Executing SQL Statements

The QSqlQuery class provides an interface for executing SQL statements and navigating through the result set of a query.

The QSqlQueryModel and QSqlTableModel classes described in the next section provide a higher-level interface for accessing databases. If you are unfamiliar with SQL, you might want to skip directly to the next section (Using the SQL Model Classes).

Executing a Query

To execute an SQL statement, simply create a QSqlQuery object and call QSqlQuery::exec() like this:

```
QSqlQuery query;
query.exec("SELECT name, salary FROM employee WHERE salary > 50000");
```

The QSqlQuery constructor accepts an optional QSqlDatabase object that specifies which database connection to use. In the example above, we don't specify any connection, so the default connection is used.

If an error occurs, exec() returns false. The error is then available as QSqlQuery::lastError().

Navigating the Result Set

QSqlQuery provides access to the result set one record at a time. After the call to exec(), QSqlQuery's internal pointer is located one position *before* the first record. We must call QSqlQuery::next() once to advance to the first record, then next() again repeatedly to access the other records, until it returns false. Here's a typical loop that iterates over all the records in order:

```
while (query.next()) {
    QString name = query.value(0).toString();
    int salary = query.value(1).toInt();
    qDebug() << name << salary;
}
```

The QSqlQuery::value() function returns the value of a field in the current record. Fields are specified as zero-based indexes. QSqlQuery::value() returns a QVariant, a type that can hold various C++ and core Qt data types such as int, QString, and QByteArray. The different database types are automatically mapped into the closest Qt equivalent. In the code snippet, we call QVariant::toString() and QVariant::toInt() to convert variants to QString and int.

For an overview of the recommended types for use with Qt-supported Databases, please refer to this table.

You can navigate within the dataset using QSqlQuery::next(), QSqlQuery::previous(), QSqlQuery::first(), QSqlQuery::last(), and QSqlQuery::seek(). The current row index is returned by QSqlQuery::at(), and the total number of rows in the result set is available as QSqlQuery::size() for databases that support it.

To determine whether a database driver supports a given feature, use QSqlDriver::hasFeature(). In the following example, we call QSqlQuery::size() to determine the size of a result set of the underlying database supports that feature; otherwise, we navigate to the last record and use the query's position to tell us how many records there are.

```
QSqlQuery query;
int numRows;
query.exec("SELECT name, salary FROM employee WHERE salary > 50000");

QSqlDatabase defaultDB = QSqlDatabase::database();
if (defaultDB.driver()->hasFeature(QSqlDriver::QuerySize)) {
    numRows = query.size();
} else {
    // this can be very slow
    query.last();
    numRows = query.at() + 1;
}
```

If you navigate within a result set, and use next() and seek() only for browsing forward, you can call QSqlQuery::setForwardOnly(true) before calling exec(). This is an easy optimization that will speed up the query significantly when operating on large result sets.

Inserting, Updating, and Deleting Records

QSqlQuery can execute arbitrary SQL statements, not just SELECTs. The following example inserts a record into a table using INSERT:

```
QSqlQuery query;
query.exec("INSERT INTO employee (id, name, salary) "
           "VALUES (1001, 'Thad Beaumont', 65000)");
```

If you want to insert many records at the same time, it is often more efficient to separate the query from the actual values being inserted. This can be done using

placeholders. Qt supports two placeholder syntaxes: named binding and positional binding. Here's an example of named binding:

```
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary) "
              "VALUES (:id, :name, :salary)");
query.bindValue(":id", 1001);
query.bindValue(":name", "Thad Beaumont");
query.bindValue(":salary", 65000);
query.exec();
```

Here's an example of positional binding:

```
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary) "
              "VALUES (?, ?, ?)");
query.addBindValue(1001);
query.addBindValue("Thad Beaumont");
query.addBindValue(65000);
query.exec();
```

Both syntaxes work with all database drivers provided by Qt. If the database supports the syntax natively, Qt simply forwards the query to the DBMS; otherwise, Qt simulates the placeholder syntax by preprocessing the query. The actual query that ends up being executed by the DBMS is available as QSqlQuery::executedQuery().

When inserting multiple records, you only need to call QSqlQuery::prepare() once. Then you call bindValue() or addBindValue() followed by exec() as many times as necessary.

Besides performance, one advantage of placeholders is that you can easily specify arbitrary values without having to worry about escaping special characters.

Updating a record is similar to inserting it into a table:

```
QSqlQuery query;
query.exec("UPDATE employee SET salary = 70000 WHERE id = 1003");
```

You can also use named or positional binding to associate parameters to actual values.

Finally, here's an example of a DELETE statement:

```
QSqlQuery query;
query.exec("DELETE FROM employee WHERE id = 1007");
```

Transactions

If the underlying database engine supports transactions, QSqlDriver::hasFeature(QSqlDriver::Transactions) will return true. You can use QSqlDatabase::transaction() to initiate a transaction, followed by the SQL commands you want to execute within the context of the transaction, and then either QSqlDatabase::commit() or QSqlDatabase::rollback(). When using transactions you must start the transaction before you create your query.

Example:

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec("SELECT id FROM employee WHERE name = 'Torild Halvorsen'");
if (query.next()) {
    int employeeId = query.value(0).toInt();
    query.exec("INSERT INTO project (id, name, ownerid) "
               "VALUES (201, 'Manhattan Project', "
               + QString::number(employeeId) + ')');
}
QSqlDatabase::database().commit();
```

Transactions can be used to ensure that a complex operation is atomic (for example, looking up a foreign key and creating a record), or to provide a means of canceling a complex change in the middle.