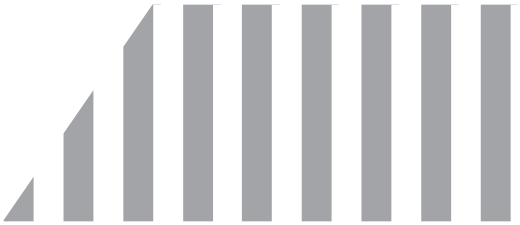
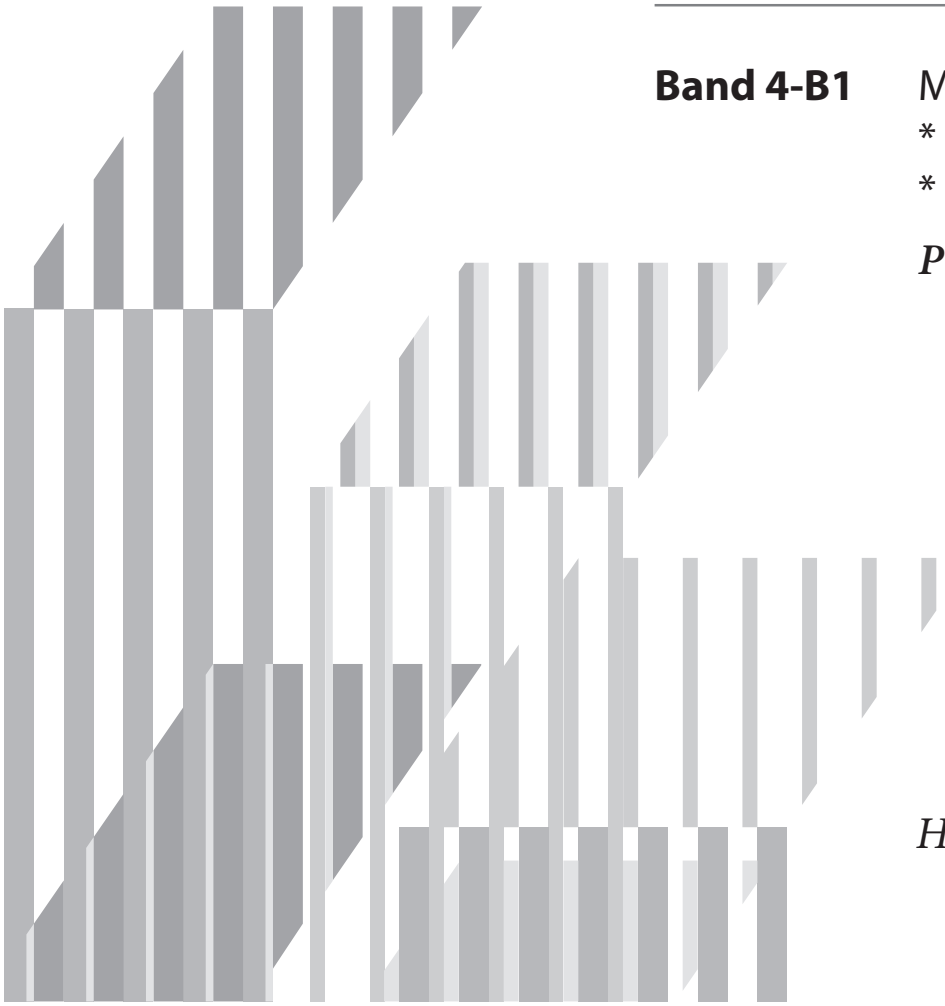
Band 4-B1



MODERNE STRUKTURIERTE PROGRAMMIERUNG

Objektorientiert und algorithmisch
ENTWERFEN | IMPLEMENTIEREN | TESTEN



Band 4-B1 Mischen
* sequentiell
* relational

Praxis

Horst van Bremen

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Gestaltung	Katharina Schmitt
Titel- und Einbandgrafik	Monika Sylvia Gomm † 1971
Englisch-Korrektorat	David Roseveare
Verlag	BoD · Books on Demand GmbH, Überseering 33, 22297 Hamburg, bod@bod.de
Druck	Libri Plureos GmbH, Friedensallee 273, 22763 Hamburg
Version / Datum	Version 1.0.2 vom 1.7.2026
Literaturverzeichnis	siehe Kapitel „13.1 Literaturverzeichnis“ auf Seite 379 f
Rechtliches	siehe Kapitel „13.2 Rechtliche Hinweise“ auf Seite 381 ff
ISBN des Ringbuchs	9783695784202

Über den Autor

Horst van Bremen
Diplom 1972 an der TU Darmstadt – Elektrotechnik und Informatik
39 Berufsjahre in der IT, davon 18 als Freiberufler

