Abfrage weiter bearbeitet werden, so wird mit einem rechten Mausklick über der Abfrage das **Kontextmenü** geöffnet und **Bearbeiten...** ausgewählt. Hier kann es manchmal dazu kommen, dass die GUI die Abfrage nicht richtig deuten kann. Manchmal passiert das schon beim Öffnen, so dass die Abfrage im SQL-Modus geöffnet wird. Es kann auch sein, dass die GUI die Abfrage öffnet, aber beim Umschalten in den SQL-Modus einen «Fehler in der SQL Syntax» findet, der gar nicht da ist. Schließlich hat die Abfrage vorher funktioniert.

Hier hilft es dann, direkt aus dem **Kontextmenü** über der Abfrage **In SQL-Ansicht bearbeiten...** zu wählen.

Abfrageerweiterungen im SQL-Modus

Wird von der grafischen Eingabe über **Ansicht → Design-Ansicht an-, ausschalten** die Design-Ansicht ausgeschaltet, so erscheint der SQL-Befehl, der bisher in der Design-Ansicht erstellt wurde. Für Neueinsteiger ist dies der beste Weg, die Standardabfragesprache für Datenbanken kennen zu lernen. Manchmal ist es auch der einzige Weg, eine Abfrage an die Datenbank abzusetzen, da die GUI die Abfrage nicht in den für die Datenbank notwendigen SQL-Befehl umwandeln kann.

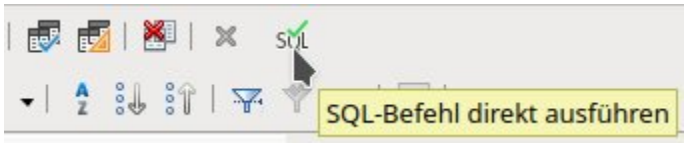
```
001 SELECT
002    *
003 FROM "Tabellenname"
```

Dies zeigt wirklich alles an, was in der Tabelle "Tabellenname" steht. Das «*» berücksichtigt sämtliche Felder der Tabelle.

```
001 SELECT
002    *
003 FROM "Tabellenname"
004 WHERE "Feldname" = 'Karl'
```

Eine deutliche Einschränkung wurde gemacht. Jetzt werden nur noch die Datensätze angezeigt, die in dem Feld "Feldname" den Begriff 'Karl' stehen haben – aber wirklich nur den Begriff, nicht z. B. 'Karl Egon'.

Manchmal sind Abfragen in Base nicht über die GUI ausführbar, da bestimmte Kommandos nicht bekannt sind. Hier hilft es dann die Design-Ansicht zu verlassen und über **Bearbeiten → SQL-Befehl direkt ausführen** den direkten Weg zur Datenbank zu wählen. Diese Methode hat allerdings den Nachteil, dass in dem angezeigten Abfrageergebnis keine Eingaben mehr möglich sind. Siehe hierzu *Eingabemöglichkeit in Abfragen*.



Die direkte Ausführung ist auch über die grafische Benutzeroberfläche erreichbar. Wie in der Abbildung zu sehen ist muss aber auch hier die Entwurfsansicht ausgeschaltet sein. Entsprechende Abfrageanweisungen sind teilweise mit SQL gekennzeichnet.

Hinweis	Wird eine SQL-Anweisung als direkte Anweisung geschrieben und ist SQL-Befehl direkt ausführen gewählt, so bleiben sämtliche Formatierungen des SQL-Kommandos erhalten. Hier können dann auch Kommentare in den SQL-Code eingefügt werden. Zeichen hierfür sind «**--** Kommentar...» für eine Kommentarzeile und «**/*** Kommentar ...***/**» für mehrere Kommentarzeilen direkt hintereinander.

Hier jetzt also die recht umfangreichen Möglichkeiten, an die Datenbank Fragen zu stellen und auf ein entsprechendes Ergebnis zu hoffen:

```
001 SELECT [{LIMIT <offset> <limit> | TOP <limit>}][ALL | DISTINCT]
```

```
002 { <Select-Formulierung> | "Tabellenname".* | * } [, ...]
003 [INTO [CACHED | TEMP  | TEXT] "neueTabelle"]
004 FROM "Tabellenliste"
005 [WHERE SQL-Expression]
006 [GROUP BY SQL-Expression [, ...]]
007 [HAVING SQL-Expression]
008 [{ UNION [ALL | DISTINCT] | {MINUS [DISTINCT] | EXCEPT [DISTINCT] } |
009 INTERSECT [DISTINCT] } Abfrageaussage]
010 [ORDER BY Ordnungs-Expression [, ...]]
011 [LIMIT <limit> [OFFSET <offset>]];
```

[{LIMIT <offset> <limit> | TOP <limit>}]: (HSQLDB**)**
[{FIRST <limit> | SKIP <limit>}]: (FIREBIRD**)**

Hiermit wird die Menge der anzuzeigenden Datensätze begrenzt.
HSQLDB:Mit **LIMIT 10 20** werden ab dem 11. Datensatz die folgenden 20 Datensätze angezeigt. Mit **TOP 10 (~~HSQLDB~~)** werden immer die ersten 10 angezeigt. Dies ist gleichbedeutend mit **LIMIT 0 10**. **LIMIT 10 0** lässt die ersten 10 Datensätze aus und zeigt alle Datensätze ab dem 11. Datensatz an.
FIREBIRD: Mit **FIRST 10** werden die ersten 10 Datensätze angezeigt. Mit **FIRST 10 SKIP 5** die Datensätze von 6 bis 15.

Den gleichen Sinn erfüllt die zum Schluss der SELECT-Bedingung erscheinende Formulierung **[LIMIT <limit> [OFFSET <offset>]] (**HSQLDB**,** FIREBIRD**)**. **LIMIT 10** zeigt lediglich 10 Datensätze an. Wird **OFFSET 20** hinzugefügt, so beginnt die Anzeige ab dem 21. Datensatz. Für die zum Schluss stehende Begrenzung ist eine Anweisung zur Sortierung (ORDER BY …) oder eine Bedingung (WHERE …) Voraussetzung.
FIREBIRD: Firebird kennt **LIMIT** eigentlich nicht. Es ist nur für die GUI implementiert worden. Hier gibt es **[ROWS <limit> TO <rownr>]**. Mit **ROWS 10** werden die ersten 10 Datensätze angezeigt. Mit **ROWS 10 TO 20** werden die Datensätze 10 bis 20 angezeigt. **rownr** muss also immer größer oder mindestens gleich **limit** sein.

Sämtliche Begrenzungen des anzuzeigenden Abfrageergebnisses sind bis einschließlich der Version LO 4.0 nur über die direkte Ausführung des SQL-Kommandos verfügbar. Erst ab LO 4.1 ist eine Limitierung ohne Sortierung oder Bedingung in der grafischen Benutzeroberfläche möglich. Dies wird sogar dann noch aufrecht erhalten, wenn in den direkten SQL-Modus umgeschaltet wurde. Die Limitierung kann in LO 4.1 in der SQL-Ansicht um den «OFFSET» ergänzt werden. Die folgende Abfrage ist also ab LO 4.1 editierbar:

```
001 SELECT * FROM "Tabelle" LIMIT 20 OFFSET 10
```
(HSQLDB, FIREBIRD)

Alle Eingaben in der Limitierung können nur direkt als Ganzzahl erfolgen. Es ist nicht möglich, die Eingaben durch eine Unterabfrage zu ersetzen, so dass z. B. fortlaufend die 5 letzten Datensätze einer Datenreihe angezeigt werden können.

[ALL | DISTINCT]

SELECT ALL ist die Standardeinstellung. Es werden alle Ergebnisse angezeigt, auf die die Bedingungen zutreffen. Beispiel:
SELECT ALL "Name" FROM "Tabellenname" gibt alle Namen an; kommt «Peter» dreifach und «Egon» vierfach in der Tabelle vor, so werden eben drei und vier Datensätze angezeigt.
SELECT DISTINCT "Name" FROM "Tabellenname" sorgt hingegen dafür, dass alle Abfrageergebnisse mit gleichem Inhalt unterdrückt werden. Hier würden also «Peter» und «Egon» nur einmal erscheinen. **DISTINCT** bezieht sich dabei auf den ganzen Datensatz, der in der Abfrage erfasst wird. Wird z. B. auch der Nachname erfasst, so unterscheiden sich die Datensätze mit «Peter Müller» und «Peter Maier». Sie werden also auch bei der Bedingung **DISTINCT** auf jeden Fall angezeigt.

<Select-Formulierung>

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

Feldnamen, Berechnungen, Zählen der gesamten Datensätze – alles mögliche Eingaben, die hier erfolgen können.

Außerdem stehen in der Felddarstellung auch verschiedene Funktionen zur Verfügung. Mit Ausnahme von **COUNT(*)** (zählt alle Datensätze) berücksichtigen die verschiedenen Funktionen keine Felder, die **NULL** sind.

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

COUNT("Name") zählt alle Felder, die einen Namen enthalten.
FIREBIRD zusätzlich: **COUNT(DISTINCT "Name")** zählt alle unterschiedlichen Namen.
MIN("Name") zeigt den ersten Namen im Alphabet. Das Ergebnis dieser Funktion ist so formatiert, wie es dem Feldinhalt entspricht. Text wird im Ergebnis Text, Ganzzahl zu Ganzzahl, Dezimalzahl zu Dezimalzahl usw.
MAX("Name") zeigt entsprechend den letzten Namen im Alphabet.
SUM("Zahl") kann nur Werte aus Zahlenfeldern addieren. Die Funktion versagt bei Datumsfeldern.

Hinweis	Berechnungen in einer Datenbank können manchmal zu erstaunlichen Ergebnissen führen. Angenommen in der Datenbank würden Zensuren einer Klassenarbeit verwaltet und mit der Berechnung sollte die Durchschnittszensur ermittelt werden. Zuerst werden die Zensurenwerte addiert. Die Summe ergibt z. B. 80. Jetzt wird durch die Zahl der Klassenarbeiten (30) dividiert. Die Abfrage ergibt 2. Dies liegt daran, dass es sich bei der Berechnung um eine Rechnung mit Feldern des Typs INTEGER handelt. Auch die Berechnung gibt dann nur Ergebnisse des Zahlentyps INTEGER aus. In der Berechnung muss mindestens ein Feld enthalten sein, das vom Zahlentyp DEZIMAL ist. Dies kann entweder durch Umwandlung mittels einer Funktion der HSQLDB erfolgen oder, einfacher, indem z. B. durch 30.0 dividiert wird. Dann erscheint eine Nachkommastelle, bei 30.00 zwei Nachkommastellen usw. Zu beachten ist hier, dass bei Berechnungen Dezimalzahlen mit dem in der englischen Schreibweise üblichen Dezimalpunkt dargestellt werden. Ein Komma ist bei Abfragen für die Trennung von Feldern reserviert. Das Ergebnis wird bei der Anzeige von Nachkommastellen in der internen HSQLDB gerundet, bei Firebird aber nicht.

Tipp

HSQLDB: Auch bei Zeitfeldern versagt die Funktion. Hier kann folgendes hilfreich sein:

```
001 SELECT
002 (SUM( HOUR("Zeit") )*3600 + SUM( MINUTE("Zeit") ))*60 +
    SUM( SECOND("Zeit") ) AS "Sekunden"
003 FROM "Tabelle"
```

Die Summe wird von den Stunden, Minuten und gegebenenfalls Sekunden gebildet. Anschließend wird so erweitert, dass alles zusammen in einer Maßeinheit, hier in Sekunden, wiedergegeben wird. Es werden also weiter nur Zahlen mit der Summenformel addiert. Mit

```
001 SELECT
002 ((SUM( HOUR("Zeit") )*3600 + SUM( MINUTE("Zeit") ))*60 + SUM(
    SECOND("Zeit") )) / 3600.0000 AS "Stunden"
003 FROM "Tabelle"
```

wird aus der Addition wieder eine Zeit in Stunden mit den Minuten und Sekunden als Nachkommastellen. Daraus kann dann über entsprechende Formatierung auch wieder eine Zeitdarstellung in einer Abfrage oder einem Formular erreicht werden.

FIREBIRD: Stunde, Minute und Sekunde müssen über die Funktion **EXTRACT()** ermittelt werden. Allerdings kennt Firebird die Funktion **DATEADD**, so dass sich hier eine Möglichkeit ergibt, direkt über die Summe von Zeiten wieder eine Zeit zu erhalten:

```
001 SELECT
002 DATEADD( SUM( DATEDIFF( SECOND FROM TIME '00:00' TO "Zeit"))
    SECOND TO TIME '00:00') AS "NeueZeit"
003 FROM "Tabelle"
```

Die Zeitdifferenz zur Zeit 00:00 (Mitternacht) wird mittels **DATEDIFF** in Sekunden berechnet. Die Sekunden werden mit **SUM** addiert. Zu der Zeit 00:00 wird jetzt mittels **DATEADD** die Anzahl der Sekunden addiert und wieder als Zeit formatiert.

AVG("Zahl") zeigt den Mittelwert der Inhalte einer Spalte. Auch diese Funktion beschränkt sich auf Zahlenfelder. Hier sollte darauf geachtet werden, dass bei Ganzzahlen auch nur ein Ergebnis in Ganzzahlen ermittelt wird. **AVG("Ganzzahl" * 1.00)** zeigt dann in der HSQLDB einen Mittelwert mit maximal 2 Nachkommastellen.
FIREBIRD zusätzlich: **AVG(DISTINCT "Zahl")** erstellt den Durchschnitt aller unterschiedlichen Zahlen.
SOME("Ja_Nein"), EVERY("Ja_Nein"): SOME zeigt bei Ja/Nein Feldern (boolschen Feldern) die Version an, die nur einige Felder erfüllen. Da ein boolsches Feld die Werte 0 und 1 in der HSQLDB wiedergibt, erfüllen nur einige (**SOME**) die Bedingung 1, aber jeder (**EVERY**) mindestens die Bedingung 0. Über eine gesamte Tabelle abgefragt wird bei SOME also immer 'Ja' erscheinen, wenn mindestens 1 Datensatz mit 'Ja' angekreuzt ist. **EVERY** wird so lange 'Nein' ergeben, bis alle Datensätze mit 'Ja' angekreuzt sind. Beispiel:

```
001 SELECT
002    "Klasse",
003    EVERY("Schwimmer")
004 FROM "Tabelle1"
005 GROUP BY "Klasse";
```

Die Klassen werden alle angezeigt. Erscheint irgendwo kein Kreuz für 'Ja', so muss auf jeden Fall eine Betreuung für das Nichtschwimmerbecken im Schwimmunterricht dabei sein, denn es gibt mindestens eine Person in der Klasse, die nicht schwimmen kann. (HSQLDB, ~~FIREBIRD~~)

Hinweis

Ja/Nein-Felder können in der HSQLDB sowohl mit **TRUE** und **FALSE** als auch mit **1** und **0** abgefragt werden. FIREBIRD hingegen verlangt zwingend nach der Angabe **TRUE** bzw. **FALSE**.

VAR_POP | VAR_SAMP | STDDEV_POP | STDDEV_SAMP sind statistische Funktionen und greifen nur bei Ganzzahl- und Dezimalzahlfeldern.
Alle Funktionen ergeben 0, wenn die Werte einer Gruppe alle gleich sind.
Die statistischen Funktionen erlauben nicht die Einschränkung von **DISTINCT**. Sie rechnen also grundsätzlich über alle Werte, die die Abfrage beinhaltet. **DISTINCT** hingegen würde Datensätze mit gleichen Werten von der Anzeige ausschließen.
VAR_POP: (SUM(expr2) – SUM(expr)2 / COUNT(expr)) / COUNT(expr). Dies gibt die Populationsabweichung eines Satzes von Zahlen zurück, nachdem die Nullwerte in diesem Satz verworfen wurden
VAR_SAMP: (SUM(expr2) – SUM(expr)2 / COUNT(expr)) / (COUNT(expr) – 1). Dies gibt die Stichprobenabweichung eines Satzes von Zahlen zurück, nachdem die Nullwerte in diesem Satz verworfen wurden.
STDDEV_POP: SQRT(VAR_POP). Berechnet die Populationsstandardabweichung und gibt die Quadratwurzel der Populationsabweichung zurück.
STDDEV_SAMP: SQRT(VAR_SAMP). Berechnet die kumulative Stichproben-Standardabweichung und gibt die Quadratwurzel der Probenvarianz zurück.

