3 Kapitel

Einbindung von SQL

Einbindung von SQL

In den beiden ersten Kapiteln haben wir den Einsatz von SQL mittels interaktiver Tools besprochen. Die besprochenen Methoden bilden eine solide Grundlage für die Einbindung von SQL in Ihr Anwendungsdesign. Es macht keinen Sinn, die Feinheiten der Kombinierung von SQL und hochentwickelten Sprachen wie RPG IV zu erörtern, wenn nicht die Grundlagen von SELECT, INSERT, UPDATE und DELETE beherrscht werden. Versierte iSeries-Programmierer kennen sich für gewöhnlich mit Sprachen, wie z. B. RPG IV, COBOL und CL. Bei diesen Sprachen werden für den Datenbank-Zugriff eigene, spezifische Methoden verwendet. In diesem Kapitel wird besprochen, wie man die gängigen Datenbank-Zugriffsmethoden von RPG IV durch SQL-basierte Maßnahmen ersetzt. COBOL und andere Sprachen stützen zudem Embedded SQL, außerdem ähneln sich die Codes sehr. In diesem Buch verwenden wir jedoch Beispiele, die in RPG IV geschrieben sind.

Warum SQL in RPG IV einbetten?

Wie Sie vielleicht bereits wissen, greift RPG IV üblicherweise über „F"-Spezifikationen auf die Datenbank zu (vgl. Abbildung 3.1).

```
FCUSTMAST  IP   E             DISK
FITEMMAST  IF   E           K DISK
```

Abbildung 3.1: Beispiel für F-Spezifikationen

Diese Spezifikationen werden zur Kompilierzeit aufgelöst und steuern, welche Datentabellen gelesen werden und in welcher Reihenfolge sie ablaufen. Warum sollte man diese Spezifikationen durch SQL-basierte Instruktionen ersetzen? Haben sie uns in der Vergangenheit nicht gut gedient? Sind sie nicht die Grundlage für unglaublich viele Anwendungen? Es stimmt

zwar, dass die gängigen RPG IV-Dateizugriffsmethoden bisher einen guten Dienst geleistet haben, doch der Wettbewerb zwingt uns alle dazu, einerseits Kosten zu reduzieren und andererseits die Produktivität zu verbessern. Der Einsatz von flexibleren Programmen ermöglicht eine bessere Wiederverwendbarkeit der Objekte, eine schnellere Anwendungsentwicklung und senkt die Unterhaltskosten.

Die nächste und dabei offensichtlichste Frage ist, wie uns SQL bei der Entwicklung von flexibleren Anwendungen unterstützt. Die Antwort: durch Timing. Wie bei guten Komödien liegt das Geheimnis im Timing. Bei gängigen RPG- und COBOL-Programmen müssen Datenbankzugriffsentscheidungen bei der Entwicklung des Anwendungsdesigns getroffen werden. Beispielsweise ist festzulegen, welche Dateien verwendet und wie diese sortiert werden sollen. Bei SQL ist es möglich, solche Entscheidungen erst zu fällen, wenn die Datenbankabfrage gemacht wird. Das Potential, solche Entscheidungen verschieben zu können, ermöglicht die Erzeugung viel flexiblerer, modularisierter Anwendungen, die für eine Steigerung der Produktivität ausschlaggebend sind.

Denken Sie beispielsweise nur an die Schwierigkeiten, die man bekommt, wenn man ein einfaches Absatzdaten-Programm generieren möchte, bei dem die User bestimmen sollen, welche Datensätze und Sortierfelder verwendet werden. Gängige RPG IV- und COBOL-Programme benötigen für die Sortierung der Daten eine große Anzahl von Dateispezifikationen. Nachdem jeder Satz gelesen wurde, muss zudem die gesamte Datenselektion abgewickelt werden. Ersetzt man den gängigen Dateizugriff durch SQL, kann das SELECT-Statement erst zur Laufzeit generiert werden (Abbildung 3.2). In diesem Fall führt die SQL-Engine die gesamte Auswahl und Sortierung durch, so dass der Datenbankzugriff bei Anwendungsänderungen viel einfacher zu codieren und zu handhaben ist.

```
D MySQL       S            512
C               EVAL     MySQL = 'SELECT * FROM SALESHIST'
```

Abbildung 3.2: Erzeugung eines einfachen SQL-Statements

Der Rest dieses Kapitels beschäftigt sich mit der Untersuchung der Methoden, die zum Einsatz kommen, wenn der gängige Dateizugriff von RPG IV durch SQL ersetzt wird.

Wie man SQL in RPG IV einbettet

RPG IV-Programme, in die SQL eingebettet ist, sind im Wesentlichen noch RPG IV-Programme. Diese Programme werden gewöhnlich in die Quelldatei QRPGLESRC gespeichert, obwohl jede Quelldatei mit einer Satzgröße von 112 Bytes geeignet wäre. Der Source-Typ wird statt auf RPGLE, auf SQLRPGLE eingestellt. Dies signalisiert den Tools, wie z. B. PDM, dass diese Programme unterschiedlich kompiliert werden sollten. Programme mit Embedded SQL laufen durch einen Precompiler, bevor der abschließende Kompilierprozess durchgeführt wird. Der Precompiler sucht nach eingebetteten SQL-Statements, die mittels Kompilierer-Direktiven identifiziert werden. Jedes eingebettete SQL-Statement beginnt mit /EXEC SQL, fängt in Spalte 7 an, und endet mit einem anderen Statement, das in seiner siebten Spalte mit /END-EXEC anfängt. Jede Code-Zeile zwischen ihnen muss ein „+"-Zeichen als Fortsetzungsbuchstaben in Spalte 7 haben. Codieren Sie innerhalb eines execute/end-execute-Paars nur einen Code, und beginnen Sie das Statement nie auf der selben Zeile mit /EXEC SQL. Abbildung 3.3 illustriert die Verwendung eines eingebetteten SQL-Statements in einem RPG IV-Programm.

```
D MySQL           S            512
D Cust            S              5  0
D Name            S             30
C/EXEC SQL
C+ SELECT CUSTNAM INTO :Name FROM KPFSQL/CUSTMAST
C+ WHERE Custno = :Cust
C/END-EXEC
```

Abbildung 3.3: Eingebettetes SQL-Statement in RPG IV-Programm

Das Source Entry Utility (SEU) unterstützt die „F4"-Bedienerführung auf eingebetteten SQL-Statements, solange diese nicht auf derselben Linie wie das /EXEC SQL beginnen. Ein Beispiel für ein SQL-Statement, zu dem man aufgefordert wird, ist in Abbildung 3.4 zu sehen.