IT-Systemarchitekt
Datenbankexperte

Programmiererfahrung seit 1969
Strukturierte Programmierung nach Jackson seit 1974
Freier Autor seit 2011

Vorwort zur B1-Ausgabe

Dieser Band legt die Implementierungen für die Lösungsvarianten der Mischaufgaben aus dem Band 3-B1 vor und dokumentiert die Tests. Sequentiell konzipierte Lösungen wurden in C realisiert, relationale in Java®. Eine umfangreiche – auf MySQL® ausgerichtete und erweiterbare – Zugriffsschicht vereinfacht die Java-Implementierungen und erleichtert es, diese zu verstehen. Sie ist am Ende des Bands 4-B2 dokumentiert.

Aus der Sicht der Leserinnen und Leser handelt es sich hier um einen ziemlich hohen Berg von Code, der auf den ersten Blick sicherlich keine attraktive Lektüre darstellt. Über das Primärziel hinaus, Implementierungen für die im Band 3-B1 entworfenen Lösungen vorzustellen, gibt es jedoch einige weitere Aspekte, die es als lohnend erscheinend lassen, sich zumindest punktuell mit diesen Programmen zu befassen:

- Vor allem die relationalen Lösungen folgen einem Erkenntnisweg. Das erste Beispiel, SQL_Example0, ist noch eine ziemlich triviale, nicht restartfähige und ineffektive Implementierung. Es führt aber schon den Nutzen der Zugriffsschicht vor, weil enorm viel Code-Ballast dorthin ausgelagert werden konnte. Dadurch wird die Anwendungsprogrammierung weitaus straffer und somit übersichtlicher. Angenehm ist, dass zeilenbezogene Zähler nun nicht mehr im Programm deklariert und geführt werden müssen, sondern von der Zugriffsschicht bereitgestellt werden, wie das Laufprotokoll zeigt. SQL_Example0 und SQL_Example1 deklarieren die SQL Statements noch im Code der `main()`, während die höherentwickelten Varianten dafür die Klasse `SQL_ID` verwenden. Bereits bei SQL_Example1 tritt die Klasse `ColumnID` hinzu, die Spaltennummern und -namen enumeriert. Das macht den Code robuster und schützt vor Irrtümern.
- Schon SQL_Example1 ist restartfähig, allerdings ohne übergreifende Statistik. SQL_Example2 behebt dieses Manko, indem es eine Restart-Tabelle führt. Diese Tabelle ermöglicht es zugleich, den Restart-Vorgang je nach Abbruchort differenziert zu handhaben. Daher eignet sie sich generell als Basis für eine transparente und effiziente Restartprogrammierung. Diese muss zwar auf das jeweilige Programm zugeschnitten werden, besitzt aber auch generalisierbare Eigenschaften. SQL_Example5r demonstriert Restartfähigkeit, die sich auf Wiederaufsetz-Schlüssel in der Restart-Tabelle stützt.
- In Umgebungen mit Konkurrenzverarbeitung können während einer laufenden oder unterbrochenen Verarbeitung neue Zeilen in Schnittstellentabellen eintrudeln, die aus Konsistenzgründen nicht an Restarts teilnehmen sollen, sondern erst später verarbeitet werden dürfen. SQL_Example3 und SQL_Example4 berücksichtigen das, indem sie beim ersten Start eine Zeitgrenze namens „Buchungsschnitt“ ziehen und diese bei Restarts beibehalten. Auch dies ist eine realistische Vorgabe für Produktionsprogramme.
- Durchgehend wurde Wert darauf gelegt, eine angemessene Protokollierung der Tests zu ermöglichen. Das ist nicht selbstverständlich, denn viele Produktionsprogramme sind „unter-instrumentiert“, ermöglichen es also im Fehlerfall oft nicht, die Ursachen anhand von ein- und ausschaltbaren Testlaufausgaben zu lokalisieren. Dabei wurde auch auf eine gewisse Einheitlichkeit der Darstellung geachtet, was den Vergleich zwischen Programmvarianten erleichtert.
- Auch für die Laufstatistik-Ausgaben wurde einiger Aufwand getrieben. Solche Statistiken sind für Produktionsprogramme wertvoll, weil sie einen raschen Überblick ermöglichen. Defekte machen sich häufig schon auf der Statistik-Ebene bemerkbar, beispielsweise dann, wenn Zählerwerte nicht plausibel sind oder nicht zusammenpassen. Es kostet etwas Mühe, jeweils eine korrekte Zählung aller relevanten Partikel zu implementieren, aber das zahlt sich später auf jeden Fall aus. Konsistente und erwartbare Zählerwerte sind ein Gesundheitszeichen; alles andere ist dagegen ein Warnsignal.