[AS] "anzuzeigende Bezeichnung": Den Feldern kann in der Abfrage eine andere Bezeichnung (Alias) gegeben werden. **AS** ist zur Einführung eines Alias nicht erforderlich.

"Tabellenname".* | * [, ...]
Jedes anzuzeigende Feld kann mit seinem Feldnamen, getrennt durch Komma, angegeben werden. Werden Felder aus mehreren Tabellen in der Abfrage aufgeführt, so ist zusätzlich eine Kombination mit dem Tabellennamen notwendig: **"Tabellenname"."Feldname"**.

Statt einer ausführlichen Formulierung kann auch der gesamte Inhalt einer Tabelle angezeigt werden. Hierfür steht das Symbol «*».

[INTO [CACHED | TEMP | TEXT] "neueTabelle"] (HSQLDB**)**
Das Ergebnis dieser Abfrage soll direkt in eine **neue** Tabelle geschrieben werden. Die neue Tabelle wird hier benannt. Die Definition der Feldeigenschaften der neuen Tabelle wird dabei aus der Definition der Felder, die in der Abfrage enthalten sind, erstellt.
Das Schreiben in eine Tabelle funktioniert nicht vom Abfrageeditor aus, da dieser nur anzeigbare Ergebnisse liefert. Hier muss die Eingabe über **Extras → SQL** erfolgen. Die Tabelle, die entsteht, ist anschließend erst einmal nicht editierbar, da ein Primärschlüsselfeld fehlt.

Standardmäßig wird der Inhalt in eine Tabelle des Typs **CACHED** geschrieben, der dem Typ der anderen Tabellen entspricht. Der Typ **TEMP** ist, wie bei den Tabellen beschrieben, nur begrenzt in Base nutzbar. Der Typ **TEXT** hingegen exportiert den Inhalt der Abfrage in eine kommaseparierte Textdatei, die auch von Tabellenkalkulationsprogrammen eingelesen werden kann. Die Datei liegt anschließend in dem Verzeichnis, in dem auch die *.odb-Datei der Datenbank liegt. Außerdem wird die Datei anschließend als Ansicht in dem Tabellenordner der Datenbank angezeigt. Hierbei ist allerdings zu beachten, dass ein solcher Export standardmäßig im ASCII- Zeichensatz erfolgt, Sonderzeichen also nicht berücksichtigt werden.

Um Sonderzeichen anzeigen zu können, muss der Zeichensatz geändert werden. Dies kann entweder durch einen SQL-Befehl geschehen, der den Zeichensatz für die aktuelle Sitzung einstellt, oder aber durch eine dauerhafte Einstellung in der Datenbankdatei.

Die vorübergehende Änderung ist durch die Eingabe von
SET PROPERTY "textdb.encoding" 'UTF-8';
unter **Extras → SQL** möglich. Hier kann natürlich auch entsprechend 'ansi' gewählt werden.

Zur dauerhaften Änderung des Zeichensatzes muss die Datenbankdatei entpackt werden. Das in dieser Datei liegende Verzeichnis **database** enthält eine Datei **properties**. Diese muss um einen Eintrag erweitert werden: **textdb.encoding=UTF-8** für Linux-Systeme oder **textdb.encoding=ansi** für Windows-Systeme. Wie der Zugriff auf diese Einstellungen möglich ist, wird im Anhang im Kapitel zur «Datenbankreparatur» beschrieben.

FROM <Tabellenliste>

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

Die Tabellen, aus denen die Daten zusammengesucht werden sollen, werden in der Regel durch Komma getrennt aufgeführt. Die Beziehung der Tabellen zueinander wird anschließend mit dem Schlüsselwort **WHERE** definiert.

Werden die Tabellen durch einen **JOIN** miteinander verbunden, so wird die Beziehung der Tabellen zueinander direkt nach der jeweils folgenden Tabelle mit dem Begriff **ON** beginnend definiert.

Ein einfacher **JOIN** bewirkt, dass nur die Datensätze angezeigt werden, auf die die Bedingung in beiden Tabellen zutrifft. Beispiel:

```
001 SELECT
002    "Tabelle1"."Name",
003    "Tabelle2"."Klasse"
004 FROM "Tabelle1",
005    "Tabelle2"
006 WHERE "Tabelle1"."KlasseID" = "Tabelle2"."ID"
```

entspricht von der Wirkung her

```
001 SELECT
002    "Tabelle1"."Name",
003    "Tabelle2"."Klasse"
004 FROM "Tabelle1"
005    JOIN "Tabelle2"
006       ON "Tabelle1"."KlasseID" = "Tabelle2"."ID"
```

Es werden hier die Namen und die dazugehörigen Klassen aufgelistet. Fehlt zu einem Namen eine Klasse, so wird der Name nicht aufgelistet. Fehlen zu einer Klasse Namen, so werden diese ebenfalls nicht aufgelistet. Der Zusatz **INNER** bewirkt hierbei keine Änderung.

```
001 SELECT
002    "Tabelle1"."Name",
003    "Tabelle2"."Klasse"
004 FROM "Tabelle1"
005    LEFT JOIN "Tabelle2"
006       ON "Tabelle1"."KlasseID" = "Tabelle2"."ID"
```

Bei dem Zusatz **LEFT** würden auf jeden Fall alle Inhalte von "Name" aus "Tabelle1" angezeigt – auch die, zu denen keine "Klasse" existiert. Beim Zusatz **RIGHT** hingegen würden alle Klassen angezeigt – auch die, zu denen kein Name existiert. Der Zusatz **OUTER** muss hier nicht unbedingt mit angegeben werden.

Zusätzliche Filter einzelner Tabellen sollten als **Unterabfrage** eingefügt werden:

```
001 SELECT
002    "Tabelle1".*,
003    "Tabelle2".*,
004    "Tabelle3".*,
005 FROM "Tabelle1"
006    LEFT JOIN
007    (SELECT * FROM "Tabelle2" WHERE "Name" = 'BigBoss') AS "Tabelle2"
008       ON "Tabelle1"."Tab2ID" = "Tabelle2"."ID"
009    LEFT JOIN
010    (SELECT * FROM "Tabelle3" WHERE "Ort" = 'Hintertupfingen')
          AS "Tabelle3"
011       ON "Tabelle2"."Tab3ID" = "Tabelle3"."ID"
```

Von "Tabelle1" werden **alle** Datensätze angezeigt. Aus "Tabelle2" werden nur die Datensätze angezeigt, die im Feld "Name" 'BigBoss' stehen haben. "Tabelle3" wird an "Tabelle2" angehängt. Es werden also in "Tabelle3" nur die Datensätze angezeigt, die mit Datensätzen von "Tabelle2" zu verbinden sind, in denen im Feld "Name" 'BigBoss' steht. Zusätzlich wird

die Anzeige bei "Tabelle3" auf die Datensätze begrenzt, die den "Ort" 'Hintertupfingen' enthalten.

Wird statt einer Unterabfrage die Bedingung direkt an die Beziehungsdefinition (mit **WHERE** oder **AND**) angehängt, so werden dadurch alle Datensätze gefiltert und gegebenenfalls nicht alle Datensätze von "Tabelle1" angezeigt.

```
001 SELECT
002    "Tabelle1"."Spieler1",
003    "Tabelle2"."Spieler2"
004 FROM "Tabelle1" AS "Tabelle1"
005    CROSS JOIN "Tabelle2" AS "Tabelle2"
006 WHERE "Tabelle1"."Spieler1" <> "Tabelle2"."Spieler2"
```

Beim **CROSS JOIN** müssen auf jeden Fall die Tabellen mit einem Aliasnamen versehen werden, wobei das Hinzufügen des Begriffes **AS** nicht unbedingt notwendig ist. Es werden einfach alle Datensätze aus der ersten Tabelle mit allen Datensätzen der zweiten Tabelle gekoppelt. So ergibt die obige Abfrage alle möglichen Paarungen aus der ersten Tabelle mit denen aus der zweiten Tabelle mit Ausnahme der Paarungen, bei denen es sich um gleiche Spieler handelt. Die Bedingung darf beim **CROSS JOIN** allerdings **keine Verknüpfung der Tabellen mit ON** enthalten. Stattdessen können unter **WHERE** Bedingungen eingegeben werden. Würde hier die Bedingung genauso formuliert wie beim einfachen JOIN, so wäre das Ergebnis gleich:

```
001 SELECT
002    "Tabelle1"."Name",
003    "Tabelle2"."Klasse"
004 FROM "Tabelle1"
005    JOIN "Tabelle2"
006 ON "Tabelle1"."KlasseID" = "Tabelle2"."ID"
```

liefert das gleiche Ergebnis wie

```
001 SELECT
002    "Tabelle1"."Name",
003    "Tabelle2"."Klasse"
004 FROM "Tabelle1" AS "Tabelle1"
005    CROSS JOIN "Tabelle2" AS "Tabelle2"
006 WHERE "Tabelle1"."KlasseID" = "Tabelle2"."ID"
```

[WHERE SQL-Expression]

Die Standardeinleitung, um Bedingungen für eine genauere Filterung der Daten zu formulieren. Hier werden in der Regel auch die Beziehungen der Tabellen zueinander definiert, sofern die Tabellen nicht mit JOIN verbunden sind.

[GROUP BY SQL-Expression [, …]]

Wenn Felder mit einer bestimmten Funktion bearbeitet werden (z. B. **COUNT, SUM** …), so sind alle Felder, die nicht mit einer Funktion bearbeitet werden, aber angezeigt werden sollen, mit **GROUP BY** zu einer Gruppe zusammen zu fassen.
Beispiel:

```
001 SELECT
002    "Name",
003    SUM("Einnahme"-"Ausgabe") AS "Saldo"
004 FROM "Tabelle1"
005 GROUP BY "Name";
```

Datensätze mit gleichen Namen werden jetzt aufsummiert. Im Ergebnis wird jeweils **Einnahme − Ausgabe** ermittelt und darüber die Summe, die jede Person erzielt hat, aufgelistet. Das Feld wird unter dem Namen **Saldo** dargestellt.

[HAVING SQL-Expression]

Die **HAVING**-Formulierung ähnelt sehr der **WHERE**-Formulierung. Sie kann für Bedingungen eingesetzt werden, die mit Hilfe von Funktionen wie MIN, MAX formuliert werden. **Nur**

HAVING ist dafür geeignet, in den Bedingungen auch Berechnungen durchzuführen. HAVING erscheint dabei nach einer eventuell vorhandenen GROUP BY – Formulierung. Beispiel:

```
001 SELECT
002    "Name",
003    "Laufzeit"
004 FROM "Tabelle1"
005 GROUP BY "Name",
006    "Laufzeit"
007 HAVING MIN("Laufzeit") < '00:40:00';
```

Es werden alle Namen und Laufzeiten aufgelistet, bei denen die Laufzeit weniger als 40 Minuten beträgt.

[SQL Expression]

SQL-Ausdrücke werden nach dem folgenden Schema miteinander verbunden:

```
001 [NOT] Bedingung [{ OR | AND } Bedingung]
```

Beispiel:

```
001 SELECT
002    *
003 FROM "Tabellenname"
004 WHERE
005    NOT "Rückgabedatum" IS NULL
006    AND "LeserID" = 2;
```

Aus der Tabelle werden die Datensätze ausgelesen, bei denen ein "Rückgabedatum" eingetragen wurde und die "LeserID" gleich 2 ist. Das würde in der Praxis bedeuten, dass alle Medien, die eine bestimmte Person ausgeliehen und wieder zurückgegeben hat, damit ermittelt werden könnten. Die Bedingungen sind nur durch **AND** miteinander verbunden. Das **NOT** bezieht sich rein auf die erste Bedingung.

```
001 SELECT
002    *
003 FROM "Tabellenname"
004 WHERE NOT ("Rückgabedatum" IS NULL AND "LeserID" = 2);
```

Wird eine Klammer um die Bedingung gesetzt und **NOT** steht außerhalb der Klammer, so werden genau die Datensätze angezeigt, die die in den Klammern stehenden Bedingungen zusammen komplett nicht erfüllen. Das wären alle Datensätze mit Ausnahme derer, die "LeserID" mit der Nummer 2 noch nicht zurückgegeben hat.

[SQL Expression]: Bedingungen

```
001 { Wert [|| Wert]
```

Ein Wert kann einzeln oder mit mehreren Werten zusammen über zwei senkrechte Striche **||** kombiniert werden. Dies gilt dann natürlich auch für Feldinhalte.

```
001 SELECT
002    "Nachname" || ', ' || "Vorname" AS "Name"
003 FROM "Tabellenname"
```

Die Inhalte aus den Feldern "Nachname" und "Vorname" werden in einem Feld "Name" gemeinsam angezeigt. Dabei wird ein Komma und eine Leertaste zwischen "Nachname" und "Vorname" eingefügt.

```
001 | Wert { = | < | <= | > | >= | <> | != } Wert
```

Die Zeichen entsprechend den aus der Mathematik bekannten Operatoren:
{ Gleich | kleiner als | kleiner oder gleich | größer als | größer oder gleich | nicht gleich | nicht gleich }
Firebird kennt hier noch die folgenden Alternativen, die allerdings nur mit direkten SQL funktionieren:

```
001 { ~= | ^= | !> | ~> | ^> | !< | ~< | ^< }
```

{ Nicht gleich | nicht gleich | nicht größer als | nicht größer als | nicht größer als | nicht kleiner als | nicht kleiner als | nicht kleiner als }

```
001 | Wert IS [NOT] NULL
```

Das entsprechende Feld hat keinen Inhalt, ist auch nicht beschrieben worden. Dies kann in der GUI nicht unbedingt beurteilt werden, denn ein leeres Textfeld bedeutet noch nicht, dass das Feld völlig ohne Inhalt ist. Die Standardeinstellung von Base ist aber so, dass leere Felder in der Datenbank auf **NULL** gesetzt werden.

```
001 | EXISTS(Abfrageaussage)
```

Beispiel:

```
001 SELECT
002    "Name"
003 FROM "Tabelle1"
004 WHERE EXISTS
005    (SELECT
006       "Vorname"
007    FROM "Tabelle2"
008    WHERE "Tabelle2"."Vorname" = "Tabelle1"."Name")
```

Es werden die Namen aus Tabelle1 aufgeführt, die als Vornamen in Tabelle2 verzeichnet sind.

```
001 | SINGULAR(Abfrageaussage) (~~HSQLDB~~, FIREBIRD)
```

Beispiel:

```
002 SELECT
003    "Name"
004 FROM "Tabelle1"
005 WHERE SINGULAR
006    (SELECT
007       "Vorname"
008    FROM "Tabelle2"
009    WHERE "Tabelle2"."Vorname" = "Tabelle1"."Name")
```

Es werden die Namen aus Tabelle1 aufgeführt, die als Vornamen in Tabelle2 nur genau einmal verzeichnet sind.

```
001 | Wert BETWEEN Wert AND Wert
```

BETWEEN Wert1 AND Wert2 gibt alle Werte ab Wert1 bis einschließlich Wert2 wieder. Werden hier Buchstaben als Werte eingesetzt, so wird die alphabetische Sortierung angenommen, wobei Kleinbuchstaben und Großbuchstaben die gleichen Werte haben.

```
001 SELECT
002    "Name"
003 FROM "Tabellenname"
004 WHERE "Name" BETWEEN 'A' AND 'E';
```

Diese Abfrage gibt alle Namen wieder, die mit A, B, C und D beginnen (ggf. auch mit entsprechendem Kleinbuchstaben). Da als unterer Begrenzung E gesetzt wurde, sind alle Namen mit E nicht mehr in der Auswahl enthalten. Der Buchstabe E würde in einer Sortierung ganz am Anfang der Namen mit E stehen.

```
001 | Wert [NOT] IN ( {Wert [, ...] | Abfrageaussage } )
```

Hier wird entweder eine Liste von Werten oder eine Abfrage eingesetzt. Die Bedingung ist erfüllt, wenn der Wert in der Werteliste bzw. im Abfrageergebnis enthalten ist.

```
001 | Wert [NOT] LIKE Wert [ESCAPE] Wert }
```

Der **LIKE**-Operator ist derjenige, der in vielen einfachen Suchfunktionen benötigt wird. Die Angabe der Werte erfolgt hier nach folgendem Muster:
'%' steht für beliebig viele, ggf. auch 0 Zeichen,
'_' ersetzt genau ein Zeichen.

