Kapitel 9

Das QtSql-Modul

Ohne relationale Datenbank im Rücken sind viele Applikationen kaum mehr vorstellbar. Qt bietet daher mit dem QtSql-Modul eine Reihe von Klassen an, die mit verschiedenen relationalen Datenbankmanagementsystemen (DBMS) zusammenarbeiten. Tabellen und Abfragen lassen sich auch als Modell für Interview nutzen.

9.1 Aufbau des QtSql-Moduls

Beim QtSql-Modul handelt es sich um eine eigenständige Bibliothek, die bei Bedarf zusätzliche Plugins nachlädt. Anders als QtCore und QtGui bindet **qmake -project** sie nicht standardmäßig in die generierten Projekte ein. Um die Bibliothek zu nutzen, ist daher folgender Eintrag in der **.pro**-Datei erforderlich:

```
QT += sql
```

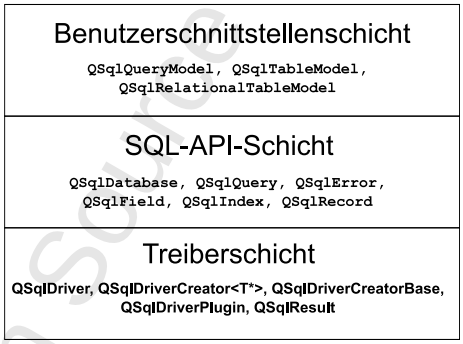
Um mit den Klassen des Moduls zu arbeiten, stellt Qt auch für dieses Paket ein Meta-Include zur Verfügung, das alle Klassendefinitionen aus dem Modul enthält. Das Kommando zum Einbinden in eine Quelldatei lautet:

```
#include <QtSql>
```

Die Klassen des Moduls gehören zu einer von drei Schichten: Die *Treiber-Schicht* implementiert die Schnittstelle zwischen den Treibern für verschiedene Datenbanken (siehe Tabelle 9.1) und der *API-Schicht*. Diese bietet Anwendungsentwicklern Zugriff auf die Datenbanken und erlaubt typische SQL-Operationen, etwa Tabellen einzusehen und zu verändern oder Daten abzufragen.

Um Ergebnisse von Abfragen in Interview-Ansichten einzufügen, stellt die *Benutzerschnittstellenschicht* Modelle zur Verfügung, denen SQL-Tabellen oder -Anfragen zu Grunde liegen. Abbildung 9.1 gibt eine Übersicht über die Schichten und die dazugehörigen Klassen.

Abbildung 9.1: Der Aufbau des QtSql-Moduls



9.2 Den passenden Treiber wählen

Da die Lizenz der Client-API mancher Datenbanksysteme nicht GPL-kompatibel ist, fehlen einige Treiber (in Tabelle 9.1 mit *) markiert) in der Open-Source-Edition.

Tabelle 9.1: Treiber für QtSql

Treibername	Datenbanksystem
QDB2	IBM DB2 (Version 7.1 und neuer)*)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle Call Interface Treiber (Versionen 8, 9, and 10)*)

Fortsetzung:

Treibername	Datenbanksystem
QODBC	Open Database Connectivity (ODBC), verwendet von Microsoft SQL Server und anderen ODBC-fähigen Datenbanken
QPSQL	PostgreSQL (Version 7.3 und neuer)
QSQLITE2	SQLite (Version 2)
QSQLITE	SQLite (Version 3)
QTDS	Sybase Adaptive Server*)

Stammt die Qt-Version aus Paketen eines Linux-Distributors, müssen gegebenenfalls Pakete nachinstalliert werden. Ubuntu lagert die SQL-Bibliothek in das Paket **libqt4-sql** aus, OpenSUSE erfordert zusätzlich zur Installation von **qt-sql** ein DBMS-spezifisches Datenbankpaket, für MySQL etwa **qt-sql-mysql**.