Es gibt in diesem Band also einiges zu entdecken. Viel Spaß dabei!

Lemgo, den 1.7.2026

Übersichtsverzeichnis Band 4-B1

Inhaltsverzeichnis Band 4-B1

06/2022 Seerose im botanischen Garten von Palermo (Sizilien)

SQL_Example0 in Java

SQL_Example0 ist ähnlich wie ProcessWordList im Band 2-A1 aufgebaut und entstand daher zunächst als Klon, der nach und nach massiv umgebaut wurde. Dennoch sind noch Parallelen vorhanden, wie beispielsweise die Klassen `Trace` und `Counters`, deren Verwendung hier deshalb nicht mehr erklärt werden muss. Teilweise wurde Code aus anderen Programmen übernommen und für die Nutzung in SQL_Example0 angepasst. Folgende Anmerkungen erscheinen vorab als geboten:

- Die Klasse `SQL_ID` gibt analog zur Klasse `StreamCode` den von SQL_Example0 gelesenen oder bearbeiteten Entitäten eine numerische Identifikation. Diesen Entitäten entsprechen jedoch keine physischen Objekte, sondern SQL Statements, die zusammen mit den zugehörigen Handles und Zusatzdaten durch die SQL Services im SQL Array verwaltet werden. Die SQL_IDs und die Schnittstellenmethoden der Klasse `SQL_Util` entkoppeln das Anwendungsprogramm von der Datenbankschicht, was sich auch daran zeigt, dass keinerlei SQL-bezogene Java®-Library-Klassen seitens der SQL_Example0-Klassen importiert werden müssen.
- Die Phasen 1 und 2 bestehen aus Iterationen. Hierfür wurde das schon mehrfach praktizierte Verfahren genutzt, die Schleifen-Kerne als Objekte eigener Klassen zu instantiieren, nämlich als `Phase1Core` und `Phase2Core`. Das dient auch der Übersichtlichkeit des Codes.
- Die hohe Kontrolldichte im Programm und die rigorose Fehlerbehandlung sind auffallend. Jeder Fehler, der von den SQL Services rückgemeldet wird, führt zum Abbruch des Laufs. Andere Programme könnten da toleranter sein, aber hier ist dafür kein Freiraum. Aufgrund der zwangsläufig unterschiedlichen Datentypen der Rückmeldewerte gibt es zwei leicht verschiedene Methoden in der Klasse `SQL_Example0`, um zu kontrollieren, ob ein Aufruf fehlerfrei ablief. Diese Methoden werden auch von den Klassen `Phase1Core` und `Phase2Core` genutzt und daher dort importiert.
- In der Laufstatistik werden sowohl die programmintern geführten Zähler als auch die dazu teilweise korrespondierenden Zähler der Zugriffsschicht gezeigt. Daraus ergibt sich, dass auf das programminterne Zählen der per Cursor eingelesenen Zeilen verzichtet werden kann, weil die SQL Services dieselben Zahlenwerte abliefern.
- Wie bereits früher moniert, werden die Zugangsdaten einschließlich Passwort im Programmkopf offengelegt. Das ist für ein solches Programm tolerabel, in einer professionellen Produktionsumgebung allerdings nicht vertretbar. Es ist allerdings nicht Aufgabe dieses Buchs, hierfür Alternativen vorzustellen.
- Nachfolgend sind die Klassen wiedergegeben, aus denen SQL_Example0 besteht. Die Hilfsklassen `Trace` und `Counters`, sowie `SQL_ID` stehen am Anfang. Dann folgen die Klassen `SQL_Example0`, `Phase1Core` und `Phase2Core`. Diese Klassen sind so trivial, dass darauf verzichtet werden kann, sie hier zu zeigen. Die Abbildung „Klassendiagramm von SQL_Example0“ im Band 3-B1 reicht zur Orientierung völlig aus.

1.1 Hilfsklassen

1.1.1 Klasse Trace

```
package sql_example0;

/**
 * This enumeration defines the switches for the trace statements.
 */
public enum Trace
{
   DETAIL    ( true, "Display detail data"),
   METHOD    ( true, "Display method names");

   private final boolean traceFlag;
   private final String  purpose;

   Trace(boolean traceFlag, String purpose)
   {
      this.traceFlag = traceFlag;
      this.purpose   = purpose;
   }

   public boolean active()     { return traceFlag; }

   public String  getPurpose() { return purpose; }
}
```

Anmerkung

Die Trace-Flags sind hier noch für das später gezeigte Protokoll des Testlaufs auf true gesetzt, müssen dann aber abgeschaltet werden.