Um nach '%' oder '_' selbst zu suchen müssen die Zeichen direkt nach einem zweiten Zeichen auftauchen, das nach **ESCAPE** definiert wird.

```
001 SELECT
002    "Name"
003 FROM "Tabellenname"
004 WHERE "Name" LIKE '\_%' ESCAPE '\'
```

Diese Abfrage zeigt alle Namen auf, die mit einem Unterstrich beginnen. Als **ESCAPE**-Zeichen ist hier '\' definiert worden.

```
001 | Wert [NOT] STARTING WITH Wert }(HSQLDB, FIREBIRD)
```

Hier wird nur nach dem Beginn des Strings in einem Feld gesucht.
STARTING WITH 'Li' ergibt das Gleiche wie **LIKE 'Li%'**.

```
001 | Wert [NOT] CONTAINING Wert }(HSQLDB, FIREBIRD)
```

Ein Feld wird mit einem Wert (String) verglichen. Enthält das Feld den String oder die Zahlenkombination, so wird das Feld wiedergegeben.
CONTAINING 'Li' ergibt das Gleiche wie **LIKE '%Li%'**.

```
001 | Wert IS [NOT] DISTINCT FROM Wert }(HSQLDB, FIREBIRD)
```

Ein Wert ist dann **NOT DISTINCT** von einem anderen Wert, wenn er gleich ist oder wenn beide Inhalte **NULL** sind.
"Feld1" NOT DISTINCT "Feld2" ergibt das Gleiche wie **"Feld1" = "Feld2" OR ("Feld1" IS NULL AND "Feld2" IS NULL)**

```
001 | string-expression [NOT] SIMILAR TO <pattern> [ESCAPE <escape-char>]
    (HSQLDB, FIREBIRD)
```

```
<pattern>     ::= ein regulärer SQL-Ausdruck
<escape-char> ::= ein einzelnen Zeichen
```

SIMILAR TO[1] vergleicht eine Zeichenkette, also auch den Inhalt eines Tabellenfeldes, mit einem in SQL definierten regulären Ausdruck. Anders als in einigen anderen Sprachen muss das Muster mit der gesamten Zeichenfolge übereinstimmen, um erfolgreich zu sein - die Übereinstimmung eines Teilstrings reicht nicht aus. Wenn ein Operand **NULL** ist, ist das Ergebnis **NULL**. Andernfalls ist das Ergebnis **TRUE** oder **FALSE**.

Die folgende Syntax definiert das Format des regulären SQL-Ausdrucks. Es handelt sich um eine vollständige Top-Down-Definition. Die Definition ist sehr formell, ziemlich lang und wahrscheinlich perfekt geeignet, um alle zu entmutigen, die nicht bereits einige Erfahrungen mit regulären Ausdrücken (oder mit sehr formalen, ziemlich langen Top-down-Definitionen) haben. Deshalb wird in einem weiteren Abschnitt die Erstellung regulärer Ausdrücke an Beispielen erklärt.

Syntax des regulären SQL-Ausdrucks

```
<regular expression>      ::= <regular term> ['|' <regular term> ...]
<regular term>            ::= <regular factor> ...
<regular factor>          ::= <regular primary> [<quantifier>]
<quantifier>              ::= ?
                            | *
                            | +
                            | '{' <m> [,[<n>]] '}'
<m>, <n>                  ::= Integer ohne Vorzeichen, mit <m> <= <n>, wenn beide
                              angegeben werden
<regular primary>         ::= <character>
                            | <character class>
                            | %
                            | (<regular expression>)
<character>               ::= <escaped character>
                            | <non-escaped character>
<escaped character>       ::= <escape-char> <special character>
```

1 Übersetzung aus der Firebird 2.5 Language Reference siehe:*http://www.firebirdsql.org/en/documentation/*

```
                            | <escape-char> <escape-char>
<special character>     ::= eines der Zeichen []()|^-+*%_?{
<non-escaped character>    ::= Eines der Zeichen, das nicht ein <special character>
                            und nicht gleich <escape-char> ist, falls definiert
<character class>          ::= '_'
                            | '[' <member> ... ']'
                            | '[^' <non-member> ... ']'
                            | '[' <member> ... '^' <non-member> ... ']'
<member>, <non-member>     ::= <character>
                            | <range>
                            | <predefined class>
<range>                    ::= <character>-<character>
<predefined class>         ::= '[:' <predefined class name> ':]'
<predefined class name>    ::= ALPHA | UPPER | LOWER | DIGIT | ALNUM | SPACE |
                            WHITESPACE
```

Erstellen regulärer Ausdrücke

Zeichen

Innerhalb regulärer Ausdrücke repräsentieren sich die meisten Zeichen selbst. Die einzigen Ausnahmen sind die folgenden Sonderzeichen:

```
[ ] ( ) | ^ - + * % _ ? {
```

...und das Escape-Zeichen, wenn es definiert ist.

Ein regulärer Ausdruck, der keine Sonder- oder Escape-Zeichen enthält, stimmt nur mit identischen Strings überein (abhängig von dem verwendeten Zeichensatz). Das heißt, es funktioniert genau wie der "=" - Operator:

```
'Apple' similar to 'Apple'  -- true
'Apples' similar to 'Apple' -- false
'Apple' similar to 'Apples' -- false
'APPLE' similar to 'Apple'  -- abhängig vom verwendeten Zeichensatz
```

Wildcards

Die bekannten SQL-Wildchards entsprechen einem einzelnen Zeichen (_)und einem String jeder beliebigen Länge (%):

```
'Birne' similar to 'B_rne'   -- true
'Birne' similar to 'B_ne'    -- false
'Birne' similar to 'B%ne'    -- true
'Birne' similar to 'Bir%ne%' -- true
'Birne' similar to 'Birr%ne' -- false
```

% kann auch für einen leeren String stehen.

Zeichenklassen

Ein Sammlung von Zeichen, die in Klammern eingeschlossen sind, definiert eine Zeichenklasse. Ein Zeichen in dem String entspricht einer Klasse im Muster, wenn das Zeichen in der Klasse enthalten ist:

```
'Citroen' similar to 'Cit[arju]oen'  -- true
'Citroen' similar to 'Ci[tr]oen'     -- false
'Citroen' similar to 'Ci[tr][tr]oen' -- true
```

Wie aus der zweiten Zeile ersichtlich ist, entspricht die Klasse nur einem einzelnen Zeichen, nicht mehreren Zeichen hintereinander.

Innerhalb einer Klassendefinition definieren zwei Zeichen, die durch einen Bindestrich verbunden sind, einen Bereich. Ein Bereich umfasst die beiden Endpunkte und alle Zeichen, die zwischen ihnen in der aktiven Sortierung liegen. Bereiche können an beliebiger Stelle in der Klassendefinition platziert werden, ohne dass spezielle Trennzeichen vorhanden sind, um sie von den anderen Elementen getrennt zu halten.

```
'Datte' similar to 'Dat[q-u]e'        -- true
'Datte' similar to 'Dat[abq-uy]e'     -- true
'Datte' similar to 'Dat[bcg-km-pwz]e' -- false
```

Die folgenden vordefinierten Zeichenklassen können auch in einer Klassendefinition verwendet werden:

[:ALPHA:]
Buchstaben a..z und A..Z. Abhängig vom Zeichensatz der Datenbank enthält dies auch entsprechende Sonderzeichen.
[:DIGIT:]
Dezimalziffern 0..9.
[:ALNUM:]
Zusammenschluss von [:ALPHA:] und [:DIGIT:].
[:UPPER:]
Großgeschriebene Buchstaben A..Z. Abhängig vom Zeichensatz auch Kleinbuchstaben wie z.B. 'ß'.
[:LOWER:]
Kleingeschriebene Buchstaben a..z. Abhängig vom Zeichensatz gegebenenfalls auch Großbuchstaben.
[:SPACE:]
Leerzeichen (ASCII 32).
[:WHITESPACE:]
Vertikaler Tabulator (ASCII 9), Zeilenvorschub (ASCII 10), horizontaler Tabulator (ASCII 11), Seitenvorschub (ASCII 12), Wagenrücklauf (ASCII 13) und Leerzeichen (ASCII 32).

Der Einschluss einer vordefinierten Klasse hat die gleiche Wirkung wie die Aufzählung aller ihrer Mitglieder. Vordefinierte Klassen sind nur innerhalb von Klassendefinitionen zulässig. Wenn nur mit einer vordefinierten Klasse verglichen werden soll, so muss ein zusätzliches Paar von Klammern um die vordefinierte Klasse gelegt werden:

```
'Erdbeere' similar to 'Erd[[:ALNUM:]]eere'    -- true
'Erdbeere' similar to 'Erd[[:DIGIT:]]eere'    -- false
'Erdbeere' similar to 'Erd[a[:SPACE:]b]eere'  -- true
'Erdbeere' similar to [[:ALPHA:]]               -- false
'E' similar to [[:ALPHA:]]                    -- true
```

Wenn eine Klassendefinition mit einem Caret '^' beginnt, darf alles, was folgt, nicht im String an der Stelle existieren:

```
'Framboise' similar to 'Fra[^ck-p]boise'       -- false
'Framboise' similar to 'Fr[^a][^a]boise'       -- false
'Framboise' similar to 'Fra[^[:DIGIT:]]boise'  -- true
```

Wenn das Caret nicht am Anfang der Sequenz platziert wird, enthält die Klasse alles vor dem Caret, mit Ausnahme der Elemente, die auch nach dem Caret auftreten:

```
'Grapefruit' similar to 'Grap[a-m^f-i]fruit'  -- true
'Grapefruit' similar to 'Grap[abc^xyz]fruit'  -- false
'Grapefruit' similar to 'Grap[abc^de]fruit'   -- false
'Grapefruit' similar to 'Grap[abe^de]fruit'   -- false
'3' similar to '[[:DIGIT:]^4-8]'              -- true
'6' similar to '[[:DIGIT:]^4-8]'              -- false
```

Schließlich ist die bereits erwähnte Wildcard "_" eine eigenständige Zeichenklasse, die mit jedem einzelnen Zeichen übereinstimmt.

Quantoren

Ein Fragezeichen unmittelbar nach einem Zeichen oder einer Klasse weist darauf hin, dass das vorhergehende Element 0 oder 1 Mal auftreten kann:

```
'Hallon' similar to 'Hal?on'             -- false
'Hallon' similar to 'Hal?lon'            -- true
'Hallon' similar to 'Halll?on'           -- true
'Hallon' similar to 'Hallll?on'          -- false
'Hallon' similar to 'Halx?lon'           -- true
'Hallon' similar to 'H[a-c]?llon[x-z]?'  -- true
```

Ein Stern '*', der unmittelbar einem Zeichen oder einer Klasse folgt, zeigt an, dass das vorhergehende Element 0 oder mehrere Male auftreten kann:

```
'Icaque' similar to 'Ica*que'       -- true
```

```
'Icaque' similar to 'Icar*que'      -- true
'Icaque' similar to 'I[a-c]*que'    -- true
'Icaque' similar to '_*'            -- true
'Icaque' similar to '[[:ALPHA:]]*'  -- true
'Icaque' similar to 'Ica[xyz]*e'    -- false
```

Ein Pluszeichen unmittelbar nach einem Zeichen oder einer Klasse weist darauf hin, dass das vorhergehende Element 1 oder mehrere Male auftreten muss:

```
'Jujube' similar to 'Ju_+'                -- true
'Jujube' similar to 'Ju+jube'             -- true
'Jujube' similar to 'Jujuber+'            -- false
'Jujube' similar to 'J[jux]+be'           -- true
'Jujube' similar to 'J[[:DIGIT:]]+ujube'  -- false
```

Wenn einem Zeichen oder einer Klasse eine in geschweiften Klammern eingeschlossene Zahl folgt, muss die Überprüfung genau so oft wiederholt werden:

```
'Kiwi' similar to 'Ki{2}wi'     -- false
'Kiwi' similar to 'K[ipw]{2}i'  -- true
'Kiwi' similar to 'K[ipw]{2}'   -- false
'Kiwi' similar to 'K[ipw]{3}'   -- true
```

Wenn der Zahl ein Komma folgt, muss das Element mindestens so oft wiederholt werden:

```
'Limone' similar to 'Li{2,}mone'     -- false
'Limone' similar to 'Li{1,}mone'     -- true
'Limone' similar to 'Li[nezom]{2,}'  -- true
```

Wenn die geschweiften Klammern zwei Zahlen enthalten, die durch ein Komma getrennt sind, und die zweite Zahl nicht kleiner als die erste ist, dann muss das Element mindestens so oft wie die erste Zahl und höchstens so oft die zweite Zahl wiederholt werden:

```
'Mandarijn' similar to 'M[a-p]{2,5}rijn'   -- true
'Mandarijn' similar to 'M[a-p]{2,3}rijn'   -- false
'Mandarijn' similar to 'M[a-p]{2,3}arijn'  -- true
```

Die Quantoren ?, * und + entsprechen jeweils Kurzformen von {0,1}, {0,} und {1,}.

ODER-Ausdrücke

Reguläre Ausdrücke können mit dem '|'-Operator erzeugt werden. Eine Übereinstimmung ist dann gegeben, wenn der Argumentstring mit mindestens einem der Begriffe übereinstimmt:

```
'Nektarin' similar to 'Nek|tarin'         -- false
'Nektarin' similar to 'Nektarin|Persika'  -- true
'Nektarin' similar to 'M_+|N_+|P_+'       -- true
```

Unterausdrücke

Ein oder mehrere Teile des regulären Ausdrucks können in Unterausdrücken (auch als Untermuster bezeichnet) gruppiert werden, indem sie zwischen Klammern platziert werden. Ein Unterausdruck ist ein regulärer Ausdruck in seinem eigenen Bereich. Er kann alle Elemente enthalten, die in einem regulären Ausdruck erlaubt sind. Es können auch Quantifizierer hinzugefügt werden:

```
'Orange' similar to 'O(ra|ri|ro)nge'       -- true
'Orange' similar to 'O(r[a-e])+nge'        -- true
'Orange' similar to 'O(ra){2,4}nge'        -- false
'Orange' similar to 'O(r(an|in)g|rong)?e'  -- true
```

Maskieren von Sonderzeichen

Um einem Zeichen zu entsprechen, das in regulären Ausdrücken enthalten ist, muss dieses Zeichen maskiert werden. Es gibt kein Standard-Zeichen zur Maskierung (Escape-Zeichen). Das Escape-Zeichen legt der Benutzer bei Bedarf fest:

```
'Peer (Poire)' similar to 'P[^ ]+ \(P[^ ]+\)' escape '\'  -- true
'Pera [Pear]' similar to 'P[^ ]+ #[P[^ ]+#]' escape '#'   -- true
'Päron-Äppledryck' similar to 'P%$-Ä%' escape '$'         -- true
'Pärondryck' similar to 'P%--Ä%' escape '-'               -- false
```

Die letzte Zeile zeigt, dass das Escape-Zeichen bei Bedarf auch zur Maskierung des eigenen Zeichens genutzt werden kann.