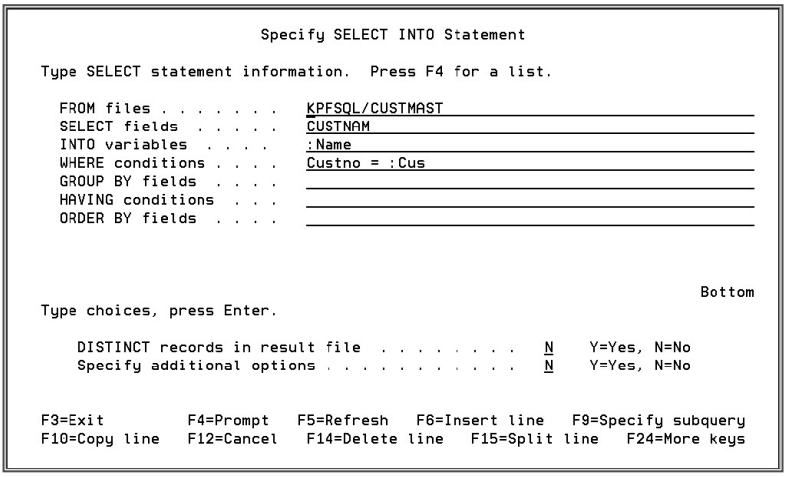

Abbildung 3.4: Beispiel einer Eingabeaufforderung

Statisches SQL

Die einfachste Form von Embedded SQL ist das statische Statement, bei dem es sich im Wesentlichen um ein hard-kodiertes SQL-Statement handelt. Dies kann wie ein Widerspruch zu unserer vorhergehenden Diskussion über flexible Programme wirken, aber dieses einfache Statement bildet einen guten Ausgangspunkt. Vor vielen Jahren habe ich mich das erste Mal mit SQL im Rahmen von Visual Basic beschäftigt. Zuerst hatte ich Schwierigkeiten, weil ich die Dateizugriffe wollte, die ich aus RPG kannte. Wo war die CHAIN-Funktion? Nachdem ich mich nun mit SQL besser auskenne, weiß ich, dass es ein Leichtes ist, CHAIN durch ein SELECT-Statement zu ersetzen. Der Zugriff auf eine Kundenstammdatei zur Abfrage eines Kun-

dennamens kann - anstatt mit „CHAIN" - mit der Eingabe eines einfachen SQL-Statements erfolgen (vgl. Abbildung 3.3).

Wie in jedem Programm, sind die Variablen „Cust" und „Name" auch im RPG IV-Programm definiert. Nimmt man in einem SQL-Statement auf diese Variablen Bezug, wird ihnen ein Doppelpunkt vorangestellt. Die SQL-Bezeichnung für diese beiden Variablen ist eine Host Variable. Jedes Mal, wenn sich das SQL-Statement auf eine Variable innerhalb des Host-Programms bezieht, wird sie als Host Variable eingestuft und ihr muss ein Doppelpunkt vorangestellt werden.

```
D MySQL           S            512
D Cust            S              5  0
D Name            S             30
D City            S             30
D State           S              2
C/EXEC SQL
C+ SELECT CUSTNAM, CUSTCTY, CUSTST INTO :Name, :City, :State
C+ FROM KPFSQL/CUSTMAST WHERE Custno = :Cust
C/END-EXEC
```

Abbildung 3.5: SELECT INTO mehrere Host Variablen

In Abbildung 3.5 ist eine realistischere und ein wenig kompliziertere Version des Statements zu sehen, mit dem man Kundenname, Ort und Staat aus dem Satz, der zu einer bestimmten Kundennummer passt, abfragen kann. Mit der INTO-Klausel können die Host Variablen (Name, Stadt und Staat) aufgelistet werden, die durch ein SELECT-Statement geladen wurden.

Merke:

Diese Methode kann nur dann angewendet werden, wenn das SELECT-Statement einen und nur einen einzigen Satz liefert. Wenn mehr als ein Satz die Selektionskriterien aufweist, dann scheitert das Statement.

Bei der Arbeit mit konkreten Betriebsdatenbankdateien liegen oft Dateien, die mehrere hundert Felder haben, vor. Die Identifikation jedes einzelnen Felds, das erzeugt wurde, ist eine umfangreiche, langwierige und kostspielige Angelegenheit. Es ist sinnvoller, eine extern definierte Datenstruktur wie die in Abbildung 3.6 zu verwenden, anstatt jedes Feld einzeln aufzulisten.

```
D Custmast      E DS                  EXTNAME(CUSTMAST)
```

Abbildung 3.6: Extern beschriebene Datenstruktur

Eine extern definierte Datenstruktur definiert für jede Tabellenspalte automatisch ein Subfeld. Wie in Abbildung 3.7 zu sehen ist, kann das SELECT-Statement dann alle Spalten in den Satz einbeziehen und den Datenstrukturnamen in seiner INTO-Klausel verwenden.

```
D MySQL           S            512
D Custmast      E DS                  EXTNAME(CUSTMAST)
C/EXEC SQL
C+ SELECT * INTO :Custmast FROM KPFSQL/CUSTMAST
C+ WHERE Custno = :Cust
C/END-EXEC
```

Abbildung 3.7: SELECT INTO externe Datenstruktur

Prüfen, ob gescheiterte SQL-Statements vorliegen

Über einen Precompiler erhält jedes RPG IV-Programm, das eingebettete SQL-Statements enthält, zusätzliche Variablen. Die zwei wichtigsten Variablen sind SQLCOD und SQLSTT, die SQL-Code- und SQL-State-Variablen. Beide zeigen den Status des zuvor durchgeführten SQL-Statements an. SQLCOD wird häufiger verwendet als SQLSTT, obwohl der Gebrauch von SQLSTT in Zukunft möglicherweise zunehmen wird.

Nach jedem erfolgreich durchgeführten SQL-Statement wird SQLCOD auf „0" gesetzt. Wenn eine Warnung festgestellt wird, setzt man eine positive Zahl; scheitert das Statement, wird eine negative Zahl eingetragen.

Merke:

Gescheiterte SQL-Statements führen nicht zu einem Programmstop. Es ist erforderlich, dass Sie nach jedem SQL-Statement, SQLCOD oder SQLSTT testen.

Fehler bei den SQL-Instruktionen führen in der Regel nicht zu Programmausfällen. Bei den gängigen RPG-Programmen musste man sich selten mit eingebetteten Fehlern beschäftigen. Doch die Tatsache, dass SQL-Fehler nur dem Programm gemeldet werden und keine Programmstörungen erzeugen, obliegt die unangenehme Aufgabe zu prüfen, ob SQL-Fehler vorliegen, dem Programmierer.