1.1.2 Klasse Counters

```
package sql_example0;

import commonmethods.Counter;

/**
 * The class Counters provides a container for the statistics counters. These
 * counters are public in order to avoid another layer of getter / setter methods.
 */
public class Counters
{
   public Counter ts1rowsRead;
   public Counter ts2rowsRead;
```

```
   public Counter ts3insertsFromTs1;
   public Counter ts3insertsFromTs2;
   public Counter ts3updatesFromTs2;

   /**
    * Constructor for objects of class Counters.
    */
   public Counters()
   {
      if (Trace.METHOD.active())
         { System.out.println("\n Counters: Constructor"); }

      this.ts1rowsRead       = new Counter(0);                    // op. 3
      this.ts2rowsRead       = new Counter(0);                    // op. 4
      this.ts3insertsFromTs1 = new Counter(0);                    // op. 12
      this.ts3insertsFromTs2 = new Counter(0);                    // op. 14
      this.ts3updatesFromTs2 = new Counter(0);                    // op. 16
   }
}
```

1.1.3 Klasse SQL_ID

```
package sql_example0;

/**
 * This enumeration defines the SQL IDs of SQL_Example0 and their roles.
 */
public enum SQL_ID
{
   TS1_SELECT  ( 0, "SELECT from TS1 without filter"),
   TS2_SELECT  ( 1, "SELECT from TS2 without filter"),
   TS3_INSERT  ( 2, "INSERT rows into TS3 from TS1 or TS2 rows"),
   TS3_UPDATE  ( 3, "UPDATE rows in TS3 from TS2 rows");

   private final int    statementID;
   private final String statementRole;

   SQL_ID(int statementID, String statementRole)
   {
      this.statementID   = statementID;
      this.statementRole = statementRole;
   }

   public int    getID()   { return statementID; }

   public String getRole() { return statementRole; }
```

```
}
```

Anmerkung

Die Konstanten in der Enumerations-Spalte `statementID` und die Rollen-Texte `statementRole` erhalten in dieser Klasse jeweils einen Namen. Erstere dienen dazu, Aufrufe der `SQL_Util`-Methoden mit dem jeweiligen aktuellen Parameterwert von `SQL_ID` zu bestücken, der jeweils mittels `getID()` ausgelesen wird. Der SQL Array in der Service-Klasse `SQL_Register` muss folglich mindestens vier Einträge besitzen. Mehr ist möglich, aber sinnlos.