Hinweis	Datumswerte können bei der internen Datenbank im Format 'YYYY-MM-DD' angegeben werden. Bei externen Datenbanken kann es aber passieren, dass das Datum unter diesen Umständen nicht korrekt gelesen werden kann. Hier kann die ältere Version {D 'YYYY-MM-DD'} oder die neuere Version {d 'YYYY-MM-DD'} zum Erfolg führen. Entsprechende Formate existieren auch für Zeitfelder und für Datums-Zeit-Felder: {t 'HH:MI:SS[.SS]'} bzw. {ts 'YYYY-MM-DD HH:MI:SS[.SS]'}. Entsprechend sind die Buchstaben «D» und «T» sowie die Kombination «TS» reservierte Abkürzungen, die nicht an anderer Stelle alleinstehend benutzt werden können. Eine Parameterabfrage mit dem Parameter «:D» wird z.B. nicht funktionieren. Siehe hierzu auch die Hilfe von LO zum Stichwort «Abfrageentwurf».

[SQL Expression]: Werte

```
001 [+ | -] { Ausdruck [{ + | - | * | / | || } Ausdruck]
```

Vorzeichen vor den Werten sind möglich. Die Addition, Subtraktion, Multiplikation, Division und Verkettung von Ausdrücken ist erlaubt. Beispiel für eine Verkettung:

```
001 SELECT
002    "Nachname"||', '||"Vorname"
003 FROM "Tabelle"
```

Auf diese Art und Weise werden Datensätze in der Abfrage als ein Feld ausgegeben, in der "Nachname, Vorname" steht. Die Verkettung erlaubt also jeden der weiter unten genannten Ausdrücke.

```
001 | ( Bedingung )
```

Siehe hierzu den vorhergehenden Abschnitt

```
001 | Funktion ( [Parameter] [,...] )
```

Siehe hierzu im Anhang das Kapitel «Eingebaute Funktionen und abgespeicherte Prozeduren».

Die folgenden Abfragen werden auch als Unterabfragen (Subselects) bezeichnet.

```
001 | Abfrageaussage, die nur genau einen Wert ergibt
```

Da ein Datensatz für jedes Feld nur einen Wert darstellen kann, kann auch nur die Abfrage komplett angezeigt werden, die genau einen Wert ergibt.

```
001 | {ANY|ALL} (Abfrageaussage, die den Inhalt einer ganzen Spalte
    wiedergibt)
```

Manchmal gibt es Bedingungen, bei denen ein Ausdruck mit einer ganzen Gruppe von Werten verglichen wird. Zusammen mit **ANY** bedeutet das, dass der Ausdruck mindestens einmal in der Gruppe vorkommen muss. Dies ließe sich auch mit der **IN**-Bedingung beschreiben. **= ANY** ergibt das gleiche Ergebnis wie **IN**, funktioniert aber nur mit einer Unterabfrage. Statt **ANY** kann in Firebird alternativ auch **SOME** genutzt werden.

Zusammen mit **ALL** bedeutet dies, dass alle Werte der Gruppe dem einen Ausdruck entsprechen müssen.

[SQL Expression]: Ausdrücke

```
001 { 'Text' | Ganzzahl | Fließkommazahl
001 | ["Tabelle".]"Feld" | TRUE | FALSE | NULL }
```

Als Grundlage dienen Werte, die, abhängig vom Quellformat, mit unterschiedlichen Ausdrücken angegeben werden. Wird nach Textinhalten gesucht, so ist der Inhalt in Hochkommata zu setzen. Ganzzahlen werden ohne Hochkommata geschrieben, ebenso Fließkommazahlen (statt Komma ist in SQL direkt der Dezimalpunkt zu setzen).

Felder stehen für die Werte, die sich in den Feldern einer Tabelle befinden. Meist werden entweder Felder miteinander verglichen oder Felder mit Werten. In SQL werden die Feldbezeichnungen besser in doppelte Anführungsstriche gesetzt, da sonst eventuell Feldbezeichnungen nicht richtig erkannt werden. Üblicherweise geht SQL ohne doppelte Anführungsstriche davon aus, dass alles ohne Sonderzeichen, in einem Wort und mit Großbuchstaben geschrieben ist. Sind mehrere Tabellen in der Abfrage enthalten, so ist neben dem Feld auch die Tabelle, vom Feld getrennt durch einen Punkt, aufzuführen.
TRUE und **FALSE** stammen üblicherweise von Ja/Nein-Feldern.
NULL bedeutet, dass nichts angegeben wurde. Es ist nicht gleichbedeutend mit 0, sondern eher mit Leer.

UNION [ALL | DISTINCT] Abfrageaussage **SQL** ~~GUI~~

Hiermit werden Abfragen so verknüpft, dass der Inhalt der 2. Abfrage unter die erste Abfrage geschrieben wird. Dazu müssen alle Felder der beiden Abfragen vom Typ her übereinstimmen. Diese Verknüpfung von mehreren Abfragen funktioniert nur über die direkte Ausführung des SQL-Kommandos.

```
001 SELECT
002    "Vorname"
003 FROM "Tabelle1"
004    UNION DISTINCT
005    SELECT
006       "Vorname"
007    FROM "Tabelle2";
```

Diese Abfrage liefert alle Vornamen aus Tabelle1 und Tabelle2; der Zusatz **DISTINCT** zeigt an, dass keine doppelten Vornamen ausgegeben werden. **DISTINCT** ist in diesem Zusammenhang die Standardeinstellung. Die Vornamen sind dabei standardmäßig nach dem Alphabet aufsteigend sortiert. Mit **ALL** werden einfach alle Vornamen aus Tabelle1 und Tabelle2 angezeigt. Sie sind jetzt standardmäßig nach dem ersten Feld der Anzeige, hier also "Vorname", sortiert.

Mit Hilfe dieser Abfragetechnik ist es auch möglich, Werte einer Datenzeile z.B. für eine Liste untereinander in einer Spalte anzuordnen. Angenommen es existiert eine Tabelle "Ware", in der der "Verkaufspreis" sowie ein "Rabattpreis_1" und ein "Rabattpreis_2" enthalten sind. Daraus soll der Inhalt für ein Kombinationsfeld erstellt werden, das genau diese Preise untereinander auflistet:

```
001 SELECT
002    "Verkaufspreis"
003 FROM "Ware" WHERE "Ware_ID" = 1
004    UNION
005    SELECT
006       "Rabattpreis_1"
007    FROM "Ware" WHERE "Ware_ID" = 1
008    UNION
009    SELECT
010       "Rabattpreis_2"
011    FROM "Ware" WHERE "Ware_ID" = 1;
```

Der Primärschlüsselwert für die Ware müsste hier natürlich entsprechend über ein Makro gesetzt werden, das dem Kombinationsfeld je nach Ware einen entsprechenden Inhalt zuweist.

Hinweis

Die Sortierung von mit UNION verknüpften Inhalten gelingt bei FIREBIRD nicht über die Spaltennamen. Stattdessen muss die Position der Spalte übergeben werden:

```
012 SELECT
013    "Vorname", "Nachname"
014 FROM "Tabelle1"
015    UNION
016    SELECT
017       "Vorname", "Nachname"
018    FROM "Tabelle2"
019    ORDER BY 2, 1;
```

sortiert die Daten in Firebird zuerst nach dem Nachnamen (Spalte 2) und dann nach dem Vornamen (Spalte 1).

MINUS [DISTINCT] | EXCEPT [DISTINCT] Abfrageaussage SQL ~~GUI~~ **(**HSQLDB**,** ~~FIREBIRD~~**)**

```
001 SELECT
002    "Vorname"
003 FROM "Tabelle1"
004    EXCEPT
005    SELECT
006       "Vorname"
007    FROM "Tabelle2";
```

Zeigt alle Vornamen aus Tabelle1 mit Ausnahme der Vornamen an, die in Tabelle 2 enthalten sind. **MINUS** und **EXCEPT** führen zum gleichen Ergebnis. Sortierung ist alphabetisch. Dies funktioniert zur Zeit nur, wenn das SQL-Kommando direkt ausgeführt wird.

INTERSECT [DISTINCT] Abfrageaussage SQL ~~GUI~~ **(**HSQLDB~~, FIREBIRD~~**)**

```
001 SELECT
002    "Vorname"
003 FROM "Tabelle1"
004    INTERSECT
005    SELECT
006       "Vorname"
007    FROM "Tabelle2";
```

Hier werden nur die Vornamen angezeigt, die in beiden Tabellen vorhanden sind. Die Sortierung ist wieder alphabetisch. Dies funktioniert zur Zeit nur, wenn das SQL-Kommando direkt ausgeführt wird.

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

Hier können Feldnamen, die Nummer der Spalte (beginnend mit 1 von links), ein Alias (formuliert z. B. mit **AS**) oder eine Wertzusammenführung (siehe [SQL Expression]: Werte) angegeben werden. Die Sortierung erfolgt in der Regel aufsteigend (**ASC**). Nur wenn die Sortierung absteigend erfolgen soll, muss **DESC** angegeben werden.

```
001 SELECT
002    "Vorname",
003    "Nachname" AS "Name"
004 FROM "Tabelle1"
005 ORDER BY "Nachname";
```

ist identisch mit

```
001 SELECT
002    "Vorname",
003    "Nachname" AS "Name"
004 FROM "Tabelle1"
005 ORDER BY "Nachname" ASC;
```

ist identisch mit

```
001 SELECT
002    "Vorname",
003    "Nachname" AS "Name"
```

```
004 FROM "Tabelle1"
005 ORDER BY "Name";
```

Unter FIREBIRD funktioniert diese Sortierung allerdings nicht wie gewünscht, wenn Umlaute in den Bezeichnungen enthalten sind. Hier muss zusätzlich die Art der **Collation** angegeben werden:

```
001 SELECT
002    "Name"
003 FROM "Tabelle1"
004 ORDER BY "Name" COLLATE UNICODE ASC;
```

Leider funktioniert diese Abfrage nur im direkten SQL-Modus und auch nicht mit den Sortierpfeilen in der Abfrage- und Tabellen-GUI. Besser ist es, die «Sortierung von Groß- und Kleinschreibung und auch Umlauten» direkt vor der Erstellung der Tabellen in Firebird anzugeben.

Verwendung eines Alias in Abfragen

Durch Abfragen können auch Felder in einer anderen Bezeichnung wiedergegeben werden.

```
001 SELECT
002    "Vorname",
003    "Nachname" AS "Name"
004 FROM "Tabelle1"
```

Dem Feld "Nachname" wird in der Anzeige die Bezeichnung "Name" zugeordnet.

Wird eine Abfrage aus zwei Tabellen erstellt, so steht vor den jeweiligen Feldbezeichnungen der Name der Tabelle:

```
001 SELECT
002    "Tabelle1"."Vorname",
003    "Tabelle1"."Nachname" AS "Name",
004    "Tabelle2"."Klasse"
005 FROM "Tabelle1",
006    "Tabelle2"
007 WHERE "Tabelle1"."Klasse_ID" = "Tabelle2"."ID"
```

Auch den Tabellennamen kann ein Aliasname zugeordnet werden, der allerdings in der Tabellenansicht nicht weiter erscheint. Wird so ein Alias zugeordnet, so müssen sämtliche Tabellenbezeichnungen in der Abfrage entsprechend ausgetauscht werden:

```
001 SELECT
002    "a"."Vorname",
003    "a"."Nachname" AS "Name",
004    "b"."Klasse"
005 FROM "Tabelle1" AS "a",
006    "Tabelle2" AS "b"
007 WHERE "a"."Klasse_ID" = "b"."ID"
```

Die Zuweisung eines Aliasnamens kann auch verkürzt ohne den Zuweisungsbegriff **AS** erfolgen:

```
001 SELECT
002    "a"."Vorname",
003    "a"."Nachname" "Name",
004    "b"."Klasse"
005 FROM "Tabelle1" "a",
006    "Tabelle2" "b"
007 WHERE "a"."Klasse_ID" = "b"."ID"
```

Dies erschwert allerdings die eindeutige Lesbarkeit des Codes. Daher sollte nur in Ausnahmefällen auf die Verkürzung zurückgegriffen werden.

Über den Aliasnamen kann auch eine Tabelle mit entsprechenden Filterungen mehrmals innerhalb einer Abfrage genutzt werden:

```
001 SELECT "Kasse"."Betrag", "Kasse"."Datum",
002    "a"."Betrag" AS "Haben",
003    "b"."Betrag" AS "Soll"
004 FROM "Kasse"
005    LEFT JOIN "Kasse" AS "a"
006       ON "Kasse"."ID" = "a"."ID" AND "a"."Betrag" >= 0
007    LEFT JOIN "Kasse" AS "b"
008       ON "Kasse"."ID" = "b"."ID" AND "b"."Betrag" < 0
```

Abfragen für die Erstellung von Listenfeldern

In Listenfeldern wird ein Wert angezeigt, der nicht an die den Formularen zugrundeliegenden Tabellen weitergegeben wird. Sie werden schließlich eingesetzt, um statt eines Fremdschlüssels zu sehen, welchen Wert der Nutzer damit verbindet. Der Wert, der schließlich in Formularen abgespeichert wird, darf bei den Listenfeldern nicht an der ersten Position stehen.

```
001 SELECT
002    "Vorname",
003    "ID"
004 FROM "Tabelle1";
```

Diese Abfrage würde alle Vornamen anzeigen und den Primärschlüssel "ID" an die dem Formular zugrundeliegende Tabelle weitergeben. So ist das natürlich noch nicht optimal. Die Vornamen erscheinen unsortiert; bei gleichen Vornamen ist außerdem unbekannt, um welche Person es sich denn handelt.

```
001 SELECT
002    "Vorname"||' '||"Nachname",
003    "ID"
004 FROM "Tabelle1"
005 ORDER BY "Vorname"||' '||"Nachname";
```

Jetzt erscheint der Vorname, getrennt von dem Nachnamen, durch ein Leerzeichen. Die Namen werden unterscheidbarer und sind sortiert. Die Sortierung folgt der Logik, dass natürlich nach den vorne stehenden Buchstaben zuerst sortiert wird, also Vornamen zuerst, dann Nachnamen. Andere Sortierreihenfolgen als nach der Anzeige des Feldes würden erst einmal verwirren.

```
001 SELECT
002    "Nachname"||', '||"Vorname",
003    "ID"
004 FROM "Tabelle1"
005 ORDER BY "Nachname"||', '||"Vorname";
```

Dies würde jetzt zu der entsprechenden Sortierung führen, die eher unserer Gewohnheit entspricht. Familienmitglieder erscheinen zusammen untereinander, bei gleichen Nachnamen und unterschiedlichen Familien wird allerdings noch durcheinander gewürfelt. Um dies zu unterscheiden müsste eine Gruppenzuordnung in der Tabelle gemacht werden.

Letztes Problem ist noch, dass eventuell auftauchende Personen mit gleichem Nachnamen und gleichem Vornamen nicht unterscheidbar sind. Variante 1 wäre, einfach einen Namenszusatz zu entwerfen. Aber wie sieht das denn aus, wenn ein Anschreiben mit Herr «Müller II» erstellt wird?

```
001 SELECT
002    "Nachname"||', '||"Vorname"||' - ID:'||"ID",
003    "ID"
004 FROM "Tabelle1"
005 ORDER BY "Nachname"||', '||"Vorname"||"ID";
```

Hier wird auf jeden Fall jeder Datensatz unterscheidbar. Angezeigt wird «Nachname, Vorname – ID:IDWert».