SQLCOD-Fehlercode-Werte

Wenn spezifische Fehler festgestellt werden, wird SQLCOD in der Regel durch SQLSTT ersetzt. SQLCOD verfügt über einen einfachen Indikator, der anzeigt, ob eine Maßnahme erfolgreich durchgeführt wurde oder missglückt ist. Die besten Ergebnisse werden erzielt, wenn man einfach festlegt, ob

SQLCOD negativ ist, und dann SQLSTT für den exakten Fehler als Bezugswert festlegt.

SQLSTT-Fehlercode-Werte

SQLSTT ist in zwei Teile aufgeteilt. Die ersten beiden Stellen geben die Kategorie an, und die restlichen drei Stellen identifizieren den spezifischen Fehler. In Tabelle 3.1 sind die SQLSTT-Kategorien aufgelistet und es wird kurz beschrieben, wodurch Fehler in der jeweiligen Kategorie ausgelöst werden.

Tabelle 3.1: SQLSTT Classes

Class	Error	Class	Error
00	Successful Completion	28	Invalid Authorization Specification
01	Warning	2D	Transaction Termination Error
02	No Data	2E	Connection Name Error
07	Dynamic SQL Error	2F	SQL Function Error
08	Connection Error	34	Cursor Name Error
09	Triggered Action Error	38	External Function Error
0A	Not Supported	39	External Function Call Error
0E	Invalid Schema	3B	Savepoint Error
0F	Invalid Token	3C	Ambiguous Cursor Name
0K	Handler Not Active	42	Syntax Error or Access Rule Error
20	Case Not Found	44	WITH CHECK OPTION Error
21	Cardinality Error	46	JavaTM Errors
22	Data Error	51	Application State Error
23	Constraint Error	54	SQL or Product Limit Error
24	Cursor State Error	55	Object Not in Correct State
25	Transaction State Error	56	Miscellaneous SQL Error
26	SQL Statement ID Error	57	Resource Not Available Error
27	Triggered Data Change Error	58	System Error

Weitere Informationen zu den SQLCOD- und SQLSTT-Werten finden Sie im Internet auf der Seite

http://publib.boulder.ibm.com/iseries/v5r2/ic2924/index.htm
unter „Database" und „SQL message finder".

Programme mit eingebetteten SQL-Statements können die Prüfung von SQLCOD oder SQLSTT, nachdem jedes SQL-Statement durchgeführt wurde, enthalten. Ein Beispiel hierzu finden Sie in Abbildung 3.8.

```
D MySQL            S             512
D Custmast       E DS                    EXTNAME(CUSTMAST)
C/EXEC SQL
C+ SELECT * INTO :CUSTMAST FROM CUSTMAST
C+ WHERE Custnbr = :Custnbr
C/END-EXEC
C                    IF        SQLCOD = 100
C                    EXSR      NOTFOUND
C                    ELSE
C                    IF        SQLCOD < *ZERO
C                    EXSR      SQLERROR
C                    ENDIF
C                    ENDIF
C     SQLERROR       BEGSR
C                    DUMP(A)
C                    MOVE      *ON            *INLR
C                    RETURN
C                    ENDSR
```

Abbildung 3.8: Prüfung von SQLCOD

In diesem Beispiel sieht man, wie SQL- und RPG IV-Codes vermischt werden, und wie sie sich ergänzen. Das Programm selektiert (SELECT) die Kundenadresse eines bestimmten Kunden aus der Kundenstammdatei. Wird der Satz nicht gefunden, dann wird SQLCOD auf 100 und SQLSTT auf 02000 gesetzt, und das Programm läuft weiter.

Sofort nachdem das SQL-Statement durchgeführt wurde, testet das Programm SQLCOD, um festzustellen, ob SQLCOD gleich 100 ist, und ruft dann die NOTFOUND-Subroutine ab. Diese Subroutine läuft nicht sichtbar ab, aber es führt jede Standardfehler-Bearbeitung aus, die Sie wählen. Falls SQLCOD einen negativen Wert enthält, wird eine generische Error-Handling-Subroutine abgerufen. In diesem Fall führt die SQLERROR-Subroutine einen DUMP-Befehl aus, um einen Programm-Speicherabzug auszudrucken. Der Programm-Speicherabzug enthält sowohl Fehlerinformationen, als auch eine Liste aller Variablen des Programms und deren aktuelle Inhalte. Der (A) Operation Extender auf dem DUMP erlaubt, dass der Speicherabzug sogar dann funktioniert, wenn das DEBUG-Schüsselwort in der H-Zeile nicht mit aufgenommen wurde.

Nachdem DUMP durchgeführt wurde, setzt das Programm LR auf ON und löst RETURN aus, um die Kontrolle wieder dem Anrufer zu übergeben. Möglicherweise möchten Sie diese Situation nicht immer so handhaben, aber wenn Sie wollen, dass das Programm bei ernsten SQL-Fehlern beendet wird, ist dies eine Möglichkeit dies zu erreichen.

Es gibt einige Beispiele, bei denen Fehler mit dem SQL WHENEVER-Statement behoben werden. Ich empfehle Ihnen, dieses Statement zu vermeiden. Beim Einsatz dieses Statements durch den Precompiler sind die Folgen weder klar, noch vorhersehbar. Anstatt widersprüchliches Programmverhalten zu riskieren, schlage ich vor, Sie vermeiden den Einsatz von

WHENEVER und prüfen nach jedem Statement einfach SQLCOD, wie im obigen Beispiel veranschaulicht wurde.

Andere SQL-Statements durchführen

Die vorangegangenen Beispiele arbeiten alle mit SELECT-Statements. Abgesehen von SELECT, was ist mit den vielen anderen Optionen? Wie implementiert man UPDATE, INSERT, DELETE und all die anderen SQL-Befehle, die keinen Ergebnissatz erzeugen? Diese Statements können einfach direkt in das Programm kodiert werden.