1.2 Klasse SQL_Example0

```
package sql_example0;

import sql_service.SQL_Code;
import sql_service.SQL_Util;

/**
 * This formal class SQL_Example0 contains the main() method.
 */

public class SQL_Example0
{
   public static void main(String[] args)
   {
      String url         = "jdbc:mysql://localhost:3306/tuple_db" +
                           "?autoReconnect=true";
      String user        = "<user>";
      String password    = "<password>";
      String ts1select   = "SELECT Tuple_Key, Tuple_Value FROM TS1_0_View";
      String ts2select   = "SELECT Tuple_Key, Tuple_Value FROM TS2_0_View";
      String ts3insert   = "INSERT into TS3_0_Table (Tuple_Key, Tuple_Value) " +
                           "values (?, ?)";
      String ts3update   = "UPDATE TS3_0_Table " +
                           "set Tuple_Value = Tuple_Value + ? "+
                           "where Tuple_Key = ? ";
      int    result      = 0;
      int    ts1rowCount = 0;
      int    ts2rowCount = 0;

      System.out.println("*** SQL_Example0 ***");

      System.out.println("Trace settings");
      System.out.printf("%-23s: %b%n", Trace.METHOD.getPurpose(),
                        Trace.METHOD.active());
      System.out.printf("%-23s: %b%n", Trace.DETAIL.getPurpose(),
                        Trace.DETAIL.active());
```

```
// instantiate a Counters object: op. 3,4,12,14,16
Counters counters = new Counters();

// op. 27: open the connection and build the SQL array
result = SQL_Util.buildConnection(url, user, password, 4);

handleReturnedResult(result);

// op. 29: compile the TS1 SELECT
result = SQL_Util.compileSQL_Statement(SQL_ID.TS1_SELECT.getID(),
                                       ts1select, -1, -1, -1);
handleReturnedResult(result);

// op. 30: compile the TS2 SELECT
result = SQL_Util.compileSQL_Statement(SQL_ID.TS2_SELECT.getID(),
                                       ts2select, -1, -1, -1);
handleReturnedResult(result);

// op. 31: compile the TS3 INSERT
result = SQL_Util.compileSQL_Statement(SQL_ID.TS3_INSERT.getID(),
                                       ts3insert, -1, -1, -1);
handleReturnedResult(result);

// op. 32: compile the TS3 UPDATE
result = SQL_Util.compileSQL_Statement(SQL_ID.TS3_UPDATE.getID(),
                                       ts3update, -1, -1, -1);
handleReturnedResult(result);

// op. 1: open the TS1 cursor
result = SQL_Util.openCursor(SQL_ID.TS1_SELECT.getID());

handleReturnedResult(result);

// op. 2: open the TS2 cursor
result = SQL_Util.openCursor(SQL_ID.TS2_SELECT.getID());

handleReturnedResult(result);

// op.5: FETCH TS1 row from TS1 cursor
result = SQL_Util.fetchFromCursor(SQL_ID.TS1_SELECT.getID());

handleReturnedResult(result);

// op.7: FETCH TS2 row from TS2 cursor
result = SQL_Util.fetchFromCursor(SQL_ID.TS2_SELECT.getID());
```

```
handleReturnedResult(result);

// abnormal termination when TS2 is empty
if (result == SQL_Util.END_REACHED)
{
   System.out.println("++++ TS2 empty");

   SQL_Util.closeConnection();

   return;
}

// instantiate the Phase1Core
Phase1Core phase1core = new Phase1Core();

// I1: copy TS1 to TS3
while (SQL_Util.hasInputData(SQL_ID.TS1_SELECT.getID()))
   { phase1core.copyTS1rowToTS3(counters); }

// save the TS1 cursor's row count
ts1rowCount = SQL_Util.getRowCount(SQL_ID.TS1_SELECT.getID());

handleSavedResult();

// op. 23: close TS1 cursor
result = SQL_Util.closeCursor(SQL_ID.TS1_SELECT.getID());

handleReturnedResult(result);

// instantiate the Phase2Core
Phase2Core phase2core = new Phase2Core();

// I2: apply TS2 data to TS3
while (SQL_Util.hasInputData(SQL_ID.TS2_SELECT.getID()))
   { phase2core.applyTS2rowToTS3(counters); }

// save the TS2 cursor's row count
ts2rowCount = SQL_Util.getRowCount(SQL_ID.TS2_SELECT.getID());

handleSavedResult();

// op. 24: close TS2 cursor
result = SQL_Util.closeCursor(SQL_ID.TS2_SELECT.getID());

handleReturnedResult(result);

// op. 25: output SQL_Example0 statistics
```

```
   System.out.println("\nSQL_Example0 Statistics");
   System.out.println("=== Input (internal variables) ===");
   System.out.println("TS1 rows read         = " +
                      counters.ts1rowsRead.getValue());
   System.out.println("TS2 rows read         = " +
                      counters.ts2rowsRead.getValue());
   System.out.println("=== Output ===");
   System.out.println("TS3 inserts from TS1  = " +
                      counters.ts3insertsFromTs1.getValue());
   System.out.println("TS3 inserts from TS2  = " +
                      counters.ts3insertsFromTs2.getValue());
   System.out.println("TS3 updates from TS2  = " +
                      counters.ts3updatesFromTs2.getValue());
   System.out.println("=== SQL Service variables ===");
   System.out.println("TS1 rows read         = " + ts1rowCount);
   System.out.println("TS2 rows read         = " + ts2rowCount);
   System.out.println("TS1 SELECT exec calls = " +
                      SQL_Util.getExecCalls(SQL_ID.TS1_SELECT.getID()));
   System.out.println("TS2 SELECT exec calls = " +
                      SQL_Util.getExecCalls(SQL_ID.TS2_SELECT.getID()));
   System.out.println("TS3 INSERT exec calls = " +
                      SQL_Util.getExecCalls(SQL_ID.TS3_INSERT.getID()));
   System.out.println("TS3 UPDATE exec calls = " +
                      SQL_Util.getExecCalls(SQL_ID.TS3_UPDATE.getID()));

   // op. 28: close the connection
   result = SQL_Util.closeConnection();

   handleReturnedResult(result);

   System.out.println("*** Successful completion ***");

   // op. 26: stop (implicit)
}

static void handleReturnedResult(int result)
{
   String errorMessage;

   if (Trace.METHOD.active())
      { System.out.println("\n SQL_Example0: handleReturnedResult(" +
                           result + ")"); }

   if (result < SQL_Code.OK.getCode())
   {
      errorMessage = SQL_Util.getErrorMessage();
```

```
         System.out.println("Fatal error " + result + ": " +  errorMessage);

         System.exit(result);
      }
   }

   static void handleSavedResult()
   {
      int    errorCode;
      String errorMessage;

      if (Trace.METHOD.active())
         { System.out.println("\n SQL_Example0: handleSavedResult()"); }

      errorCode = SQL_Util.getErrorCode();

      if (errorCode < SQL_Code.OK.getCode())
      {
         errorMessage = SQL_Util.getErrorMessage();

         System.out.println("Fatal error " + errorCode + ": " +  errorMessage);

         System.exit(errorCode);
      }
   }
}
```

Anmerkung

Wenn TS2 leer ist, kann SQL_Example0 seine Aufgabe nicht erfüllen. Der Lauf wird dann abgebrochen. Die Aufrufe von `SQL_Util.closeXXX()` wurden allein aus Testgründen so ausführlich eingefügt. Die Methode `SQL_Util.closeConnection()` schließt nämlich alle offenen Beziehungen zu MySQL® intern.