Falls es einmal passieren sollte, dass Felder für die Vornamen leer (**NULL**) sind, so wird so ein Datensatz in den Listenfeldern natürlich erst einmal nicht angezeigt. Leere Felder, verbunden mit anderen Feldern, sind ebenfalls **NULL**. Hier hilft dann eine entsprechend eingefügte Bedingung:

```
001 SELECT
002    "Nachname"||COALESCE(', '||"Vorname",''),
003    "ID"
004 FROM "Tabelle1"
005 ORDER BY "Nachname"||COALESCE(', '||"Vorname",'');
```

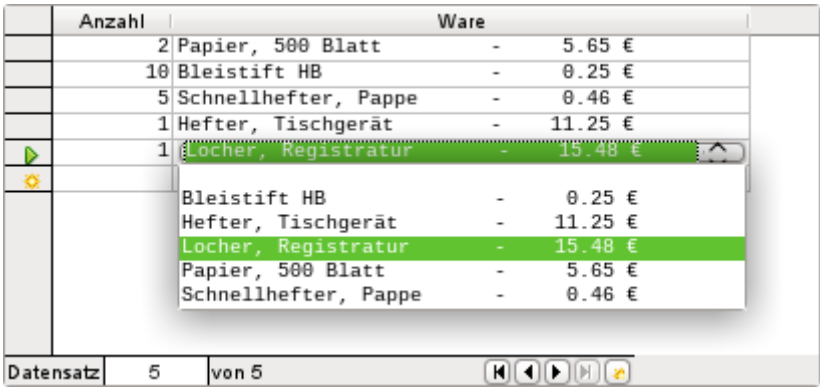
Damit wird das Komma und der anschließende Vorname durch einen leeren Text ersetzt, wenn das Feld "Vorname" **NULL** ist.

Im Formular Ausleihe wurde ein Listenfeld erstellt, das lediglich die Medien darstellen sollte, die noch nicht entliehen wurden. Dies wird über die folgende SQL-Formulierung möglich:

```
001 SELECT
002    "Titel" || ' - Nr. ' || "ID",
003    "ID"
004 FROM "Medien"
005 WHERE "ID"
006    NOT IN
007       (SELECT
008          "Medien_ID"
009       FROM  "Ausleihe"
010       WHERE "Rueck_Datum" IS NULL)
011 ORDER BY "Titel" || ' - Nr. ' || "ID" ASC
```

Dieses Listenfeld muss allerdings immer dann aktualisiert werden, wenn eine Ausleihe eines darin verzeichneten Mediums erfolgt ist.

Das folgende Listenfeld stellt die Inhalte mehrerer Felder in Tabellenform dar, so dass zusammengehörige Elemente direkt untereinander erscheinen:



Damit so eine Darstellung gelingt, muss zuerst einmal eine entsprechende nicht proportionale Schrift ausgewählt werden. Courier oder alle Mono-Schriftarten wie z.B. Liberation Mono können hier genutzt werden. Die Darstellung in tabellarischer Form gelingt über den SQL-Code (HSQLDB)

```
001 SELECT
002    LEFT("Ware"||SPACE(25),25) || ' - ' ||
          RIGHT(SPACE(8)||"Preis",8) || ' €',
003    "ID"
004 FROM "Waren"
005 ORDER BY ("Ware" || ' - ' || "Preis" || ' €') ASC
```

FIREBIRD kennt die Funktion **SPACE** nicht, dafür aber Funktionen wie **LPAD** und **RPAD**:
LEFT("Ware"||SPACE(25),25) wird ersetzt durch **RPAD("Ware",25)**
RIGHT(SPACE(8)||"Preis",8) wird ersetzt durch **LPAD("Preis",8)**
An den Inhalt des Feldes "Ware" werden entsprechend viele Leerzeichen angehängt, so dass "Ware" zusammen mit den Leerzeichen eine Mindestlänge von 25 Zeichen hat. Anschließend

werden die ersten 25 Buchstaben dargestellt, die überflüssigen Leerzeichen also abgeschnitten.

Komplizierter wird es, wenn in dem Inhalt des Listenfeldes auch nicht druckbare Zeichen wie z.B. Zeilenumbrüche enthalten sind. Dann muss der Code entsprechend angepasst werden:

```
001 SELECT
002    LEFT(REPLACE("Ware",CHAR(10),' ')||SPACE(25), 25) || ' - ' || ...
```

Damit wird ein Zeilenvorschub in Linux durch ein Leerzeichen ersetzt. In Windows muss zusätzlich noch der Wagenrücklauf (**CHAR(13)**) entfernt werden.
FIREBIRD: Die Funktion **CHAR()** heißt dort **ASCII_Char()**.

Die Anzahl der notwendigen Leerzeichen kann übrigens auch per Abfrage ermittelt werden. So wird vermieden, dass doch einmal ein Wert aus "Ware" in seiner Länge beschnitten wird.

```
001 SELECT
002    LEFT("Ware"||SPACE(
003       (SELECT MAX(CHAR_LENGTH("Ware")) FROM "Waren")),
004       (SELECT MAX(CHAR_LENGTH("Ware")) FROM "Waren"))
005       || ' - ' || RIGHT('    '||"Preis",8) || ' €',
006    "ID"
007 FROM "Waren"
008 ORDER BY ("Ware" || ' - ' || "Preis" || ' €') ASC
```

Da der Preis rechtsbündig dargestellt werden soll, wird hier vor dem Preis mit Leerzeichen aufgefüllt und entsprechend maximal 8 Zeichen von rechts aus dargestellt. Die angestrebte Darstellung reicht also für alle Preise bis 99999,99 € aus.

Soll auch noch der Punkt aus der SQL-Darstellung durch ein Komma ersetzt werden, so ist der SQL-Code entsprechend zu ergänzen. Dies wäre über

```
001 REPLACE(RIGHT('    '||"Preis",8),'.',',')
```

möglich.

Abfragen als Grundlage von Zusatzinformationen in Formularen

Will man zusätzliche Informationen im Formular im Auge haben, die so gar nicht sichtbar wären, so bieten sich verschiedene Abfragemöglichkeiten an. Die einfachste ist, mit gesonderten Abfragen diese Informationen zu ermitteln und in andere Formulare einzustellen. Der Nachteil dieser Variante kann sein, dass sich Datenänderungen auf das Abfrageergebnis auswirken würden, aber leider die Änderungen nicht automatisch angezeigt werden.

Hier ein Beispiel aus dem Bereich der Warenwirtschaft für eine einfache Kasse:

Die Tabelle für eine Kasse enthält Anzahl und Fremdschlüssel von Waren, außerdem eine Rechnungsnummer. Besonders wenig Informationen erhält der Nutzer an der Supermarktkasse, wenn keine zusätzlichen Abfrageergebnisse auf den Kassenzettel gedruckt werden. Schließlich werden die Waren nur über den Barcode eingelesen. Ohne Abfrage ist im Formular nur zusehen:

Anzahl	***Barcode***
3	17
2	24

usw.

Was sich hinter den Nummern versteckt, kann nicht mit einem Listenfeld sichtbar gemacht werden, da ja der Fremdschlüssel über den Barcode direkt eingegeben wird. So lässt sich auch nicht über das Listenfeld neben der Ware zumindest der Einzelpreis ersehen.

Hier hilft eine Abfrage:

```
001 SELECT
002    "Kasse"."RechnungID",
003    "Kasse"."Anzahl",
004    "Ware"."Ware",
005    "Ware"."Einzelpreis",
006    "Kasse"."Anzahl"*"Ware"."Einzelpreis" AS "Gesamtpreis"
007 FROM "Kasse", "Ware"
008 WHERE "Ware"."ID" = "Kasse"."WareID";
```

Jetzt ist zumindest schon einmal nach der Eingabe die Information da, wie viel denn für 3 * Ware'17' zu bezahlen ist. Außerdem werden nur die Informationen durch das Formular zu filtern sein, die mit der entsprechenden "RechnungID" zusammenhängen. Was auf jeden Fall noch fehlt, ist das, was denn der Kunde nun bezahlen muss:

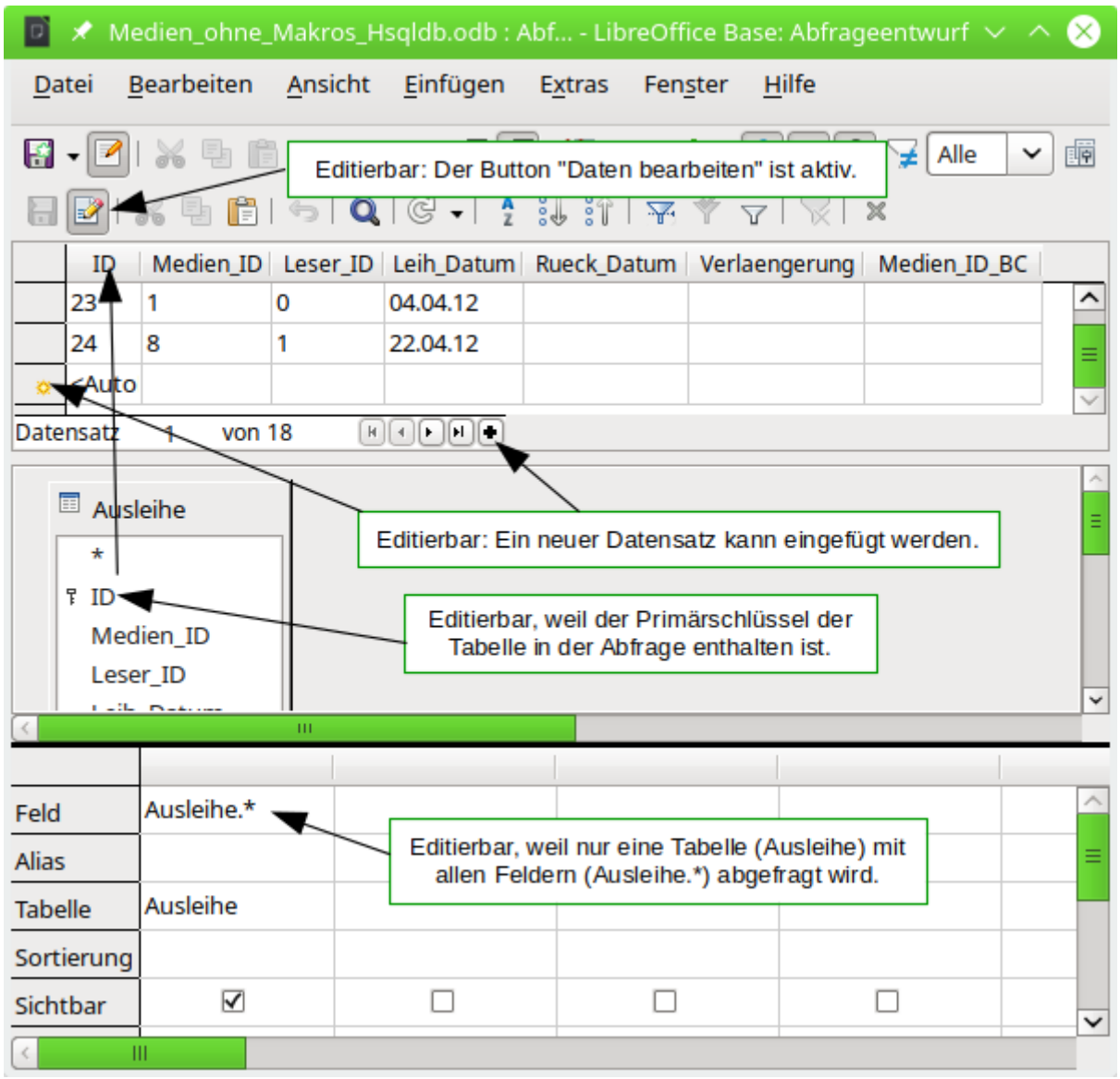
```
001 SELECT
002    "Kasse"."RechnungID",
003    SUM("Kasse"."Anzahl"*"Ware"."Einzelpreis") AS "Summe"
004 FROM "Kasse", "Ware"
005 WHERE "Ware"."ID" = "Kasse"."WareID"
006 GROUP BY "Kasse"."RechnungID";
```

Für das Formular liefert diese Abfrage nur eine Zahl, da über das Formular natürlich wieder die "RechnungID" gefiltert wird, so dass ein Kunde nicht gleichzeitig sieht, was andere vor ihm eventuell gezahlt haben.

Tipp	Sollen in einem Formular Datumswerte dargestellt werden, die in Abhängigkeit von einem anderen Datum definiert sind (z.B.: Eine Entleihzeit für ein Medium beträgt 21 Tage – wann muss das Medium zurückgegeben werden?), dann ist dies mit Standardfunktionen der HSQLDB nicht zu machen. Es fehlt eine Funktion wie **DATEADD**. Die Abfrage **SELECT "Datum", DATEDIFF('dd','1899-12-30',"Datum")+21 AS "RueckDatum" FROM "Tabelle"** würde in einem Formular als Datum formatiert das korrekte anvisierte Rückgabedatum ergeben. Mit dieser Abfrage werden die Tage ab dem 30.12.1899 gezählt. Der 30.12.1899 ist das Standarddatum, das z.B. auch Calc als 0-Wert nutzt. Allerdings handelt es sich bei dem ermittelten Wert um eine Zahl, nicht um ein Datum, das anschließend in z.B. einer Abfrage weiter genutzt werden kann. In einer Abfrage lässt sich die ermittelte Zahl schlecht nutzen, da die Formatierung von Abfragen nicht gespeichert wird. Dafür müsste eine Ansicht erstellt werden. **Achtung!** Base selbst zeigt zumindest bis LO 7.4 Datumswerte, die auf Integer-Zahlen beruhen, unterschiedlich an. In Tabellen, Abfragen und formatierbaren Feldern des Formulars wird der 30.12.1899 als 0-Wert genommen. Datumsfelder hingegen berechnen aus 0 das Datum 1.1.1900. FIREBIRD kennt die Funktion **DATEADD** und kann sogar direkt zu einem Datumswert eine Ganzzahl addieren, die dann als Tag gedeutet wird. Für FIREBIRD sähe die obige Abfrage entsprechend deutlich einfacher aus: **SELECT "Datum", "Datum"+21 AS "RueckDatum" FROM "Tabelle"**

Eingabemöglichkeit in Abfragen

Um Eingaben in Abfragen zu tätigen, muss der Primärschlüssel für die jeweils zugrundeliegende Tabelle in der Abfrage enthalten sein.



Hinweis	Ist in einer Tabelle eine Spalte ausgeblendet worden, so erscheint sie nicht als Spaltenkopf bei den Daten. Dies kann nur umgangen werden, indem die Spalte wieder angezeigt wird oder die Abfrage in direktem SQL ausgeführt wird. Durch die Ausführung in direktem SQL wird eine Abfrage aber grundsätzlich nicht für Dateneingaben editierbar.