Wer Qt aus den Quellen baut, sollte sich die Ausgabe von **./configure --help** anschauen:

```
...
-Istring ........... Add an explicit include path.
...
-qt-sql-<driver> ...... Enable a SQL <driver> in the Qt Library, by
                        default none are turned on.
-plugin-sql-<driver> .. Enable SQL <driver> as a plugin to be linked
                        to at run time.
-no-sql-<driver> ...... Disable SQL <driver> entirely.

                        Possible values for <driver>:
                        [ db2 ibase mysql oci odbc psql sqlite
                          sqlite2 tds ]

                        Auto-Detected on this system:
                        [ sqlite ]
...
```

Qt baut per Default die Treibermodule als *Plugins* für alle automatisch gefundenen Systeme – hier für SQLite. Möchte man eines davon explizit nicht kompilieren, kommt der **-no-sql-***driver*-Schalter zum Einsatz, im Falle von SQLite also zum Beispiel **-no-sql-sqlite**. Qt bringt übrigens eine eigene SQLite-Version mit. Wer stattdessen ein auf dem System installiertes SQLite verwenden möchte, muss den Schalter **-system-sqlite** mit angeben.

Findet **./configure** ein installiertes Datenbanksystem trotz installierter Entwicklungspakete nicht, gibt man mit dem Schalter **-I** das Include-Verzeichnis des Datenbanksystems mit, etwa **-I/usr/include/mysql** im Falle von MySQL. Ob ein Treiber separat als Plugin gebaut (**-plugin-sql-***driver*) oder fest in die Bibliothek einkompiliert wird (**-qt-sql-***driver*), bleibt jedem selber überlassen. Plugins sind flexibler,

einkompilierte Treiber einfacher zu handhaben, wenn die Qt-Bibliothek mit dem Programm mitgeliefert werden soll.

9.3 Verbindung aufnehmen

Die Klasse **QSqlDatabase** kontrolliert den Kontakt mit dem Server, ihre statische Methode **addDatabase()** liefert eine **QSqlDatabase**-Instanz:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL");
```

Als Argument erwartet sie mindestens den Namen des Datenbanktreibers in Stringform, also etwa **"QPSQL"** für den Postgres-Treiber. Eine in dieser Form erstellte Datenbank-Instanz dient als Standardverbindung. Wenn das Programm zu mehr als einer Datenbank Kontakt aufnehmen muss, benötigt die Methode zusätzlich einen Verbindungsnamen:

```
QSqlDatabase webdb =
        QSqlDatabase::addDatabase("QMYSQL", "WebServerDB");
QSqlDatabase personaldb =
        QSqlDatabase::addDatabase("QOCI", "PersonalDB");
QSqlDatabase embeddeddb =
        QSqlDatabase::addDatabase("QSQLITE", "EmbeddedDB");
```

Fehlte dieser im obigen Beispiel, verwiesen alle drei **QSqlDatabase**-Instanzen auf die SQLite-Datenbank, denn jeder **addDatabase()**-Aufruf ohne zusätzlichen Parameter modifiziert die Standardverbindung.

Im folgenden Beispiel reicht uns die Verbindung zu einem einzigen MySQL-Server: Mit einem **QSqlDatabase**-Objekt mit passendem Treiber zur Hand stellen wir eine Verbindung zur Datenbank her. Dazu geben wir den Servernamen, den Namen der Datenbank, den Benutzernamen und das Passwort an:

```
// sqlexample/main.cpp

#include <QtGui>
#include <QtSql>
#include <QDebug>

int main(int argc, char* argv[])
{
  QApplication app(argc, argv);

  QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
  db.setHostName("datenbankserver.example.com");
  db.setDatabaseName("firma");
  db.setUserName("user");
```

```
db.setPassword("pass");