1.3 Klasse Phase1Core

```
package sql_example0;

import static sql_example0.SQL_Example0.handleReturnedResult;
import static sql_example0.SQL_Example0.handleSavedResult;
import sql_service.SQL_Util;

/**
 * The class Phase1Core is the core of the I1 loop
 */
public class Phase1Core
{
   /**
```

```
 * constructor for objects of class Phase1Core
 */
public Phase1Core() { }

/**
 * copyTS1rowToTS3() implements the structure block "store TS1 row in TS3".
 */
public void copyTS1rowToTS3(Counters counters)
{
   String ts1tupleKey;
   int    ts1tupleValue;
   int    result;

   if (Trace.METHOD.active())
      { System.out.println("\n Phase1Core: copyTS1rowToTS3()"); }

   counters.ts1rowsRead.incByOne();                              // op. 6

   // op. 9: get the TS1 tuple key and put it in the 1st free variable
   // parameter of the INSERT statement
   ts1tupleKey = SQL_Util.getStringColumnValue(SQL_ID.TS1_SELECT.getID(), 1);

   handleSavedResult();

   result = SQL_Util.setStringPlaceholder(SQL_ID.TS3_INSERT.getID(), 1,
                                          ts1tupleKey);
   handleReturnedResult(result);

   // op. 10: get the TS1 tuple value and put it in the 2nd free variable
   // parameter of the INSERT statement
   ts1tupleValue = SQL_Util.getIntColumnValue(SQL_ID.TS1_SELECT.getID(), 2);

   handleSavedResult();

   result = SQL_Util.setIntPlaceholder(SQL_ID.TS3_INSERT.getID(), 2,
                                       ts1tupleValue);
   handleReturnedResult(result);

   // op. 11: INSERT the new row into TS3
   result = SQL_Util.executeUpdate(SQL_ID.TS3_INSERT.getID());

   handleReturnedResult(result);

   counters.ts3insertsFromTs1.incByOne();                        // op. 13

   // op.5: FETCH TS1_row from TS1 cursor
   result = SQL_Util.fetchFromCursor(SQL_ID.TS1_SELECT.getID());
```

```
      handleReturnedResult(result);
   }
}
```

1.4 Klasse Phase2Core

```
package sql_example0;

import static sql_example0.SQL_Example0.handleReturnedResult;
import static sql_example0.SQL_Example0.handleSavedResult;
import sql_service.SQL_Util;

/**
 * The class Phase2Core is the core of the I2 loop
 */
public class Phase2Core
{
   /**
    * constructor for objects of class Phase2Core
    */
   public Phase2Core() { }

   /**
    * applyTS2rowToTS3() implements the structure block
    * "pr(ocess) T2 / TS3 match case".
    */
   public void applyTS2rowToTS3(Counters counters)
   {
      String  ts2tupleKey;
      int     ts2tupleValue;
      int     result;
      int     rowsUpdated;
      boolean success;

      if (Trace.METHOD.active())
         { System.out.println("\n Phase2Core: applyTS2rowToTS3()"); }

      // Procedure Statement 1

      counters.ts2rowsRead.incByOne();                          // op. 8

      // op. 33: get the TS2 tuple key and put it in the 2nd free variable
      // parameter of the UPDATE statement
      ts2tupleKey = SQL_Util.getStringColumnValue(SQL_ID.TS2_SELECT.getID(), 1);

      handleSavedResult();
```

```
result = SQL_Util.setStringPlaceholder(SQL_ID.TS3_UPDATE.getID(), 2,
                                       ts2tupleKey);
handleReturnedResult(result);

// op. 34: get the TS2 tuple value and put it in the 1st free variable
// parameter of the UPDATE statement
ts2tupleValue = SQL_Util.getIntColumnValue(SQL_ID.TS2_SELECT.getID(), 2);

handleSavedResult();

result = SQL_Util.setIntPlaceholder(SQL_ID.TS3_UPDATE.getID(), 1,
                                    ts2tupleValue);
handleReturnedResult(result);

// op. 18: UPDATE the TS3 row
result = SQL_Util.executeUpdate(SQL_ID.TS3_UPDATE.getID());

handleReturnedResult(result);

// prepare for S4
rowsUpdated = SQL_Util.getRowCount(SQL_ID.TS3_UPDATE.getID());

handleSavedResult();

// S4: TS3 row missing?
if (rowsUpdated == 0)
{
   success = false;                                          // op. 20

   if (Trace.DETAIL.active())
      { System.out.println("\t Row not found ==> INSERT necessary"); }
}
else
{
   success = true;                                           // op. 19
   counters.ts3updatesFromTs2.incByOne();                    // op. 17

   if (Trace.DETAIL.active())
      { System.out.println("\t Row found ==> UPDATE successful"); }
}

// Procedure Statement 2

// S5: TS3 row not found?
if (success == false)
{
```

```
         // op. 21: put the TS2 tuple key in the 1st free variable parameter
         // of the INSERT statement
         result = SQL_Util.setStringPlaceholder(SQL_ID.TS3_INSERT.getID(), 1,
                                                ts2tupleKey);
         handleReturnedResult(result);

         // op. 22: put the TS2 tuple value in the 2nd free variable parameter
         // of the INSERT statement
         result = SQL_Util.setIntPlaceholder(SQL_ID.TS3_INSERT.getID(), 2,
                                             ts2tupleValue);
         handleReturnedResult(result);

         // op. 11: INSERT the new row into TS3
         result = SQL_Util.executeUpdate(SQL_ID.TS3_INSERT.getID());

         handleReturnedResult(result);

         counters.ts3insertsFromTs2.incByOne();                  // op. 15
      }

      // op.7: FETCH TS2 row from TS2 cursor
      result = SQL_Util.fetchFromCursor(SQL_ID.TS2_SELECT.getID());

      handleReturnedResult(result);
   }
}
```