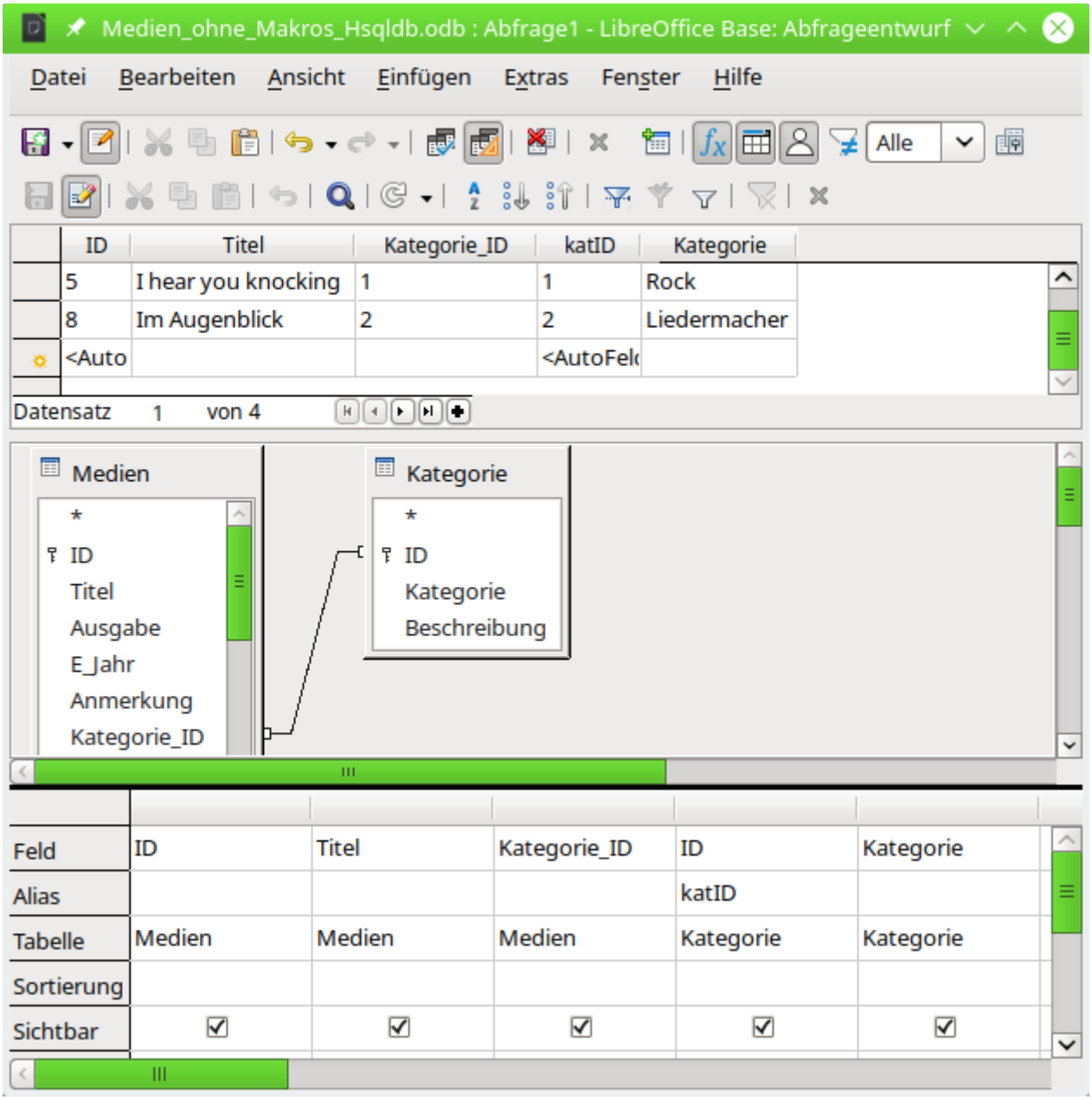
Bei der Ausleihe von Medien ist es z. B. nicht sinnvoll, für einen Leser auch die Medien noch anzeigen zu lassen, die längst zurückgegeben wurden.

```
001 SELECT
002    "ID",
003    "LeserID",
004    "MedienID",
005    "Ausleihdatum"
006 FROM "Ausleihe"
007 WHERE "Rückgabedatum" IS NULL;
```

So lässt sich im Formular innerhalb eines Tabellenkontrollfeldes all das anzeigen, was ein bestimmter Leser zur Zeit entliehen hat. Auch hier ist die Abfrage über entsprechende Formularkonstruktion (Leser im Hauptformular, Abfrage im Unterformular) zu filtern, so dass nur die tatsächlich entliehenen Medien angezeigt werden. Die Abfrage ist zur Eingabe geeignet, da **der Primärschlüssel in der Abfrage enthalten** ist.

Die Abfrage wird dann nicht mehr editierbar, wenn sie aus mehreren Tabellen besteht und die Tabellen über einen Alias-Namen angesprochen werden. Dabei ist es unerheblich, ob die Primärschlüssel in der Abfrage enthalten sind.

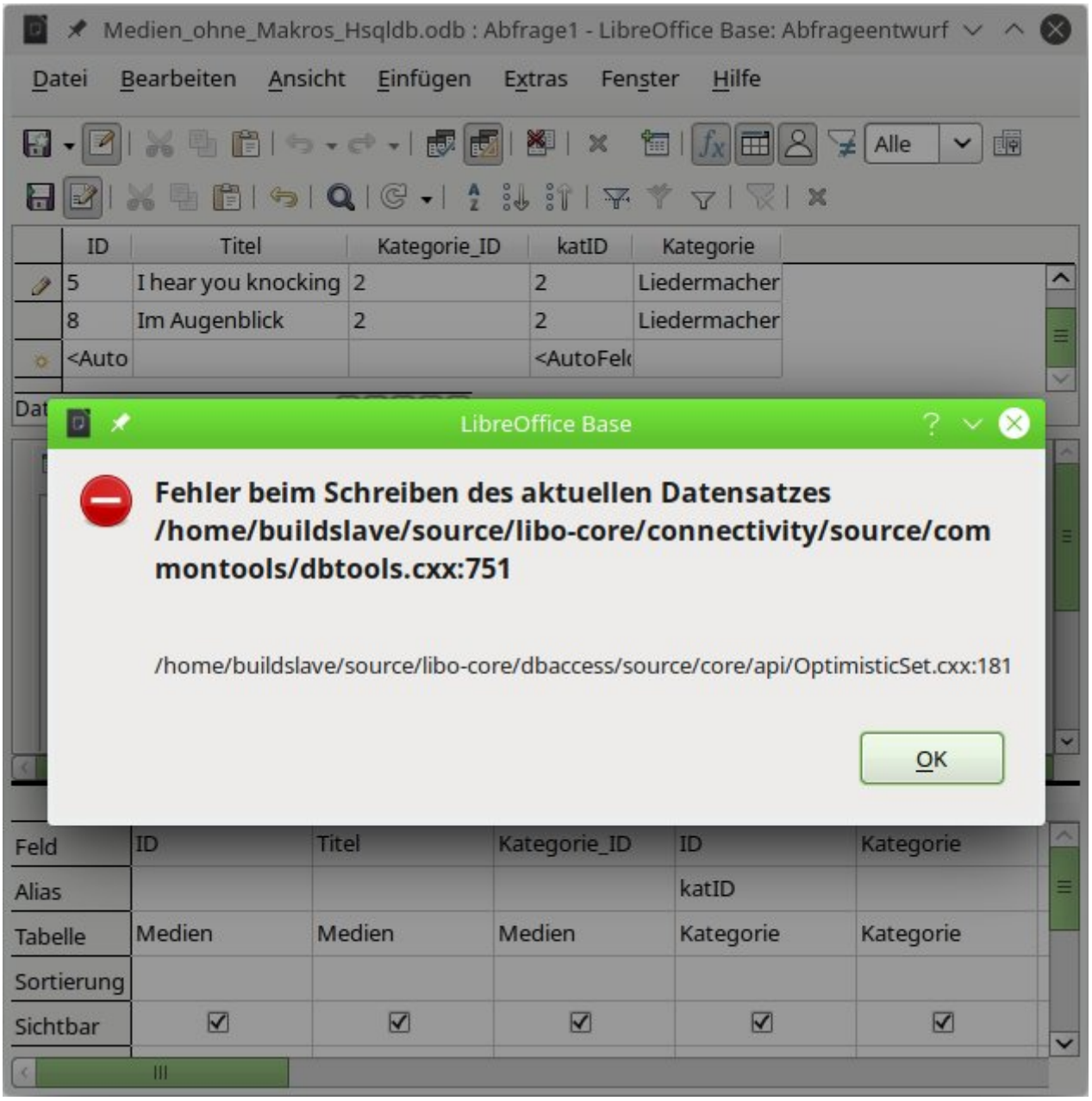
Hinweis	Sind in einer Abfrage gleiche Felder einer Tabelle mehrmals hintereinander vorhanden (z.B. einmal als "Tabelle"."Name" und einmal als "Tabelle".*), so nimmt die GUI eine Veränderung des Inhaltes nur in dem zuletzt erscheinenden Feld wahr. Es sollte daher grundsätzlich vermieden werden, das gleiche Feld mehrmals unverändert in eine Abfrage aufzunehmen. Übersicht und Editierbarkeit leiden darunter.




```
001 SELECT
002    "Medien"."ID",
003    "Medien"."Titel",
004    "Medien"."Kategorie_ID",
005    "Kategorie"."ID" AS "katID"
006    "Kategorie"."Kategorie",
007 FROM "Medien",
```

```
008    "Kategorie"
009 WHERE "Medien"."Kategorie_ID" = "Kategorie"."ID";
```

Diese Abfrage bleibt editierbar, da **beide Primärschlüssel enthalten** sind und auf die Tabellen nicht mit einem Alias zugegriffen wird. Auch bei einer Abfrage, die mehrere Tabellen miteinander verknüpft, müssen alle Primärschlüssel vorhanden sein. Außerdem muss eine Verbindung zwischen einem Feld der einen Tabelle und dem Primärschlüssel der anderen Tabelle definiert werden. Dies ist unabhängig davon, ob unter **Extras → Beziehungen** solch eine Verbindung (als Fremdschlüssel zu Primärschlüssel) bereits existiert.



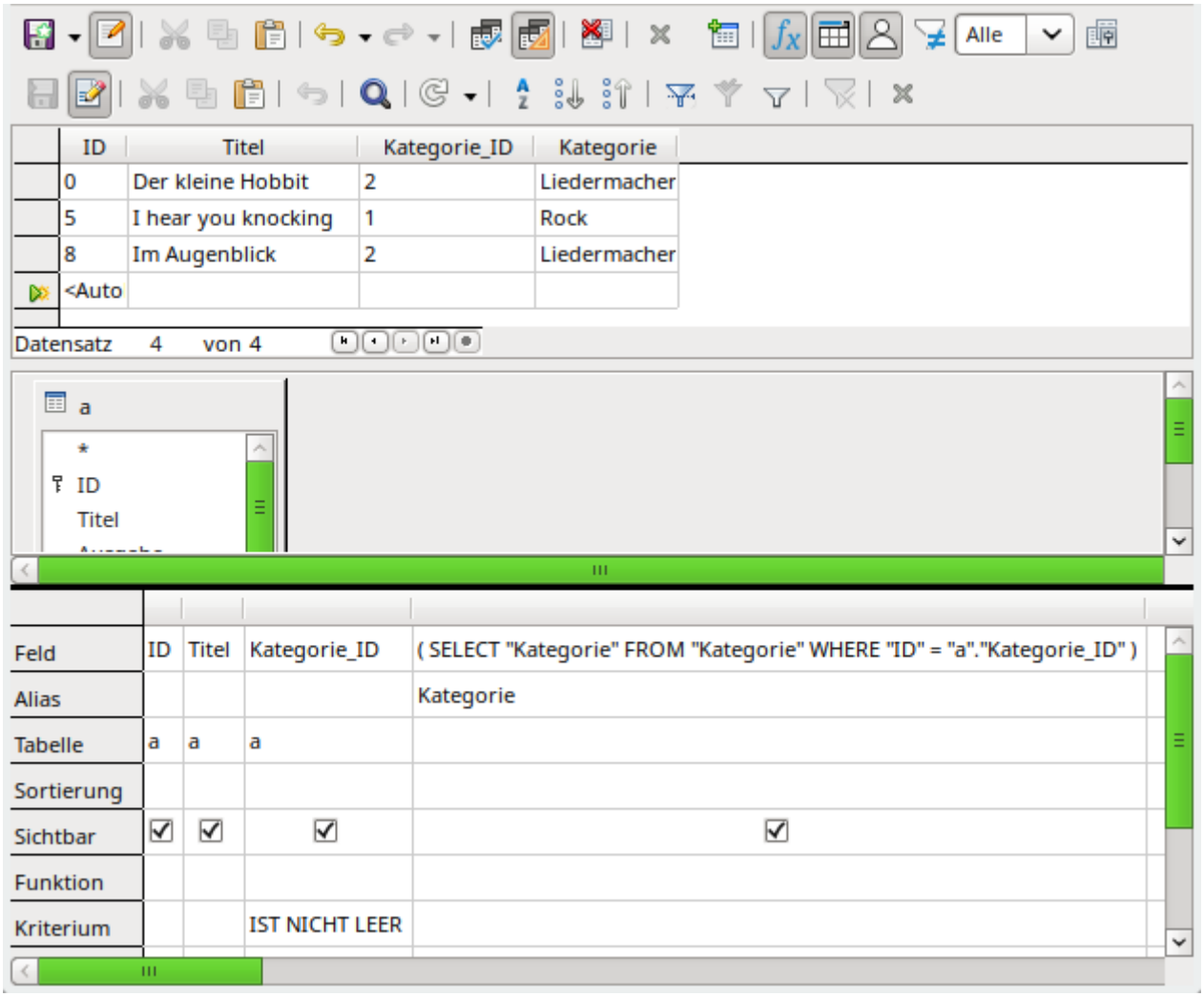
Es ist allerdings nicht möglich, in einer Abfrage, die auf mehreren Tabellen beruht, das Fremdschlüsselfeld der einen Tabelle, das sich auf die andere Tabelle bezieht, zu ändern. Im angezeigten Datensatz wurde versucht, die Kategorie für den Titel 'I hear you knocking' umzustellen. Das Feld "Kategorie_ID" wurde von '1' auf '2' geändert. Die Änderung schien vollzogen, die neue Kategorie erschien auch. Das Speichern war dann aber nicht möglich. Lediglich der erste Satz der Fehlermeldung ist hier aber für den Normalnutzer brauchbar.

Allerdings ist es möglich, den Inhalt von "Kategorie" selbst zu bearbeiten, also z.B. 'Fantasy' zum Begriff 'Fantasie' zu ändern. Das ändert dann allerdings diesen Begriff für die Kategorie, betrifft also alle damit verbundenen Datensätze.

```
001 SELECT
002    "a"."ID",
003    "a"."Titel",
004    "Kategorie"."Kategorie",
005    "Kategorie"."ID" AS "katID"
006 FROM "Medien" AS "a",
007    "Kategorie"
008 WHERE "a"."Kategorie_ID" = "Kategorie"."ID";
```

In dieser Abfrage wird auf die Tabelle "Medien" mit einem **Alias** zugegriffen. Sie wird jetzt **nicht editierbar** sein, weil das "a" für "Medien" auch in Verbindung mit Feldnamen auftaucht ("a"."ID" sowie "a"."Titel".

In dem obigen Beispiel ist das leicht vermeidbar. Wenn aber eine *Korrelierte Unterabfrage* erstellt wird, so muss mit einem Tabellenalias gearbeitet werden. So eine **Abfrage mit Alias** kann nur **dann editierbar** bleiben, wenn sie **lediglich eine Tabelle in der Hauptabfrage** enthält.



In der Design-Ansicht erscheint hier lediglich eine Tabelle. Die Tabelle "Medien" ist mit einem Alias versehen, damit auf die Inhalte des Feldes "Kategorie_ID" mit der korrelierenden Unterabfrage zugegriffen werden kann.

In so einer Abfrage ist es jetzt möglich, das Fremdschlüsselfeld "Kategorie_ID" auf eine andere Kategorie umzustellen. In dem obigen Beispiel wurde das Feld "Kategorie_ID" von '0' auf '2'

geändert. Dem "Titel" 'Der kleine Hobbit' wurde so statt der "Kategorie" 'Fantasy' die "Kategorie" 'Liedermacher' zugeordnet.

Allerdings ist es jetzt nicht mehr möglich, einen Wert in dem Feld zu ändern, das seinen Inhalt über die korrelierende Unterabfrage erhält. Die Änderung der "Kategorie" von 'Fantasy' in 'Fantasie' wird erst einmal angezeigt. Die Änderung wird allerdings nicht registriert und ist auch nicht abspeicherbar. In der Tabelle, die die Entwurfsansicht anzeigt, ist das Feld "Kategorie" schließlich nicht enthalten.

Abfragen bei externen Datenbanken (z.B. MySQL/MariaDB) über mehrere Tabellen sind manchmal deswegen nicht editierbar, weil die GUI sie bereits beim Erstellen mit einem Alias versieht: "Datenbankname"."Tabellenname" AS "Tabellenname". Hier kann einfach nachgebessert werden, indem «"Datenbankname"."Tabellenname" AS» entfernt wird. Es reichen in der Regel die Tabellenbezeichnungen.