Was passiert beispielsweise, wenn ein Programm die Adresse eines Kunden aktualisieren soll? Mit dem einfachen SQL-Statement aus Abbildung 3.9 kann dies bewerkstelligt werden.

```
D MySQL           S            512
D Custmast      E DS                  EXTNAME(CUSTMAST)
C/EXEC SQL
C+ UPDATE CUSTMAST Set Custad1 = :Custad1,
C+    Custad2 = :Custad2,
C+    CustCty = :CustCty, CustST = :CustST,
C+    CustZip = :CustZip
C+    WHERE Custnbr = :Custnbr
C/END-EXEC
C                   IF        SQLCOD < *ZERO
C                   EXSR      SQLERROR
C                   ENDIF
C     SQLERROR      BEGSR
C                   DUMP(A)
C                   MOVE      *ON           *INLR
C                   RETURN
C                   ENDSR
```

Abbildung 3.9: Aktualisierung einer Datei

Da wir sowohl die Datei, die wir aktualisieren, als auch die Spalten, die aktualisiert werden, kennen, können wir die betreffenden Werte einfach hart kodiert in das EXEC SQL-Statement einfügen. Die neuen Werte für die Kundenadresse stammen aus Host Variablen.

Das Hinzufügen eines Satzes kann etwas komplizierter sein, weil man entweder angeben muss, welche Spalten man lädt, oder man muss für jede Spalte einen Wert zur Verfügung stellen. Die Anwendung der ersten Methode ist bei kleinen Dateien relativ einfach (Abbildung 3.10).

```
D Custmast      E DS                  EXTNAME(CUSTMAST)
C/EXEC SQL
C+ INSERT INTO CUSTMAST (Caddr1, Caddr2,CCity,CState,Czip)
C+    VALUES(:Addr1, :Addr2, :City, :State, :Zip)
C/END-EXEC
C                   IF        SQLCOD < *ZERO
C                   EXSR      SQLERROR
C                   ENDIF
C     SQLERROR      BEGSR
C                   DUMP(A)
C                   MOVE      *ON           *INLR
C                   RETURN
C                   ENDSR
```

Abbildung 3.10: Daten in eine Datei einfügen und eine Spaltenliste spezifizieren

Eine Liste der zur Verfügung stehenden Felder wird direkt nach dem Dateinamen hinzugefügt. Die VALUES-Klausel stellt dann die Werte für alle entsprechenden Variablen zur Verfügung. Bei größeren Dateien mit Hunderten Feldern ist das sehr unpraktisch. Um das Laden der Werte in den Satz zu vereinfachen, kann man dieselbe Methode der extern definierten Datenstruktur verwenden, die bereits besprochen wurde. Dieses Mal verschieben wir jedoch die Daten aus der Struktur in den Satz, statt andersherum. In Abbildung 3.11 werden die Unterfelder der extern definierten Datenstruktur individuell im RPG-Code geladen. Dann wird die gesamte Datenstruktur als Referenz-Host Variable des SQL-Statements definiert.

```
DCMRec          E DS                  EXTNAME(CUSTMAST)
C                   EVAL      Caddr1 = Addr1
C                   EVAL      Caddr2 = Addr2
C                   EVAL      CCity = City
C                   EVAL      CState = State
C                   EVAL      CZip = Zip
C/EXEC SQL
C+ INSERT INTO CUSTMAST VALUES(:CMRec)
C/END-EXEC
C                   IF        SQLCOD < *ZERO
C                   EXSR      SQLERROR
C                   ENDIF
C     SQLERROR      BEGSR
C                   DUMP(A)
C                   MOVE      *ON           *INLR
C                   RETURN
C                   ENDSR
```

Abbildung 3.11: Daten über eine externe Datenstruktur in eine Datei einfügen

Dynamisches SQL

Alle bisherigen Beispiele haben sich mit statischen SQL-Statements befasst, in denen alle „Variablen" zur Kompilierzeit bekannt sind. Da bekannt war, mit welchen Dateien und Spalten gearbeitet werden sollte, konnten diese „statischen" SQL-Statements einfach hart-kodiert definiert werden. Bei dem Versuch, kompliziertere SQL-Anwendungen zu generieren, entstehen Situationen, in denen die ORDER BY-Klausel oder die WHERE-Klausel erst bei Laufzeit festgelegt werden können.

Es ist möglich eine Anwendung zu erzeugen, die dem User erlaubt, alle Datensätze aus der Umsatzdatei zu löschen, die bestimmte Kriterien aufweisen, wie z. B. einen bestimmten Verkäufer, eine bestimmte Produktlinie, oder das Datum fällt in einen bestimmten Bereich. Abbildung 3.12 zeigt, wie ein SQL-Statement in einer Zeichenfolge zusammengesetzt wird.

```
D MySQL           S            512
D CNum            S              5  0
DCMRec          E DS                    EXTNAME(CUSTMAST)
C                   EVAL      MySQL = 'DELETE FROM ' +
C                             'CUSTMAST WHERE CustNo = ' +
C                             %EDITC(CNum:'X')
C/EXEC SQL
C+ EXECUTE IMMEDIATE :MySQL
C/END-EXEC
C/EXEC SQL
C+ EXECUTE IMMEDIATE :MySQL
C/END-EXEC
C                   IF        SQLCOD < *ZERO
C                   EXSR      SQLERROR
C                   ENDIF
C     SQLERROR      BEGSR
C                   DUMP(A)
C                   MOVE      *ON           *INLR
C                   RETURN
C                   ENDSR
```

Abbildung 3.12: Zusammensetzen eines SQ-Statements

Das SQL-Statement wird dann durch den EXECUTE IMMEDIATE-Befehl durchgeführt. Die Fähigkeit, SQL-Instruktionen schnell bei Laufzeit zusammenzusetzen, und diese Instruktionen dann sofort auszuführen, ist eines der hervorstechendsten Merkmale, wodurch sich SQL von den gängigen RPG IV-Dateizugriffsmethoden abhebt. Durch den Einsatz von eingebetteten SQL-Funktionen, können Entschei-

dungen darüber, was das SQL-Statement genau tun muss, solange aufgeschoben werden, bis Kontakt zum User und vielleicht zu anderen Anwendungsprogrammen aufgenommen wurde.