1.5 Test von SQL_Example0

Die nachfolgenden Ausschnitte aus dem kompletten Testprotokoll enthalten fast nur Inhalte, die von den SQL Services generiert wurden. Die Klassen von SQL_Example0 liefern selbst nur sehr wenig Ausgabematerial.

Die ersten Schritte des Programms inklusive der Op. 27 werden im folgenden Unterkapitel protokolliert.

1.5.1 Protokoll der Initialphase

```
*** SQL_Example0 ***
Trace settings
Display method names   : true
Display detail data    : true

 Counters: Constructor

 SQL_Util: buildConnection()
        SQL_Entries = 4

 SQL_Register: getConnectionHandle()
```

```
SQL_Register: storeConnectionHandle()

SQL_Register: setupSQL_Array(4)

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(0)
```

Dieser Protokollabschnitt entspricht ab `SQL_Util: buildConnection()` der Abbildung „Verbindungsaufbau zu MySQL und Anlegen des SQL Array" im Band-3-B1-Kapitel „SQL Services". Die finale Prüfung per `handleReturnedResult()` wird in dieser Grafik nicht erwähnt, weil das zum Verständnis der Op. 27 nichts beiträgt. Unmittelbar danach folgt das Kompilieren des ersten SQL Statements (Op. 29):

```
SQL_Util: compileSQL_Statement()
       SQL_ID   = 0

SQL_Register: getConnectionHandle()

SQL_Util: checkSQL_ID() for compileSQL_Statement()
       SQL_ID   = 0

SQL_Register: getSQL_Statement(0)
       SQL_Stmt = null

SQL_Register: register(0, SQL_Stmt)
       SQL_Stmt = SELECT Tuple_Key, Tuple_Value FROM TS1_0_View

SQL_Register: showSQL_Entry(0)
SQL Array Entry 0:
  SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
  exec calls : 0
  row count  : 0
  read status:

SQL_Register: handleResults(0, OK)

result = 0

SQL_Register: register(0, preparedStmt)

SQL_Register: showSQL_Entry(0)
SQL Array Entry 0:
  SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
  Prepared Statement handle exists
  exec calls : 0
  row count  : 0
  read status:
```

```
SQL_Register: handleResults(0, OK)
       result = 0

SQL_Example0: handleReturnedResult(0)
```

Die Aufrufergebnisse von `showSQL_Entry()` ermöglichen es, die Entwicklung des betreffenden Eintrags nachzuvollziehen. Diese Sequenz entspricht der Abbildung „Kompilieren des TS1 SELECT" im Band-3-B1-Kapitel „SQL Services". Die nachfolgenden drei Compile-Protokolle sehen nahezu identisch aus und werden deshalb hier weggelassen. Danach folgt das Öffnen des TS1 Cursor (Op. 1):

```
SQL_Util: openCursor()
       SQL_ID   = 0

SQL_Util: checkSQL_ID() for openCursor()
       SQL_ID   = 0

SQL_Register: getPreparedStatement(0)

SQL_Register: incExecCalls(0)

SQL_Register: showSQL_Entry(0)
SQL Array Entry 0:
  SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
  Prepared Statement handle exists
  exec calls : 1
  row count  : 0
  read status:

SQL_Register: register(0, resultSet)

SQL_Register: showSQL_Entry(0)
SQL Array Entry 0:
  SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
  Prepared Statement handle exists
  Result Set handle exists
  exec calls : 1
  row count  : 0
  read status:

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(0)
```