Verwendung von Parametern in Abfragen

Wird viel mit gleichen Abfragen, aber eventuell unterschiedlichen Werten als Voraussetzung zu diesen Abfragen gearbeitet, so sind Parameterabfragen möglich. Vom Prinzip her funktionieren Parameterabfragen erst einmal genauso wie Abfragen, die in Unterformularen abgelegt werden:

```
001 SELECT
002    "ID",
003    "Leser_ID",
004    "Medien_ID",
005    "Ausleihdatum"
006 FROM "Ausleihe"
007 WHERE "Rückgabedatum" IS NULL
008    AND "Leser_ID"=2;
```

Diese Abfrage zeigt nur die entliehenen Medien des Lesers mit der Nummer '2' an.

```
001 SELECT
002    "ID",
003    "Leser_ID",
004    "Medien_ID",
005    "Ausleihdatum"
006 FROM "Ausleihe"
007 WHERE "Rückgabedatum" IS NULL
008    AND "LeserID" = :Lesernummer;
```

Jetzt erscheint beim Aufrufen der Abfrage ein Eingabefeld. Es fordert zur Eingabe einer Lesernummer auf. Wird hier ein Wert eingegeben, so werden die zur Zeit entliehenen Medien dieses Lesers angezeigt.

```
001 SELECT
002    "Ausleihe"."ID",
003    "Leser"."Nachname"||', '||"Leser"."Vorname",
```

```
004    "Ausleihe"."Medien_ID",
005    "Ausleihe"."Ausleihdatum"
006 FROM "Ausleihe", "Leser"
007 WHERE "Ausleihe"."Rückgabedatum" IS NULL
008    AND "Leser"."ID" = "Ausleihe"."Leser_ID"
009    AND "Leser"."Nachname" LIKE '%' || :Lesername || '%'
010 ORDER BY "Leser"."Nachname"||', '||"Leser"."Vorname" ASC;
```

Diese Abfrage ist noch deutlich komfortabler als die vorhergehende. Jetzt muss nicht eine Nummer des Lesers bekannt sein. Es reicht die Eingabe eines Teils des Nachnamen des Lesers und alle Medien von Lesern, auf die diese Beschreibung zutrifft, werden angezeigt.

Wird

```
007    AND "Leser"."Nachname" LIKE '%' || :Lesername || '%'
```

durch

```
007    AND LOWER("Leser"."Nachname") LIKE '%' || LOWER( :Lesername ) || '%'
```

ersetzt, so ist es auch noch egal, ob die Namen groß oder klein geschrieben sind.

Wird der **Parameter** in der obigen Abfrage **leer** gelassen, so werden bis LO 4.4 **alle Leser** angezeigt, da ein leeres Parameterfeld erst ab der Version LO 4.4 auch tatsächlich nicht als leerer Text, sondern als **NULL** weiter gegeben wird. Soll dies vermieden werden, so muss etwas in die Trickkiste gegriffen werden:

```
007    AND LOWER ("Leser"."Nachname") LIKE '%' ||
          IFNULL( NULLIF ( LOWER (:Lesername), '' ), '§§' ) || '%'
```

Das leere Parameterfeld gibt an die Abfrage einen leeren String, aber nicht NULL weiter. Deshalb muss erst einmal dem leeren Parameterfeld die Eigenschaft NULL mit **NULLIF** zugewiesen werden. Anschließend wird für den Fall, dass eben die Parametereingabe jetzt NULL ergibt, dieser Eingabe ein Wert zugewiesen, der in der Regel in keinem der Datensätze vorkommt. In dem obigen Beispiel ist das **'§§'**. Dieser Wert wird mit der Abfrage entsprechend auch nicht gefunden.

Ab der Version **LO 4.4** muss diese Abfragetechnik etwas angepasst werden:

```
007    AND LOWER ("Leser"."Nachname") LIKE '%' || LOWER (:Lesername) || '%'
```

ergibt zwangsläufig bei einer fehlenden Eingabe für die gesamte Kombination **'%' || LOWER (:Lesername) || '%'** den Wert **NULL**.

Dagegen hilft das Hinzufügen einer weiteren Bedingung, dass bei einem leeren Feld tatsächlich alle Werte aufgezeigt werden:

```
007    AND (LOWER ("Leser"."Nachname") LIKE '%' || LOWER (:Lesername) || '%'
          OR :Lesername IS NULL)
```

Dies sollte in Klammern gesetzt werden. So wird entweder ein Name ausgesucht oder, bei leerem Feld, also **NULL** ab LO 4.4, die zweite Bedingung erfüllt.

Für die interne Firebird-Datenbank muss der zweite Eintrag des Parameters des verbundenen Feldes in den Datentyp **VARCHAR()** umgewandelt werden. Die Länge des Strings muss natürlich wie bei Felddefinitionen mit angegeben werden, hier also **VARCHAR(50)**. Dabei ist es egal, ob der Parameter innerhalb der Abfrage mit einem Text, einer Zahl oder einem Datum verglichen wird. Der Code muss folgendermaßen in Firebird aussehen:

```
007    AND (LOWER ("Leser"."Nachname") LIKE '%' || LOWER (:Lesername) || '%'
          OR CAST( :Lesername AS VARCHAR(50) ) IS NULL)
```

Ein Parameter kann bei Formularen auch von einem Hauptformular an ein Unterformular weitergegeben werden. Es kann allerdings passieren, dass Parameterabfragen in Unterformularen nicht aktualisiert werden, wenn Daten geändert oder neu eingegeben werden.

Manchmal wäre es wünschenswert, Listenfelder in Abhängigkeit von Einstellungen des Hauptformulars zu ändern. So könnte beispielsweise ausgeschlossen werden, dass in einer Bibliothek Medien an Personen entliehen werden, die zur Zeit keine Medien ausleihen dürfen. Leider ist

aber eine derartige Listenfelderstellung in Abhängigkeit von der Person über Parameter nicht möglich.

Tipp	Parameterabfragen sind auch eine Möglichkeit, für Berichte Daten vor dem Start des Berichtes zu filtern. Daneben ermöglichen solche Abfragen auch, einem Bericht bestimmte Textbausteine mitzugeben, die sonst nicht in der Abfrage enthalten sind. Mit **SELECT "Tabelle".*, :Ueberschrift FROM "Tabelle"** wird die Variable «Ueberschrift» abgefragt und kann dann in einem Textfeld des Berichtes über **=Ueberschrift** oder **=:Ueberschrift** ausgelesen werden.

Hinweis	Der Parameter innerhalb einer Abfrage sollte nie genau die gleiche Bezeichnung haben wie ein Feld, das in der gleichen Abfrage auftaucht. In dem Moment können Formulare und Berichte die Parameter von den Feldern nicht mehr unterscheiden. Gerade in Berichten führt dies dazu, dass eventuell die Werte des Parameters statt des Feldes mit dem gleichen Namen angezeigt werden.

Unterabfragen

Unterabfragen, die in Felder eingebaut werden, dürfen immer nur genau einen Datensatz wiedergeben. Das Feld kann schließlich auch nur einen Wert wiedergeben.

```
001 SELECT
002    "ID",
003    "Einnahme",
004    "Ausgabe",
005    ( SELECT
006       SUM( "Einnahme" ) - SUM( "Ausgabe" )
007    FROM "Kasse")
008    AS "Saldo"
009 FROM "Kasse";
```

Diese Abfrage ist eingabefähig (Primärschlüssel vorhanden). Die Unterabfrage liefert nur genau einen Wert, nämlich die Gesamtsumme. Damit lässt sich nach jeder Eingabe der Kassenstand ablesen. Dies ist noch nicht vergleichbar mit der Supermarktkasse aus *Abfragen als Grundlage von Zusatzinformationen in Formularen*. Es fehlt natürlich die Einzelberechnung aus Anzahl * Einzelpreis, aber auch die Berücksichtigung der Rechnungsnummer. Es wird immer die Gesamtsumme ausgegeben. Zumindest die Rechnungsnummer lässt sich über eine Parameterabfrage einbauen:

```
001 SELECT
002    "ID",
003    "Einnahme",
004    "Ausgabe",
005    ( SELECT
          SUM( "Einnahme" ) - SUM( "Ausgabe" )
       FROM "Kasse"
       WHERE "RechnungID" = :Rechnungsnummer)
       AS "Saldo"
006 FROM "Kasse"
007 WHERE "RechnungID" = :Rechnungsnummer;
```

Bei einer Parameterabfrage muss der Parameter in beiden Abfrageanweisungen gleich sein, wenn er als ein Parameter verstanden werden soll.

Für Unterformulare können diese Parameter mitgegeben werden. Unterformulare erhalten dann statt der Feldbezeichnung die darauf bezogene Parameterbezeichnung. Die Eingabe dieser Verknüpfung ist nur in den Eigenschaften der Unterformulare, nicht über den Assistenten möglich.

Hinweis	Unterformulare, die auf Abfragen beruhen, werden nicht in Bezug auf die Parameter automatisch aktualisiert. Es bietet sich also eher an, den Parameter direkt aus dem darüber liegenden Formular weiter zu geben.

Korrelierte Unterabfrage

Mittels einer noch verfeinerten Abfrage lässt sich innerhalb einer bearbeitbaren Abfrage sogar der laufende Kontostand mitführen:

```
001 SELECT
002    "ID",
003    "Einnahme",
004    "Ausgabe",
005    ( SELECT
          SUM( "Einnahme" ) - SUM( "Ausgabe" )
       FROM "Kasse"
       WHERE "ID" <= "a"."ID" )
       AS "Saldo"
006 FROM "Kasse" AS "a"
007 ORDER BY "ID" ASC
```

Die Tabelle "Kasse" ist gleichbedeutend mit der Tabelle "a". "a" stellt aber nur den Bezug zu den in diesem Datensatz aktuellen Werten her. Damit ist der aktuelle Wert von "ID" aus der äußeren Abfrage innerhalb der Unterabfrage auswertbar. Auf diese Weise wird in Abhängigkeit von der "ID" der jeweilige Saldo zu dem entsprechenden Zeitpunkt ermittelt, wenn einfach davon ausgegangen wird, dass die "ID" über den Autowert selbständig hoch geschrieben wird.

Hinweis	Soll nach den Inhalten der Unterabfrage mit Hilfe der Filterfunktionen des Abfrageeditors gefiltert werden, so funktioniert dies zur Zeit nur, wenn statt der einfachen Klammern zu Beginn und Ende der Unterabfrage die Klammern doppelt gesetzt werden: « **((SELECT ….)) AS "Saldo"** »

Abfragen als Bezugstabellen von Abfragen

In einer Abfrage soll für alle Leser, bei denen eine 3. Mahnung zu einem Medium vorliegt, ein Sperrvermerk ausgegeben werden.

```
001 SELECT
002    "Ausleihe"."Leser_ID",
003    '3 Mahnungen - der Leser ist gesperrt' AS "Sperre"
004 FROM
005    (SELECT COUNT( "Datum" ) AS "Anzahl",
006       "Ausleihe_ID"
007    FROM "Mahnung"
008    GROUP BY "Ausleihe_ID")
009    AS "a",
010    "Ausleihe"
011 WHERE "a"."Ausleihe_ID" = "Ausleihe"."ID"
012    AND "a"."Anzahl" > 2
```

Zuerst wird die **innere Abfrage** konstruiert, auf die sich die äußere Abfrage bezieht. In dieser Abfrage wird die Anzahl der Datumseinträge, gruppiert nach dem Fremdschlüssel "Ausleihe_ID", ermittelt. Dies muss unabhängig von der "Leser_ID" geschehen, da sonst nicht nur 3 Mahnungen bei einem Medium, sondern auch drei Medien mit einer ersten Mahnung zusammengezählt würden. Die innere Abfrage wird mit einem Alias versehen, damit sie mit der "Leser_ID" der äußeren Abfrage in Verbindung gesetzt werden kann.

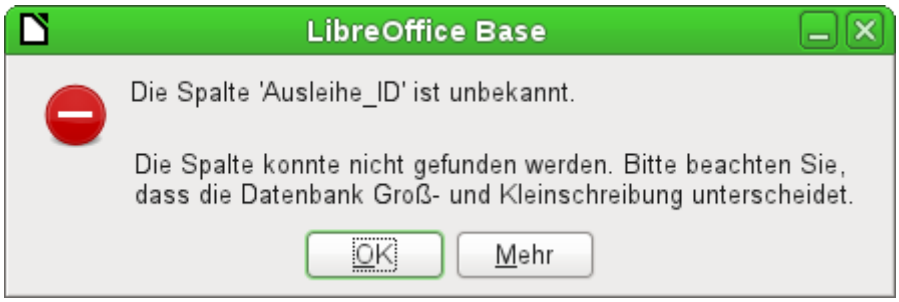
Hinweis	**Innere Abfragen** dürfen nicht sortiert werden. Die Sortierung erfolgt ausschließlich über die äußere Abfrage. Eine innere Abfrage mit Sortierung wird mit einer Fehlermeldung der HSQLDB quittiert: **`Cannot be in ORDER BY clause in statement [...]`**

Die **äußere Abfrage** bezieht sich in diesem Fall nur in der Bedingungsformulierung auf die innere Abfrage. Sie zeigt nur dann eine "Leser_ID" und den Text zur "Sperre" an, wenn "Ausleihe"."ID" *und* "a"."Ausleihe_ID" *gleich* sind sowie "a"."Anzahl" > *2* ist.

Prinzipiell stehen der äußeren Abfrage alle Felder der inneren Abfrage zur Verfügung. So ließe sich beispielsweise die Anzahl mit "a"."Anzahl" in der äußeren Abfrage einblenden, damit auch die tatsächliche Mahnzahl erscheint.

Es kann allerdings sein, dass im Abfrageeditor die grafische Benutzeroberfläche nach so einer Konstruktion nicht mehr verfügbar ist. Soll die Abfrage anschließend wieder zum Bearbeiten geöffnet werden, so erscheint die folgende Meldung:

Ist die Abfrage dann in der SQL-Ansicht zum Bearbeiten geöffnet und wird versucht von dort in die grafische Ansicht zu wechseln, so erscheint die Fehlermeldung



Die grafische Benutzeroberfläche findet also nicht das in der inneren Abfrage enthaltene Feld "Ausleihe_ID", mit dem die Beziehung von innerer und äußerer Abfrage geregelt wird.

Wird die Abfrage allerdings aufgerufen, so wird der entsprechende Inhalt anstandslos wiedergegeben. Es muss also *nicht der SQL-Befehl direkt ausgeführt* werden. Damit stehen auch die Sortier- und Filterfunktionen der grafischen Benutzeroberfläche weiter zur Verfügung.

Die folgenden Screenshots zeigen, wie der unterschiedliche Weg zu einem Abfrageergebnis mit Unterabfragen auch verlaufen kann. Hier soll in der Abfrage einer Rechnungsdatenbank ermittelt werden, was der Kunde an der Kasse letztlich zahlen muss. Die Preise werden multipliziert mit der Anzahl der entsprechenden Ware zum "Teilbetrag". Außerdem soll auch noch die Summe der Teilbeträge ausgegeben werden. Und all das soll editierbar bleiben, damit die Abfrage als Grundlage für ein Formular dienen kann.

	ID	Anzahl	warID	WarenID	Preis	Teilbetrag
▶	40	5	17	17	0,75 €	3,75
	41	1	0	0	4,23 €	4,23
	43	2	31	31	0,23 €	0,46
	44	3	24	24	1,27 €	3,81
	45	7	0	0	4,23 €	29,61
	46	4	24	24	1,27 €	5,08
✱	<AutoF					

Datensatz 1 von 6

```
SELECT
    "Abgang"."ID",
    "Abgang"."Anzahl",
    "Abgang"."warID",
    "Ware"."ID" AS "WarenID",
    "Ware"."Preis",
    "Anzahl" * "Preis" AS "Teilbetrag"
FROM "Abgang",
    "Ware"
WHERE "Abgang"."warID" = "Ware"."ID"
```

Abbildung 7: Abfrage über zwei Tabellen. Damit die Abfrage editierbar bleibt, muss aus beiden Tabellen der Primärschlüssel in der Abfrage enthalten sein.

Hinweis

Aufgrund des *Bugs 61871* aktualisiert Base leider den Teilbetrag nicht automatisch.