Jedes Mal, wenn das Statement durchgeführt wird (EXECUTED), analysiert die SQL-Engine das Statement und arbeitet einen Plan für die Statement-Implementierung aus. Dadurch wird eine gewisse Menge Overhead produziert, die in der Vergangenheit während der Kompilierzeit verarbeitet wurde. Wendet man dynamisches SQL an, können diese Kosten vermieden werden. Immer dann, wenn das Programm abläuft, muss der SQL-„Plan" für die Implementierung des Statements mindestens ein Mal erzeugt werden. Was, wenn das Statement mehrmals durchgeführt wird? Der „Plan" wird jedes Mal wieder erzeugt, wodurch noch mehr Overhead produziert wird. Wir können dieses Overhead zwar nicht eliminieren, aber wir können es reduzieren. Wird ein SQL-Statement pro Job öfters als ein Mal ausgeführt, kann das Statement mittels PREPARE-Statement (Abbildung 3.13) zur SQL-Engine geschickt werden, wo sie dann analysiert wird.

```
D MySQL           S            512
D CNum            S              5  0
DCMRec          E DS                    EXTNAME(CUSTMAST)
C                   EVAL      MySQL = 'DELETE FROM ' +
C                             'CUSTMAST WHERE ' +
C                             'CustNo =' + %EDITC(CNum:'X')
C/EXEC SQL
C+ PREPARE Stmt1 FROM :MySQL
C/END-EXEC
C/EXEC SQL
C+ EXECUTE Stmt1
C/END-EXEC
C                   IF        SQLCOD < *ZERO
C                   EXSR      SQLERR
C                   ENDIF
C     SQLERR        BEGSR
C                   DUMP(A)
C                   MOVE      *ON           *INLR
C                   RETURN
C                   ENDSR
```

Abbildung 3.13: Verwendung des SQL PREPARE-Statements

Dieser Code erzeugt einen „Plan". Später kann mit diesem Plan das gleiche Statement nochmals abgerufen werden. Immer wenn das gleiche Statement mehrmals durchgeführt wird und das Statement dabei als PREPARE-Statement erzeugt wird, nutzt dies der Performance. In diesem Beispiel wird das SQL-Statement in eine Variable geladen (wie in Abbildung 3.12). Dieses Mal wird das Statement jedoch nicht sofort durchgeführt. Stattdessen wird sie vorbereitet (PREPARE).

Das Label Stmt1 identifiziert das vorbereitete Statement. Statt das Statement in der MySQL-Variablen durchzuführen, wird der mit Stmt1 verbundene Plan verwendet. Obwohl dies schwieriger zu kodieren ist, erweist es sich in vielen Fällen als effizienter.

Der Umgang mit Anführungszeichen bei Zeichenwerten

Im vorigen Beispiel wurde getestet, ob die Firmenzahl gleich einer spezifischen numerischen Zahl war. Dies war relativ einfach. Doch wenn es sich bei dem Feld um ein alphanumerisches Feld handelt, würde diese Methode nicht funktionieren, weil keine Host Variablen im Statement verwendet werden. Das gesamte Statement wird in einer großen Kette zusammengesetzt. Vor der Durchführung sieht das Statement aus Abbildung 3.13 wie folgt aus:

```
'DELETE FROM CUSTMAST WHERE CustNo =  12345'
```

Wenn der Test statt für eine Zahl, für einen Firmennamen durchgeführt worden wäre, hätte das Statement so ausgesehen:

```
'DELETE FROM CUSTMAST WHERE CNname =  WHOEVER'
```

Dieses Statement wird scheitern, weil Buchstaben mit Anführungszeichen umschlossen werden müssen. Eine einfache Lösung hierfür ist, eine benannte Konstante zu definieren, welche Quote genannt wird, die den Wert von zwei Anführungszeichen hat. Diese Konstante wird an jedes Ende der Buchstabenkette angefügt.

```
D MySQL           S            512
D Name            S             30
D Quote           C                    CONST(''''')
DCMRec          E DS                   EXTNAME(CUSTMAST)
C                   EVAL      MySQL = 'DELETE FROM ' +
C                             'CUSTMAST WHERE ' +
C                             'CName = ' +
C                             Quote + Name + Quote
C/EXEC SQL
C+ PREPARE Stmt1 FROM :MySQL
C/END-EXEC
C/EXEC SQL
C+ EXECUTE Stmt1
C/END-EXEC
C                   IF        SQLCOD < *ZERO
C                   EXSR      SQLERROR
C                   ENDIF
C     SQLERROR      BEGSR
C                   DUMP(A)
C                   MOVE      *ON           *INLR
C                   RETURN
C                   ENDSR
```

Abbildung 3.14: Definieren einer benannten Konstante QUOTE

Immer wenn ein alphanumerisches Feld mit einem Wort-Wert verglichen wird, muss das Wort von Anführungszeichen umschlossen sein. Die resultierende Buchstabenkette, die das SQL-Statement enthält, würde wie folgt aussehen:

```
'DELETE FROM CUSTMAST WHERE CNname =  'WHOEVER''
```

Auswahl (SELECT) von Mehrfachzeilen

In den vorherigen SELECT-Beispielen wurde nur auf einen Satz zurückgegriffen. Was, wenn mehr Datensätze benötigt werden? Zum Beispiel, kann man ein Abfrageprogramm für die Kundenstammdatei erstellen, das Usern erlaubt, Datensätze aus der Kundenstammdatei anzuzeigen oder auszudrucken. (Vgl. Abbildung 3.14.) Mit dem SELECT-Statement kann man Mehrfachzeilen aus der Datenbank holen. Wie bereits oben erwähnt wurde, kann deshalb beim Abrufen dieser Daten auf den Einsatz der INTO-Klausel verzichtet werden.

SELECT-Statements erzeugen Ergebnis-Sets, die manchmal Ergebnistabellen genannt werden. Diese Ergebnistabellen können als temporäre Dateien angesehen werden, die für Ihren Job im Speicher existieren. Da diese Ergebnis-Sets aus Hunderten, Tausenden oder sogar Millionen von Datensätzen bestehen können, brauchen wir einen Mechanismus, mit dem wir uns in diesen Ergebnis-Sets bewegen und die einzelnen Datensätze verarbeiten können. Diesen Mechanismus bietet der SQL CURSOR. Man kann sich den Cursor als Speicherpuffer vorstellen, der die Ergebnis-Sets enthält. Er hat einen Zeiger bzw. „Cursor", der verfolgt, welche Zeile gerade bearbeitet wird. Die Arbeit mit einem Cursor setzt eine Anzahl verschiedener SQL-Statements voraus, die miteinander arbeiten. Jeder der folgenden Schritte muss in folgender Reihenfolge durchgeführt werden:

1. DECLARE: einen Cursor deklarieren
2. OPEN: den Cursor öffnen
3. FETCH: einen Satz holen (kann wiederholt werden)
4. CLOSE: einen Cursor schließen, wenn Sie fertig sind