if (!db.open()) {
   qDebug() << db.lastError();
   return 1;
}
```

Die **open()**-Methode stellt mit diesen Zugangsdaten die Verbindung zur Datenbank her. Ob der Verbindungsaufbau erfolgreich war, verrät sie durch ihren booleschen Rückgabewert. Im Fehlerfalle erfahren wir mit **lastError()** den Grund für das Scheitern des Verbindungsaufbaus. Die Methode liefert ein Objekt vom Typ **QSqlError**, das **qDebug()** auslesen kann. Will man den Fehler anderweitig verwerten, hilft die **QSqlError**-Klassenmethode **text()**.

9.4 Anfragen stellen

In unseren folgenden Beispielen arbeiten wir mit zwei Tabellen: Die **mitarbeiter**-Tabelle hält Informationen zu den Mitarbeitern einer Firma bereit (Tabelle 9.2), während **abteilungen** (9.3) die verschiedenen Organisationseinheiten im Unternehmen betrachtet.

id	nachname	vorname	abteilung
1	Werner	Max	1
2	Lehmann	Daniel	2
3	Roetzel	David	1
4	Scherfgen	David	2
5	Scheidweiler	Najda	2
6	Jüppner	Daniela	4
7	Hasse	Peter	4
8	Siebigteroth	Jennifer	3

Tabelle 9.2: Die Tabelle mitarbeiter *aus der Beispieldatenbank*

id	name
1	Geschäftsführung
2	Entwicklung
3	Marketing
4	Buchhaltung

Tabelle 9.3: Tabelle abteilungen *aus der Beispieldatenbank*

Für Anfragen an die Datenbank verwenden wir die Klasse **QSqlQuery**. Bekommt eine Klasse im Konstruktor ein SQL-Kommando als String übergeben, führt das instanzierte Objekt die Anfrage sofort aus. Die im Query-Objekt gelagerte Abfrage lässt sich (etwa nach einer Änderung) später erneut mit **exec()** starten. Hat man mehrere Verbindungen geöffnet, akzeptiert die Klasse eine **QSqlDatabase**-Instanz als zweiten Parameter.

War die Anfrage erfolgreich, gilt sie als aktiv, was sich mittels **isActive()** überprüfen lässt. Hat die **QSqlQuery** beispielsweise durch eine **SELECT**-Anfrage Datensätze gesammelt, kann man in ihnen navigieren: **first()** springt zum ersten Datensatz, **last()** zum letzten, **next()** zum jeweils nächsten und **previous()** zum vorhergehenden Datensatz. Mit **seek()** geht man durch Angabe eines ganzzahligen Indexes gezielt zu einem Datensatz. Die Anzahl der im Objekt enthaltenen Datensätze verrät **size()**.

Die **record()**-Methode gibt ein **QSqlRecord**-Objekt zurück. Sie enthält Informationen zur Antwort auf eine **SELECT**-Anfrage. Auf diesem Wege bringen wir z. B. per **indexOf()** die Nummer einer Spalte aus dem Abfrageergebnis in Erfahrung. Diese nutzen wir, um den Wert in der entsprechenden Spalte mit **QSqlQuery::value()** auszulesen. Welche Zeile dabei ausgewählt ist, hängt von der Position ab, die wir im Beispiel mit **next()** verändern und mit **at()** in Erfahrung bringen:

```
// sqlexample/main.cpp (fortgesetzt)


  QSqlQuery query("SELECT vorname, nachname FROM mitarbeiter");
  QSqlRecord record = query.record();
  while (query.next()) {
    QString  vorname =
       query.value(record.indexOf("vorname")).toString();
    QString  nachname =
       query.value(record.indexOf("nachname")).toString();
    qDebug() << query.at() << ":" << nachname << "," << vorname;
  }
```

Bei Anfragen, welche Daten verändern (beispielsweise **UPDATE** oder **DELETE**), gibt **numRowsAffected()** die Anzahl der betroffenen Datensätze zurück:

```
// sqlexample/main.cpp (fortgesetzt)

 query.exec("DELETE FROM mitarbeiter WHERE nachname = 'Hasse'");
 qDebug() << query.numRowsAffected(); // "1"
```

Etwas komplizierter wird es bei **INSERT**-Anweisungen. Da man damit Werte aus programmeigenen Datenstrukturen in die Datenbank schreibt, lässt sich die entsprechende SQL-Anweisung nur aufwändig als String formulieren und direkt ausführen: Deshalb geht man hier einen anderen Weg: Wir speichern die mit Platzhaltern ausgestattete Anfrage mit **prepare()** im Query-Objekt:

```
// sqlexample/main.cpp (fortgesetzt)

  query.prepare("INSERT INTO mitarbeiter (nachname, vorname, abteilung)"
                "VALUES(:nachname, :vorname, :abteilung)");
  query.bindValue(":nachname", "Hasse");
  query.bindValue(":vorname", "Peter");
  query.bindValue(":abteilung", 3);
  query.exec();
```

Die aus der Oracle-Welt stammenden *benannten Platzhalter* im VALUES-Teil des SQL-Kommandos beginnen jeweils mit einem Doppelpunkt. Mit dem Kommando bindValue() ersetzen wir sie durch die konkreten Werte.

Auch mit den aus ODBC bekannten *unbenannten Parametern* kann QSqlQuery umgehen. Jeder bindValue()-Aufruf ersetzt der Reihe nach eines der Fragezeichen:

```
// sqlexample/main.cpp (fortgesetzt)

  query.prepare("INSERT INTO mitarbeiter (nachname, vorname, abteilung)"
                "VALUES(?, ?, ?");
  query.bindValue("Schwan");
  query.bindValue("Waldemar");
  query.bindValue(3);
  query.exec();
```

Möchte man die Werte nicht der Reihenfolge nach ersetzen, nutzt man folgende überladene Variante:

```
  query.bindValue(3, 3);
  query.bindValue(1, "Schwan");
  query.bindValue(2, "Waldemar");
```

Hier gibt der erste Parameter die zu ersetzende Position, also das zu ersetzende Fragezeichen, im prepare()-String an.

Auch bei gespeicherten Prozeduren (*Stored Procedures*) spielt bindValue() eine zentrale Rolle, denn deren Parameter können sowohl als IN als auch als OUT deklariert sein. OUT deklarierte Parameter fungieren als Rückgabewert.

Um an diesen Rückgabewert zu kommen, müssen wir die bindValue()-Methode anpassen: Der übergebene Wert spielt hier keine Rolle, er wird später durch den OUT-Wert überschrieben. Wichtig ist die Angabe QSql::Out, die QSqlQuery bedeutet, den Wert zu überschreiben. Nachdem wir exec() ausgeführt haben, liegt der Wert an der entsprechenden Position. Diese erfragen wir mit boundValue():

```
// sqlexample/main.cpp (fortgesetzt)

  query.prepare("CALL zaehlePersonal(?)");
  query.bindValue(0, 0, QSql::Out);
```

```
  query.exec();
  qDebug() << query.boundValue(0).toInt()
```

Leider funktioniert dieser Ansatz aufgrund von API-Limitierungen bei MySQL 5 nicht korrekt. Um dort an die **OUT**-Werte zu kommen, müssen wir manuell zwei Anfragen absetzen: Zunächst führen wir die gespeicherte Prozedur mit **CALL** aus und lesen dann den ausgegebenen Wert mittels **SELECT** ein. Um uns auf den Wert zu beziehen, verwenden wir jeweils einen Platzhalter mit **@** als MySQL-spezifisches Präfix, um den Rückgabewert der gespeicherten Prozedur als Datensatz auszulesen:

```
// sqlexample/main.cpp (fortgesetzt)

  query.exec("CALL zaehlePersonal (@outwert)");
  query.exec("SELECT @outwert");
  query.next();
  qDebug() << query.value(0);
  return 0;
}
```

9.5 Transaktionen

Bei weitem nicht alle Datenbanksysteme unterstützen Transaktionen, die mehrere SQL-Operationen zu einer atomaren Operation zusammenfassen. Um den Code portabel zu halten, befragt man den Treiber daher mit **hasFeature()** unter anderem zu seiner Transaktionsfähigkeit:

```
if (db.driver()->hasFeature(QSqlDriver::Transactions)) ... ;
```

Unterstützt er Transaktionen, kann man sie mit der **QSqlDatabase**-Methode **transaction()** einleiten. Sind alle Operationen abgeschlossen, schließt man die Transaktion mit **commit()** ab. Trat ein Fehler auf, macht **rollback()** alle Operationen der aktuellen Transaktion ungeschehen.

9.6 Eingebettete Datenbanken

Qts SQLite-Treiber ermöglicht es, Daten auch ohne externen Datenbankserver in einer relationalen Datenbank zu speichern und diese dann abzufragen – natürlich mit Einschränkungen, aber die Ansprüche an Embedded-Datenbanken sind gewöhnlich nicht die gleichen wie an unternehmensweite Datenbankserver. So kann SQLite nicht mit gespeicherten Prozeduren umgehen und skaliert nicht so gut wie ihre großen Brüder. Dennoch eignet es sich gut für Anwendungen, die einen relationalen Datenspeicher brauchen. Ein Paradebeispiel ist der KDE-Musikspieler Amarok, der die Metadaten zu den Musikstücken in einer SQLite-Datenbank ablegt.

Um eine Verbindung zu einer SQLite-Datenbank zu öffnen, ist nur die Angabe eines Datenbanknamens erforderlich. Der SQLite-Treiber erwartet hier einen Dateinamen:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName("firma.db");
```