	ID	Anzahl	warID	Preis	Teilbetrag
▶	40	5	17	0,75	3,75
	41	1	0	4,23	4,23
	43	2	31	0,23	0,46
	44	3	24	1,27	3,81
	45	7	0	4,23	29,61
	46	4	24	1,27	5,08
✱	<AutoF				

Datensatz 1 von 6

```
SELECT
    "Abgang"."ID",
    "Abgang"."Anzahl",
    "Abgang"."warID",
    "Ware"."Preis",
    "Anzahl" * "Preis" AS "Teilbetrag"
FROM "Abgang",
        ( SELECT
            "ID",
            "Preis"
        FROM "Ware" )
    AS "Ware"
WHERE "Abgang"."warID" = "Ware"."ID"
```

Abbildung 8: Die Tabelle "Ware" wurde in eine Unterabfrage verschoben. Diese Unterabfrage wird im Tabellenbereich (nach dem Begriff «FROM») erstellt und mit einem Alias versehen. Jetzt ist der Primärschlüssel aus der Tabelle "Ware" in der Abfrage nicht mehr zwingend erforderlich. Die Abfrage bleibt auch so editierbar.

	recID	Summe
▶	0	12,25
	1	34,69

Datensatz 1 von 2

```
SELECT
    "Abgang"."recID",
    SUM("Anzahl" * "Preis") AS "Summe"
FROM "Abgang", "Ware"
WHERE "Abgang"."warID" = "Ware"."ID"
GROUP BY "Abgang"."recID"
```

Abbildung 9: Jetzt soll die Rechnungssumme noch in der Abfrage erscheinen. Bereits die einfache Abfrage der Rechnungssumme ist nicht editierbar, da hier gruppiert und summiert wird.

	ID	Anzahl	warID	Preis	Teilbetrag	Summe
▶	40	5	17	0,75	3,75	12,25
	41	1	0	4,23	4,23	12,25
	43	2	31	0,23	0,46	12,25
	44	3	24	1,27	3,81	12,25
	45	7	0	4,23	29,61	34,69
	46	4	24	1,27	5,08	34,69
✱	<AutoF					

Datensatz 1 von 6

```
SELECT
    "Abgang"."ID",
    "Abgang"."Anzahl",
    "Abgang"."warID",
    "Ware"."Preis",
    "Anzahl" * "Preis" AS "Teilbetrag",
    "Rechnungsbetrag"."Summe"
FROM "Abgang",
        ( SELECT
            "ID",
            "Preis"
        FROM "Ware" )
    AS "Ware",
        ( SELECT
            "Abgang"."recID",
            SUM( "Anzahl" * "Preis" ) AS "Summe"
        FROM "Abgang",
            "Ware"
        WHERE "Abgang"."warID" = "Ware"."ID"
        GROUP BY "Abgang"."recID" )
    AS "Rechnungsbetrag"
WHERE "Abgang"."warID" = "Ware"."ID"
        AND "Abgang"."recID" = "Rechnungsbetrag"."recID"
ORDER BY "Abgang"."recID", "Abgang"."ID"
```

Abbildung 10: Mit der zweiten Unterabfrage wird das scheinbar Unmögliche möglich. Die vorhergehenden Abfrage wird als Unterabfrage in der Tabellendefinition dieser Abfrage (nach «FROM») eingefügt. Im Ergebnis bleibt die gesamte Abfrage so editierbar. Eingaben sind in diesem Fall nur in den Spalten "Anzahl" und "warID" möglich. Dies wird im Formular anschließend berücksichtigt.

Zusammenfassung von Daten mit Abfragen

Sollen Daten in der gesamten Datenbank gesucht werden, so ist dies über die einfachen Formularfunktionen meist nur mit Problemen zu bewältigen. Ein Formular greift schließlich nur auf eine Tabelle zu, und die Suchfunktion bewegt sich nur durch die Datensätze dieses Formulars.

Einfacher wird der Zugriff auf alle Daten mittels Abfragen, die wirklich alle Datensätze abbilden. Im Kapitel *Beziehungsdefinition in der Abfrage* wurde so eine Abfragekonstruktion bereits angedeutet. Diese wird im Folgenden entsprechend der Beispieldatenbank ausgebaut.

```
001 SELECT
002    "Medien"."Titel",
003    "Untertitel"."Untertitel",
004    "Verfasser"."Verfasser"
005 FROM "Medien"
006    LEFT JOIN "Untertitel"
007       ON "Medien"."ID" = "Untertitel"."Medien_ID"
008    LEFT JOIN "rel_Medien_Verfasser"
009       ON "Medien"."ID" = "rel_Medien_Verfasser"."Medien_ID"
010    LEFT JOIN "Verfasser"
011       ON "rel_Medien_Verfasser"."Verfasser_ID" = "Verfasser"."ID"
```

Hier werden alle "Titel", "Untertitel" und "Verfasser" zusammen angezeigt.

Die Tabelle "Medien" hat insgesamt 9 "Titel". Zu zwei Titeln existieren insgesamt 8 "Untertitel". Beide Tabellen zusammen angezeigt ergäben ohne einen **LEFT JOIN** lediglich 8 Datensätze. Zu jedem "Untertitel" würde der entsprechende "Titel" gesucht und damit wäre die Abfrage zu Ende. "Titel" *ohne* "Untertitel" würden nicht angezeigt.

Jetzt sollen aber alle "Medien" angezeigt werden, auch die *ohne* "Untertitel". "Medien" steht auf der linken Seite der Zuweisung, "Untertitel" auf der rechten Seite. Mit einem **LEFT JOIN** werden alle "Titel" aus "Medien" angezeigt, aber nur die "Untertitel", zu denen auch ein "Titel" existiert. "Medien" wird dadurch zu der Tabelle, die entscheidend für alle anzuzeigenden Datensätze wird. Dies ist bereits aufgrund der Tabellenkonstruktion so vorgesehen (siehe dazu das Kapitel «Tabellen Medienaufnahme»). Da zu 2 der 9 "Titel" "Untertitel" existieren, erscheinen in der Abfrage jetzt 9 + 8 – 2 = 15 Datensätze.

Hinweis	Die normale Verbindung von Tabellen erfolgt, nachdem alle Tabellen durch Komma voneinander getrennt aufgezählt wurden, nach dem Schlüsselwort **WHERE**. Wird mit einem **LEFT JOIN** oder **RIGHT JOIN** gearbeitet, so wird die Zuweisung direkt nach den beiden Tabellennamen mit **ON** definiert. Die Reihenfolge ist also immer **Tabelle1 LEFT JOIN Tabelle2 ON Tabelle1.Feld1 = Tabelle2.Feld1 LEFT JOIN Tabelle3 ON Tabelle2.Feld1 = Tabelle3.Feld1 ...**

Zu 2 Titeln der Tabelle "Medien" existiert noch keine Verfassereingabe und kein "Untertitel". Gleichzeitig existieren bei einem "Titel" insgesamt 3 "Verfasser". Würde jetzt die Tabelle Verfasser einfach ohne **LEFT JOIN** verbunden, so würden die beiden "Medien" *ohne* "Verfasser" nicht angezeigt. Da aber ein Medium statt einem Verfasser drei Verfasser hat, würde die angezeigte Zahl an Datensätzen bei 15 Datensätzen bleiben.

Erst über die Kette von **LEFT JOIN** wird die Abfrage angewiesen, weiterhin die Tabelle "Medien" als den Maßstab für alles anzuzeigende zu nehmen. Jetzt erscheinen auch wieder die Datensätze, zu denen weder "Untertitel" noch "Verfasser" existieren, also insgesamt 17 Datensätze.

Durch entsprechende Joins wird also der angezeigte Datenbestand in der Regel größer. Durch diesen großen Datenbestand kann schließlich gesucht werden und neben den Titeln werden auch Verfasser und Untertitel erfasst. In der Beispieldatenbank können so alle von den Medien abhängigen Tabellen erfasst werden.

Hinweis	Die Abfrage-GUI ersetzt grundsätzlich **`LEFT JOIN`** durch **`LEFT OUTER JOIN`**. Sofern eine Abfrage mit der GUI und einem entsprechenden Join in der HSQLDB erstellt wurde, wird darüber hinaus der Join durch **`{ oj … }`** in Klammern gesetzt. ~~Firebird~~ kann Abfragen mit dieser Klammerung nicht lesen.

Schnellerer Zugriff auf Abfragen durch Tabellenansichten

Ansichten, in der SQL-Sprache **`View`**, sind, besonders bei externen Datenbanken, schneller als Abfragen, da sie direkt in der Datenbank verankert sind und vom Server nur das Ergebnis präsentiert wird. Abfragen werden hingegen erst einmal zum Server geschickt und dort dann verarbeitet.

Bezieht sich eine neue Abfrage auf eine andere Abfrage, so sieht das in Base in der SQL-Ansicht so aus, als ob die andere Abfrage eine Tabelle wäre. Wird daraus ein **`View`** erstellt, so zeigt sich, dass eigentlich mit Unterabfragen (Subselects) gearbeitet wird. Eine Abfrage 2, die sich auf eine andere Abfrage 1 bezieht, kann daher nicht über **SQL-Befehl direkt ausführen** ausgeführt werden, da nur die grafische Benutzeroberfläche, nicht aber die Datenbank selbst die Abfrage 1 kennt.

Auf Abfragen kann von der Datenbank her nicht direkt zugegriffen werden. Dies gilt auch für den Zugriff über Makros. Views hingegen können von Makros wie Tabellen angesprochen werden. Allerdings können in Views keine Datensätze geändert werden. Diesen Komfort bietet nur die Abfrage unter bestimmten Abfragebedingungen.

Tipp	Eine Abfrage, die über **SQL-Befehl direkt ausführen** gestartet wird, hat den Nachteil, dass sie über die GUI nicht mehr sortiert und gefiltert werden kann. Sie kann also nur begrenzt genutzt werden. Eine Tabellenansicht (*View)* hingegen ist für Base handhabbar wie eine normale Tabelle – mit der Ausnahme, dass keine Änderung der Daten möglich ist. Hier stehen also trotz direktem SQL-Befehl alle Möglichkeiten zur Sortierung und zur Filterung zur Verfügung. Auch bleibt die Formatierung von Spalten in der Ansicht im Gegensatz zu Spalten in der Abfrage beim Schließen der Datenbank erhalten. Auch zur Nutzung innerhalb von Berichten (siehe Kapitel «Berichte») funktioniert eine Tabellenansicht deutlich besser, da Funktionen und Aliasformulierungen die Interpretation des Codes innerhalb des Berichtes nicht beeinflussen.

Ansichten sind bei manchen Abfragekonstruktionen eine Lösung, um überhaupt ein Ergebnis zu erzielen. Wenn z. B. auf das Ergebnis eines Subselects zugegriffen werden soll, so lässt sich dies nur über den Umweg eines Views erledigen. Entsprechende Beispiele im Kapitel «Datenbank-Aufgaben komplett»: «Zeilennummerierung» und «Gruppieren und Zusammenfassen».

Hinweis

Ansichten können bei der internen Firebird-Datenbank erst ab Version LO 7.3.1 in der GUI bearbeitet und geändert werden. Vorher war der SQL-Code der Ansicht nicht mehr sichtbar, nachdem die Ansicht erstellt wurde. Um den SQL-Code aller Ansichten in Firebird vor LO 7.3.1 zu erhalten muss der Abfrageeditor im SQL-Modus geöffnet und die folgende Abfrage gestartet werden:

```
001 SELECT RDB$RELATION_NAME, RDB$VIEW_SOURCE FROM RDB$RELATIONS
    WHERE RDB$VIEW_SOURCE IS NOT NULL
```

Auch die Ansichten in anderen Datenbanken können nicht nachträglich in der GUI bearbeitet werden. Der SQL-Code wird hier nur dann sichtbar, wenn der eingerichtete Nutzer auch eine Berechtigung dazu hat. In MySQL/MariaDB lautet der Code dazu folgendermaßen:

```
001 SELECT TABLE_SCHEMA, TABLE_NAME, VIEW_DEFINITION FROM
    INFORMATION_SCHEMA.VIEWS
```

Bei PostgreSQL sind je nach verwendetem Treiber Ansichten gar nicht direkt erstellbar. Mit dem JDBC-Treiber ergibt sich folgendes Kommando, um die selbst definierten Ansichten zu erhalten:

```
001 SELECT views.table_catalog, views.table_schema,
    views.table_name, views.view_definition FROM
    information_schema.views AS views WHERE NOT
    views.view_definition IS NULL
```

Ansichten passen sich nicht automatisch an, wenn einer Tabelle z.B. ein zusätzliches Feld hinzugefügt wird. In einer Ansicht für eine Tabelle wird **nicht**

```
001 SELECT * FROM "Tabelle"
```

gespeichert. Dort tauchen fest die Feldbezeichnungen auf:

```
001 SELECT "ID", "Vorname", "Nachname" … FROM "Tabelle"
```

Deswegen kann auch kein Feld aus einer Tabelle entfernt werden, wenn es in einer Ansicht benötigt wird.

Ansichten können direkt im Tabellenordner erstellt werden. Es ist auch möglich erst eine Abfrage zu erstellen und dann über die rechte Maustaste im Kontextmenü **Als Ansicht erstellen** zu wählen. Dann erscheint ein kleiner Dialog zur Benennung der Ansicht. In Firebird wird nach dem Beenden des Dialogs mit OK bis LO 7.3.1 noch eine Fehlermeldung ausgegeben. Die Ansicht erscheint dann nicht direkt in der Tabellenübersicht. Nach Auswahl von **Ansicht → Tabellen aktualisieren** im Tabellenordner ist sie aber sichtbar und wurde offensichtlich einwandfrei erstellt.

Zeitdifferenzen berechnen

Eine Tabelle hat zwei Zeitangaben: Startzeit und Zielzeit. Die Zeitdifferenz soll berechnet werden.

```
001 SELECT
002    "Startzeit",
003    "Zielzeit",
004    CAST('00:00:'||DATEDIFF('ss',"Startzeit", "Zielzeit") AS TIME)
          AS "Zeitdifferenz"
005 FROM "Zeitmessung"
```

Hier wird mit dem Trick gearbeitet, dass die HSQLDB auch Zeitangaben umwandeln kann, die von der Textdarstellung her keine Zeitangaben mehr wären. So ergibt die erste Zeitdifferenz erst einmal über die Verbindung von

```
004    CAST('00:00:'||DATEDIFF('ss',"Startzeit", "Zielzeit") AS TIME)
```

den Text

```
004    00:00:3397
```

der anschließend anstandslos in eine korrekte Zeit von 56 Minuten und 37 Sekunden (3397/60 = 56 Rest 37) umgewandelt wird.

Startzeit	***Zielzeit***	***Zeitdifferenz***
12:13:34	13:10:11	00:56:37
12:01:23	14:08:13	02:06:50

Leider funktioniert die Darstellung der Zeit in einer Abfrage nur dann korrekt, wenn die Spalte nach Durchführung der Abfrage auf die entsprechende Formatierung umgewandelt wird. Dieses Problem stellt sich in Formularen und Berichten nicht, da dort die Formatierung dauerhaft gespeichert wird. Soll anschließend nicht mehr mit der Zeitdifferenz gerechnet werden, so kann die Zeitdifferenz in einen Text mit der korrekten Schreibweise umgewandelt werden:

```
001 SELECT
002    "Startzeit",
003    "Zielzeit",
004    TO_CHAR(
          CAST('00:00:'||DATEDIFF('ss',"Startzeit", "Zielzeit") AS TIME),
          'HH:MI:SS') AS "Zeitdifferenz"
005 FROM "Zeitmessung"
```

Mit der Firebird-Datenbank lässt sich dieses Problem wesentlich eleganter lösen:

```
001 SELECT
002    "Startzeit",
003    "Zielzeit",
004    "Zielzeit" - "Startzeit" AS "Zeitdifferenz"
005 FROM "Zeitmessung"
```

Firebird hat mit der Funktion DATEADD den Vorteil, dass es viele verschiedene Additions- und Subtraktionsverfahren für Datums- und Zeitwerte kennt.