Abbildung 3.15 zeigt, wie diese Statements miteinander arbeiten, um die ausgewählten Datensätze aus der Absatzdaten-Datei zu lesen.

```
D MySQL           S            512
D Cust            S              5  0
DCMRec          E DS                    EXTNAME(CUSTMAST)

C/EXEC SQL
C+ Declare @C1 CURSOR for SELECT * FROM CUSTMAST
C/END-EXEC
C/EXEC SQL
C+ OPEN @C1
C/END-EXEC
C/EXEC SQL
C+ FETCH @C1 INTO :CMRec
C/END-EXEC
C                   DOW       SQLCOD = *Zero
C                   EXSR      LDSUBF
C/EXEC SQL
C+ FETCH @C1 INTO :CMRec
C/END-EXEC
C                   IF        SQLCOD < *ZERO
C                   EXSR      SQLERROR
C                   ENDIF
C                   ENDDO
C/EXEC SQL
C+ CLOSE @C1
C/END-EXEC
C                   IF        SQLCOD < *ZERO
C                   EXSR      SQLERROR
C                   ENDIF
C     SQLERROR      BEGSR
C                   DUMP(A)
C                   MOVE      *ON           *INLR
C                   RETURN
C
```

Abbildung 3.15: Verarbeitung von Ergebnis-Sets

Um die Funktionsweise dieses Prozesses zu begreifen, ist es wichtig zu verstehen, welche Rolle jedes dieser SQL-Statements spielt.

Das DECLARE-Statement (Abbildung 3.16) analysiert das mit ihm verbundene SQL-Statement, bestimmt, dass Spalten zurückgeschickt werden, und definiert einen Satz-Puffer, der die Daten enthält, die bereitgestellt werden, wenn das Statement durchgeführt wird.

```
C/EXEC SQL
C+ Declare @C1 CURSOR for SELECT * FROM CUSTMAST
C/END-EXEC
```

Abbildung 3.16: DECLARE-Statement

Willkürlich haben wir den Cursor @C1 genannt. (In diesem Kapitel beginnen alle Cursor-Namen mit @, damit deutlich wird, dass es sich hierbei nicht um einen Datei- bzw. Feldnamen handelt, sondern um ein temporäres SQL-Objekt.) Das DECLARE-Statement hat viele Parameter. Einige davon werden später behandelt. Momentan beschäftigen wir uns aber noch mit leichteren Dingen.

In Abbildung 3.17 ist das OPEN-Statement zu sehen. Es identifiziert, welcher SQL-Cursor zu öffnen ist.

```
C/EXEC SQL
C+ OPEN @C1
C/END-EXEC
```

Abbildung 3.17: OPEN-Statement

Das SELECT-Statement wird ausgeführt, wenn der Cursor geöffnet wird. Alle Datenzeilen und -spalten die das SELECT-Statement zurückschickt, werden in den CURSOR geladen.

Vor dem ersten Satz wird der Satz- oder Zeilen-Cursor an den Dateianfang gesetzt.

FETCH (Abbildung 3.18) liest den nächsten Satz vom spezifizierten Cursor ab.

```
C/EXEC SQL
C+ FETCH @C1 INTO :CMRec
C/END-EXEC
```

Abbildung 3.18: FETCH-Statement

Die INTO-Klausel identifiziert in welche Host Variablen die ausgewählten Datenspalten geladen werden. Wie bereits erwähnt, vereinfacht eine extern definierte Datenstruktur die Spezifizierung der Host Variablen für FETCH, da dadurch jede einzelne Tabellenspalte automatisch bearbeitet wird. Selbstverständlich funktioniert das nur, wenn Sie jede Tabellenspalte selektieren.

Wenn mehrere Datensätze im Ergebnis-Set existieren, wird in der Regel ein Loop um FETCH codiert, so dass wiederholt durchgeführt wird, bis das Dateiende erreicht ist. (Das Dateiende wird im Folgenden noch besprochen.) Nachdem jeder Satz gelesen ist, wird ein zusätzlicher Code geschrieben, um die Daten aus der Fetch-Zeile zu verwenden.

Abbildung 3.19 zeigt das CLOSE-Statement, das den Cursor schließt. Hierdurch wird das temporäre Cursor-Objekt effektiv gelöscht und verhindert, dass weitere Daten daraus abgelesen werden.

```
C/EXEC SQL
C+ CLOSE @C1
C/END-EXEC
```

Abbildung 3.19: CLOSE-Statement

Testen von Dateiende und anderen Bedingungen

In dem Beispiel aus Abbildung 3.15 wird die Verknüpfung eines RPG IV-Codes mit einem SQL-Statement gezeigt. Ein gängiger DOW-Befehl kontrolliert den Ablauf des Programms, während alle Datensätze des Ergebnis-Sets durchlaufen werden. Er überwacht die SQLCOD-Variable, bis ein Wert erzeugt wird, der nicht gleich Null ist. Ein Wert von 100 signalisiert, dass das Dateiende erreicht wurde, jedoch könnten trotzdem Fehler auftreten.

Nach jedem SQL-Statement, wird SQLCOD getestet. Wenn es negative Werte aufweist, sendet das Programm eine Fehlermeldung und schaltet ab. Bei positiven Werten (ausgenommen beim Dateiende) wird ein Warnhinweis zurückgeschickt. Falls der Wert gleich Null ist, erkennt SQLCOD keine Fehler und das Programm wird fortgesetzt.

Merke: Dateiende sollte sofort nach dem Versuch, Datensätze zu holen (FETCH), getestet werden. Ein adäquates Timing bei diesem Test senkt das Risiko, dass in Ihrem Code Ungereimtheiten und Fehler auftauchen.

Vorbereitete SELECT-Statements

Wenn das SELECT-Statement in jedem Aufruf Ihres Programm mehrere Male ausgeführt werden soll, ist es effektiver, ein Statement vorzubereiten (PREPARE) (Abbildung 3.20).

```
C                   EVAL      MySQL = 'SELECT * FROM ' +
C                             'CUSTMAST '
C/EXEC SQL
C+ Prepare Stmt1 from :MySQL
C/END-EXEC
```

Abbildung 3.20: Prepare-Statement

Wurde das Statement vorbereitet, modifizieren Sie das DECLARE-Statement (vgl. Abbildung 3.21), um, anstatt auf das hart-kodierte SELECT-Statement (siehe Abbildung 3.16), auf das vorbereitete Statement zurückzugreifen.

```
C/EXEC SQL
C+ Declare @C1 CURSOR for Stmt1
C/END-EXEC
```

Abbildung 3.21: Ein vorbereitetes Statement deklarieren (DECLARE)

Wie bereits erwähnt wurde, untersucht das PREPARE-Statement das entsprechende SELECT-Statement, vergleicht es mit der Datenbank, und entwickelt einen Plan, um das SELECT-Statement durchzuführen. Der Name des Statements wird als „Zugriff" verwendet, mit dem man Bezug auf diesen spezifischen Plan nimmt.