Soll die Datenbank nur während des Programmlaufs im Speicher liegen, generiert man durch Einschließen des Datenbanknamens in Doppelpunkte eine temporäre Datenbank. Diese wird – wie die Datenbank resultate im folgenden Beispiel – nicht auf der Platte gespeichert:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName(":resultate:");
```

Mit dieser Datenbank kann man wie gewohnt arbeiten, einzig die Änderungen gehen später verloren. Eine temporäre Datenbank kann eigene Datenstrukturen sparen, wenn die Daten ohnehin relationaler Natur sind.

9.7 SQL-Modell-Klassen mit Interview verwenden

Um den Inhalt von Datenbanken darzustellen, eignen sich zumeist Tabellen, in einigen Fällen auch Listenansichten. Aus diesem Grund besitzt das QtSql-Modul eine Reihe von Modellen für Interview (siehe Kapitel 8 ab Seite 205).

9.7.1 SQL-Tabellen ohne Fremdschlüssel in Tabellen- und Baumansichten darstellen

QSqlTableModel erlaubt es, komplette Tabellen direkt in einer Tabellen- oder Baumansicht darzustellen. Die Spaltenüberschriften entsprechen den Feldnamen (Attributen, Spalten) der SQL-Tabelle. In unserer Personendatenbank aus Tabelle 9.2 auf Seite 259 sind dies id, vorname, nachname und abteilung. Jede Zeile entspricht einem Datensatz. Zur besseren Vorstellung betrachten wir folgendes Beispiel, das von einer geöffneten Standardverbindung ausgeht:

```
// sqlmvd/main.cpp

...
  QTableView tableView;
  QSqlTableModel tableModel;
  tableModel.setTable("mitarbeiter");
  tableModel.select();
  tableModel.removeColumn(0);
  tableView.setModel(&tableModel);
```

```
tableView.setWindowTitle("Tabelle 'mitarbeiter'");
tableView.show();
```

Zunächst erstellen wir eine Tabellenansicht, danach das Modell. Ihm weisen wir eine Tabelle aus der aktuellen Datenbank zu und beauftragen es mit **select()**, die Daten zu holen. Anschließend entfernen wir die Spalte **id** mit **removeColumn()** aus der Ansicht (Abbildung 9.2). Diese Methode stammt aus **QAbstractItemModel**, dem Urobjekt aller Modelle. Schließlich übergeben wir das Modell an die Tabellenansicht, geben der Tabelle einen Titel und zeigen sie an.

Abbildung 9.2: QTableModel *ist in Interview für SQL-Tabellen zuständig*

9.7.2 Fremdschlüssel-Relationen auflösen

QSqlRelationalTableModel erweitert diesen Ansatz: Objekte dieser Klasse lösen zusätzlich Fremdschlüssel-Relationen auf. Damit können wir die nichtssagende Zahl im Feld **abteilung** durch den Namen der Abteilung ersetzen, indem wir die Tabelle **abteilungen** (vgl. Tabelle 9.3 auf Seite 259) heranziehen.

Abbildung 9.3: QSqlRelationalTableModel *löst das Fremdschlüssel-Feld* id *mit Hilfe einer zweiten Tabelle auf*

Zur Beschreibung dieser Relation dient die Methode **setRelation()**: Sie erwartet die Nummer der Spalte, die die Fremdschlüssel enthält, als erstes Argument. In unserem Beispiel soll an die Stelle des Fremdschlüssels, also der **id** aus der Tabelle **abteilungen**, das Feld **name** treten.

Diese Information kapselt die Hilfsklasse QSqlRelation, die wir als zweites Argument an setRelation() übergeben. Nun können wir die Abfrage per select() starten und das Modell wie schon im vorherigen Beispiel an die Ansicht übergeben:

```
// sqlmvd/main.cpp (fortgesetzt)