PREPARE kann fast als Mini-Kompilierung verstanden werden. Das PREPARE-Statement analysiert den Quellcode des

betreffenden Statements und erzeugt aus diesem Code eine ausführbare SQL-Instruktion. Dies geschieht entweder während der Kompilierzeit oder der Laufzeit. Der ausführbare Code ähnelt dem eines ausführbaren High-Level-Sprachprogramms, jedoch kann er nur innerhalb SQL verwendet werden. Wenn alle Informationen, die für das Statement benötigt werden, zur Kompilierzeit nicht zur Verfügung stehen (Dateien nicht gefunden), wird das Statement bei Ablaufzeit fertig sein (PREPARED). Extra-Kodierungen sind für diesen Ablauf nicht notwendig, da der Precompiler diese Aufgabe übernimmt.

Die Daten im Ergebnis-Set aktualisieren

In diesem Kapitel wurde bereits besprochen, wie das UPDATE-Statement in unser RPG IV-Programm eingebettet wird. In vielen Fällen funktioniert dieses Statement gut, aber manchmal muss man eine große Anzahl von Datensätzen lesen und einige oder alle aktualisieren. Dieser Prozess ähnelt sehr einem gängigen RPG READ innerhalb eines Loops. Bevor ein Satz unserer Ergebnistabelle aktualisiert werden kann, muss dessen Deklarierung verändert werden. Das DYNAMIC-Schlüsselwort muss zum DECLARE-Statement hinzugefügt werden (vgl. Abbildung 3.22).

```
C/EXEC SQL
C+ Declare @C1 DYNAMIC SCROLL CURSOR for Stmt1
C/END-EXEC
```

Abbildung 3.22: DYNAMIC-Schlüsselwort in einem vorbereiteten Statement

Wenn das UPDATE-Statement die WHERE CURRENT OF-Klausel verwendet, kann es zur Aktualisierung der Datei eingesetzt werden (Abbildung 3.23).

```
C/EXEC SQL
C+ UPDATE CUSTMAST set CUSTAD1 = :CUSTAD1,
C+                     CUSTAD2 = :CUSTAD2,
C+                     CUSTCTY = :CUSTCTY,
C+                     CUSTST = :CUSTST,
C+                     CUSTZIP = :CUSTZIP
C+                     where current of @C1
C/END-EXEC
```

Abbildung 3.23: Aktualisierung eines Satzes des Ergebnis-Sets

Identifiziert man die aktuelle Zeile eines bestimmten Cursors als UPDATE-Ziel, wird die letzte Zeile, die von diesem Cursor gelesen wurde, aktualisiert. Das DYNAMIC-Schlüsselwort setzt voraus, dass auch SCROLL spezifisiert ist. Die Funktion des SCROLL-Schlüsselworts wird in diesem Kapitel später besprochen.

Wo kann man diese Art Update anwenden? In einem Programm, das selektierte Datensätze aus der Kundenstammdatei anzeigt und dem User erlaubt, die Ergebnistabellen durchzusehen, wird bei Drücken von Page Down jedes Mal ein neuer Satz abgerufen (FETCH) und auf dem Bildschirm angezeigt. Jegliche Änderungen, die durch die User vorgenommen werden, werden sofort im Satz eingetragen. Das gleiche hätte man mit einem UPDATE-Statement, das sich völlig von dem SELECT-Statement unterscheidet, erreicht (Abbildung 3.24).

```
D MySQL           S            512
D Cust            S              5  0
DCMRec          E DS                    EXTNAME(CUSTMAST)
C                   EVAL      MySQL = 'SELECT * FROM '
C                             'CUSTMAST '
C/EXEC SQL
C+ Prepare Stmt1 from :MySQL
C/END-EXEC
C/EXEC SQL
C+ Declare @C1 DYNAMIC SCROLL CURSOR for Stmt1
C/END-EXEC

C/EXEC SQL
C+ OPEN @C1
C/END-EXEC
C/EXEC SQL
C+ FETCH  NEXT  @C1 INTO :CMRec
C/END-EXEC
```

```
C/EXEC SQL
C+ UPDATE CUSTMAST set CUSTAD1 = :ScrnAD1,
C+                     CUSTAD2 = :ScrnAD2,
C+                     CUSTCTY = :ScrnCTY,
C+                     CUSTST = :ScrnST,
C+                     CUSTZIP = :ScrnZIP where
C+                     CUSTNBR = :ScrnCust
C/END-EXEC

C/EXEC SQL
C+ CLOSE @C1
C/END-EXEC
```

Abbildung 3.24:Einsatz eines separaten UPDATE-Statements

Dies funktioniert nur, wenn der gewünschte Satz eindeutig identifiziert werden kann. In manchen Datenbanktabellen stehen jedoch keine eindeutigen Kennzeichnungen zur Verfügung. Wenn History-Dateien Dutzende Datensätze mit gleichen Schlüsselfeldern enthalten, kann das UPDATE-Statement nicht feststellen, welcher der aktuelle Satz ist. Die WHERE CURRENT OF-Klausel ist in jedem Programm potentiell einsetzbar, doch sie ist bei der Arbeit mit Datensätzen, die keine eindeutigen Schlüssel enthalten, am nützlichsten.

Satzsperre

Wenn das Ergebnis-Set für ein Update geöffnet wird, beginnt die Sperrung von Datensätzen. Die Satzsperre beeinflusst möglicherweise andere Anwendungen. Um dies so weit wie möglich zu verhindern, muss verstanden werden, wie das Sperren funktioniert.

SQL beruht bei der Kontrolle von Satzsperren auf dem TRANSACTION ISOLATION LEVEL. Die fünf unterschiedlichen Isolierungslevel sind:

- No Commit
- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

„No Commit" ist die einzige Einstellung, die zugelassen wird, wenn die zu aktualisierenden Dateien nicht journalisiert sind. Weitere Informationen zu Journalisierung finden Sie auf http://publib.boulder.ibm.com/iseries/v5r2/ic2924/index.htm unter der Überschrift „Database, Manuals, and Journal Management". Wenn „No Commit" aktiv ist, werden Datensätze nie durch einen SELECT-Befehl gesperrt. Geholte Datensätze (FETCH) werden nur gesperrt, falls der Cursor für ein Update geöffnet wurde (indem man das DYNAMIC-Schlüsselwort zum DECLARE-Statement hinzufügt; siehe oben). Gesperrte Datensätze werden nur solange erhalten, bis ein Update durchgeführt wird oder das nächste FETCH durchgeführt wird. Dieses Modell funktioniert ähnlich wie die gängige RPG-Satzsperre. Erfahrene Programmierer werden wahrscheinlich feststellen, wie bequem mit diesem Modell gearbeitet werden kann.

„Read Uncommitted" setzt Journalisierung voraus. Über SELECT ausgewählte Datensätze sind nicht gesperrt. Aktualisierte Datensätze (UPDATE) sind dagegen solange gesperrt, bis ein COMMIT oder ROLLBACK durchgeführt wird. COMMIT und ROLLBACK werden später in diesem Abschnitt besprochen. Nicht-festgelegte (uncommitted) Änderungen anderer Programme stehen diesem Programm zur Verfügung (siehe Abbildung 3.25).

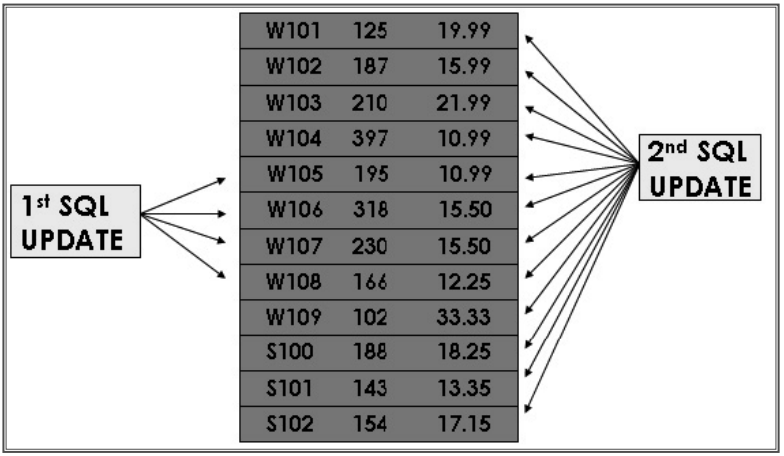

Abbildung 3.25: Transaktionsniveau von Read Uncommitted

„Read Committed" setzt Journalisierung voraus. Wie bei „Read Uncommitted" sind auch hier SELECT-Datensätze nicht gesperrt, aber der aktuelle FETCH-Satz ist gesperrt. Aktualisierte Datensätze (UPDATE) sind solange gesperrt, bis ein COMMIT oder ROLLBACK durchgeführt wird. Nicht-festgelegte (uncommitted) Änderungen aus anderen Programmen stehen diesem Cursor nicht zur Verfügung. In Abbildung 3.26 wird illustriert, dass nicht-festgelegte Änderungen, die in anderen Programmen durchgeführt werden, in diesem Programm nicht verfügbar sind.

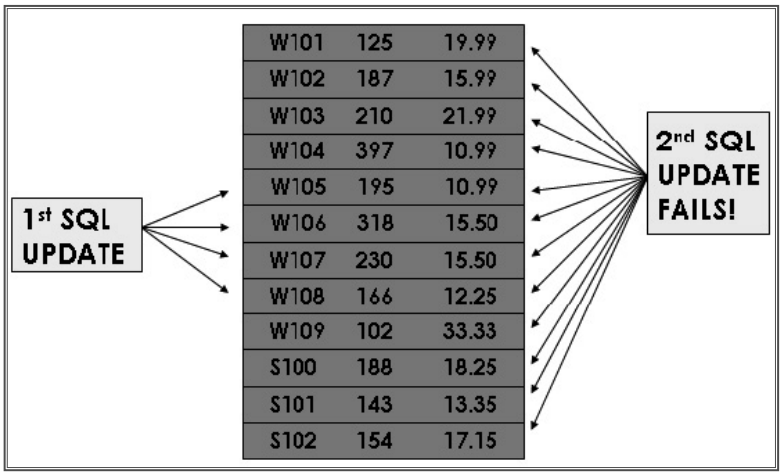

Abbildung 3.26: Transaktionsniveau von Read Committed

„Repeatable Read" setzt Journalisierung voraus. Abgesehen davon, dass bei „Repeatable Read" alle ausgewählten Datensätze und nicht nur der aktuelle FETCH-Satz gesperrt sind, handelt es sich um den gleichen Isolierungslevel wie bei „Read Commited".

„Serializable" setzt Journalisierung voraus. Wie „Repeatable Read" erfordert auch dieser Isolierungslevel eine exklusive Sperrung der betroffenen Tabellen. Damit wird verhindert, dass über andere Programme Einträge vorgenommen oder die Tabellen gelesen werden können.

„Commit" ist ein SQL-Statement, das eine Auswahl von Datenbank-Änderungen einschließt. Mit Ausnahme von „No Commit", können Datenbank-Änderungen unter einem beliebigen Isolationsniveau laufen und sind dabei solange nicht permanent, bis sie mit „Commit" festgelegt werden (Abbildung 3.27).

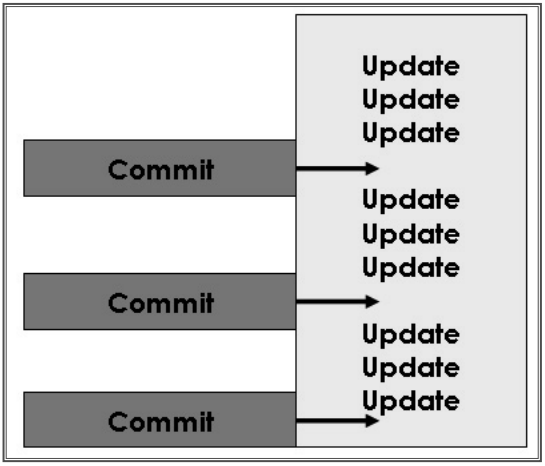

Abbildung 3.27: Verwendung des Commit-Statements

Das Commit-Statement markiert alle Datenbank-Änderungen, die nach dem letzten COMMIT oder ROLLBACK als permanente Änderungen durchgeführt wurden.

ROLLBACK ist ein SQL-Statement, das Datenbank-Änderungen löscht (ähnlich wie „undo"). Alle Änderungen, die seit dem letzten COMMIT oder ROLLBACK vorgenommen wurden, werden gelöscht. Dadurch wird die Datenbank wiederhergestellt, so als ob die Änderungen nie vorgenommen worden wären. Ist der Isolierungslevel auf „No Commit" festgelegt, kann dieses Statement nicht verwendet werden.

Weitere Informationen zu Commitment Control finden Sie in der IBM-Dokumentation auf

http://publib.boulder.ibm.com/iseries/v5r2/ic2924/index.htm
unter der Überschrift „Database, Manuals, and Commitment Control".