  QTableView tableRelationalView;
  QSqlRelationalTableModel tableRelationalModel;
  tableRelationalModel.setTable("mitarbeiter");
  QSqlRelation rel("abteilungen", "id", "name");
  tableRelationalModel.setRelation(3, rel);
  tableRelationalModel.select();
  tableRelationalView.setModel(&tableRelationalModel);
  tableRelationalView.setItemDelegate(
                new QSqlRelationalDelegate(&tableRelationalView));
  tableRelationalView.setWindowTitle(
               "Tabelle mit aufgelösten Relationen");
  tableRelationalView.show();
```

Nun folgt jedoch eine Besonderheit, die nur in Zusammenhang mit QRelationalTableModel funktioniert: Ein spezieller Delegate namens QSqlRelationalDelegate erlaubt es dem Anwender beim Editieren von Spalten, auf die eine Relation definiert ist, den Wert aus einer Liste auszuwählen (Abbildung 9.3). Diese stellt er selbständig aus der verwendeten QSqlRelation zusammen: Im Beispiel bezieht er seine Vorschläge aus der name-Spalte, der in die Tabelle zurückgeschriebene Wert stammt hingegen aus der Spalte id.

9.7.3 Abfrageergebnisse darstellen

Um die Ergebnisse insbesondere komplexer SELECT-Abfragen darzustellen, die sich nicht einfach mit einem Filter auf einem QSqlTableModel modellieren lassen, greift man auf das QSqlQueryModel zurück. Das folgende Beispiel wertet aus, wieviele Mitarbeiter jede der Abteilungen im Unternehmen hat. Außerdem sollen die Spalten, wie in Abbildung 9.4 zu sehen, aussagekräftige Namen tragen.

Abbildung 9.4: QSqlQueryModel *dient als Interview-Datenquelle für Abfragen aller Art*

Nach der Instanzierung des Modells übergeben wir die Anfrage als String an **setQuery()**. Alternativ käme auch ein **QSqlQuery**-Objekt in Frage.

Da bei komplexeren Anfragen Fehler auftreten können, sollten wir spätestens hier eine Fehlerüberprüfung einführen. **lastError()** gibt den letzten vom SQL-Server gemeldeten Fehler in einem **QSqlError**-Objekt zurück. Ist dieses gültig, ist ein Fehler aufgetreten, den wir mit **qDebug()** ausgeben:

```
// sqlmvd/main.cpp (fortgesetzt)

  QTableView queryView;
  QSqlQueryModel queryModel;
  queryModel.setQuery("SELECT abteilungen.name, "
        "COALESCE(COUNT(mitarbeiter.nachname),0) "
        "FROM abteilungen LEFT JOIN mitarbeiter "
        "ON mitarbeiter.abteilung = abteilungen.id "
        "GROUP BY mitarbeiter.abteilung");

  if (queryModel.lastError().isValid())
    qDebug() << queryModel.lastError();

  queryModel.setHeaderData(0, Qt::Horizontal,
                           QObject::tr("Abteilung"));
  queryModel.setHeaderData(1, Qt::Horizontal,
                           QObject::tr("Anzahl Mitarbeiter"));
  queryView.setModel(&queryModel);
  queryView.setWindowTitle("Mitarbeiteranzahl nach Abteilungen");
  queryView.show();
```

Zu benutzerfreundlichen Spaltenüberschriften kommen wir, indem wir mit **setHeaderData()** die beiden ersten Spaltenüberschriften ersetzen.[1] Anschließend übergeben wir das Modell an die Ansicht und zeigen sie, wie gehabt mit einer angepassten Überschrift, an.

9.7.4 Editierstrategien

All diese Tabellenmodelle sind beschreibbar. Doch zu welchem Zeitpunkt das Modell die Daten in die Datenbank zurückschreibt, haben wir noch nicht näher betrachtet.

QSqlTableModel und **QSqlRelationalTableModel** kennen drei *Editierstrategien*, die man den Modellen mittels **setEditStrategy()** zuweist:

SqlTableModel::OnRowChange
: ist der Standard in allen Modellen. Wenn diese Strategie aktiv ist, sendet das

[1] Natürlich lässt sich dies auch mit der SQL-Anweisung **AS** bewerkstelligen, allerdings muss man dann per **tr()** dafür sorgen, dass die Anfrage internationalisierbar ist, sonst lassen sich die Spaltenüberschriften nicht in andere Sprachen übertragen.

Modell ein UPDATE für den Datensatz, sobald der Benutzer einen anderen Datensatz, also eine andere Zeile in der Ansicht, anwählt.

SqlTableModel::OnFieldChange
: überträgt jede Änderung unmittelbar, nachdem der Benutzer ein Datum in einem Feld geändert hat, in die Datenbank.

SqlTableModel::OnManualSubmit
: speichert alle Änderungen im Modell zwischen, bis entweder der Slot submitAll(), der alle Änderungen an die Datenbank übermittelt, oder der Slot revertAll() ausgelöst wird. Letzterer verwirft alle gecachten Daten und stellt den Status aus der Datenbank wieder her (siehe dazu jedoch auch Kapitel 9.7.5 auf Seite 268).

Letzteres Szenario stellen wir nach, indem wir das Beispiel von Seite 263 so abändern, dass es zusätzlich zwei Schaltflächen erhält, die in einem Layout unter der Tabellenansicht angeordnet sind. Die anderen Befehle übernehmen wir 1:1:

```
// sqlmvd/main.cpp (fortgesetzt)

  QWidget w;
  QPushButton *submitPb = new QPushButton(
                QObject::tr("Änderungen speichern"));
  QPushButton *revertPb = new QPushButton(
                QObject::tr("Änderungen zurückrollen"));
  QGridLayout *lay = new QGridLayout(&w);
  QTableView *manualTableView = new QTableView;
  lay->addWidget(manualTableView, 0, 0, 1, 2);
  lay->addWidget(submitPb, 1, 0);
  lay->addWidget(revertPb, 1, 1);
  QSqlTableModel manualTableModel;
  manualTableModel.setTable("mitarbeiter");
  manualTableModel.select();
  manualTableModel.setEditStrategy(
                QSqlTableModel::OnManualSubmit);
  manualTableView->setModel(&manualTableModel);
  QObject::connect(submitPb, SIGNAL(clicked(bool)),
                &manualTableModel, SLOT(submitAll()) );
  QObject::connect(revertPb, SIGNAL(clicked(bool)),
                &manualTableModel, SLOT(revertAll()) );
  w.setWindowTitle("Manuell rücksetzbare Tabelle");
  w.show();

  return app.exec();
}
```

Nachdem wir die Editierstrategie auf OnManualSubmit umgestellt haben, fügen wir noch zwei Signal-Slot-Verbindungen ein: Ein Klick auf den submitPb-Knopf ruft den submitAll()-Slot auf, während revertPb revertAll() auslöst.

Abbildung 9.5: Mit der Editierstrategie OnManualSubmit *übermittelt man lokale Änderungen zu einem beliebigen Zeitpunkt an die Datenbank*

Nun dürfen wir nicht vergessen, das Hauptwidget **w** als neues Top-Level-Widget anzuzeigen. Das Ergebnis illustriert Abbildung 9.5.

9.7.5 Fehler der Tabellenmodelle

Einige Probleme, die im Zusammenhang mit den Tabellenmodellen in Qt 4.1 auftreten, sollen an dieser Stelle nicht unerwähnt bleiben: Zum einen funktionieren Editoroperationen nach dem Entfernen von Spalten nicht zuverlässig. Das **QSqlRelationalTableModel** ignoriert die **removeColumn()**-Anweisung sogar gänzlich. Als Workaround empfiehlt sich hier ein Proxymodell, das die unerwünschten Datensätze herausfiltert. Sollen die Daten nur angezeigt werden, kann man stattdessen einfach eine SQL-Anfrage über das **QSqlQueryModel** stellen.

Ein weiteres Problem betrifft den Slot **revertAll()**, der bei relationalen Tabellen mit der Editierstrategie **OnManualSubmit** alle Änderungen rückgängig machen soll. Auf den Spalten, auf denen zuvor mit **setRelation()** eine Fremdschlüsselbeziehung definiert wurde, setzt **revertAll()** die Werte jedoch nicht zurück. Hier hilft bislang nur, den Slot der Schaltfläche mit einem selbstgeschriebenen Slot zu verbinden, der das aktuelle Modell gegen ein neues mit gleichen Eigenschaften austauscht. Da das Modell die Daten zwischenspeichert, gehen sie auf diese Weise verloren, und das neue Modell zeigt wieder die Originaldaten aus der Datenbank.