Der Aufruf von `incExecCalls()` erhöht den Wert von `execCalls` nach der erfolgreichen Ausführung von `executeQuery()` auf 1. Das von der Java-Zugriffsschicht nach der Interaktion mit MySQL rückgemeldete Handle des Result Set wird nun eingetragen. Dieser Ablauf entspricht dem oberen Teil der Abbildung „Öffnen

des TS1 Cursor und Lesen der ersten TS1-Zeile“ im Band-3-B1-Kapitel „SQL Services“. Im Programm folgt zunächst das Öffnen des zweiten Cursor und dann erst der erste Lesezugriff auf TS1 (Op. 5):

```
SQL_Util: fetchFromCursor()
        SQL_ID   = 0

 SQL_Util: checkSQL_ID() for fetchFromCursor()
        SQL_ID   = 0

 SQL_Register: getResultSet(0)

 SQL_Register: storeFetchResults(0, true)

 SQL_Register: showSQL_Entry(0)
 SQL Array Entry 0:
   SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
   Prepared Statement handle exists
   Result Set handle exists
   exec calls : 1
   row count  : 1
   read status: D

 SQL_Register: handleResults(0, OK)

 SQL_Example0: handleReturnedResult(2)
```

`execCalls` enthält immer noch 1, weil für den FETCH kein weiterer Exec-Aufruf erforderlich ist. Da mit `next()` erstmals eine Zeile eingelesen wurde, steht der `rowCount` des Eintrags nun auf 1 und der `readStatus` hat den Inhalt `'D'` = Data. Das drückt sich auch im Rückgabewert 2 der Methode aus, dem die Konstante `ROW_FETCHED` in `SQL_Util` entspricht. Der danach folgende erste Lesezugriff auf TS2 verläuft analog.

1.5.2 Protokoll der Phase 1

SQL_Example0 fragt anschliessend den `readStatus` des TS1 Cursor mittels `hasInputData()` ab und tritt in die I1-Schleife über `copyTS1rowToTS3()` ein:

```
 SQL_Register: hasInputData(0)
        result = true

 Phase1Core: copyTS1rowToTS3()
```

Diese Methode hat die Aufgabe, die TS1-Zeilen einzeln nach TS3 zu kopieren.

Hinweise

- Eine Besonderheit der Nutzung der Java SQL-Schnittstelle ist, dass das Vorlesen nicht zwingend auch die Daten der betreffenden Zeile in den Adressbereich des Programms bringt. Diese können vielmehr nach und nach spaltenweise ausgelesen werden, wobei erstens kein Vollständigkeitszwang besteht und zweitens diese Schnittstellenaufrufe in anderen Programmkomponenten als derjenigen stattfinden können, die den `next()`-Aufruf zum Vor- oder Nachlesen veranlasst hat. Wenn es also zunächst nur darum geht, ob überhaupt eine Zeile gelesen wurde, muss die lesende Methode die Spalteninhalte nicht – beispielsweise in einer statischen Klasse wie `InputUtil` von ProcessWordList – zwischenpuffern. Nur dann, wenn der Cursor noch während der Auswertung der betreffenden Zeile fortgeschaltet wird, ist diese Pufferung zwingend erforderlich, denn sonst wären diese Daten verloren.
- Für jeden Datentyp, der in einer relationalen Tabelle auftreten kann, gibt es je eine Getter- und eine Update-Methode auf der Zeilenebene. Außerdem erfordert das Zuweisen realer Werte auf Platzhalterpositionen eine Setter-Methode pro Datentyp. An dieser Stelle der Entwicklung der SQL Services waren nur jeweils zwei Getter- und Setter-Methoden (für `String` und `int`) definiert, nämlich `getStringColumnValue()` und `getIntegerColumnValue()`, `setStringPlaceholder()` und `setIntPlaceholder()`.

`copyTS1rowToTS3()` beschafft zuerst den Wert des Zeilenschlüssels Tuple_Key und überträgt ihn auf die erste Position der VALUE-Klausel des INSERT Statement:

```
SQL_Util: getStringColumnValue()
       SQL_ID   = 0

SQL_Util: checkSQL_ID() for getStringColumnValue()
       SQL_ID   = 0

SQL_Register: getResultSet(0)

SQL_Register: handleResults(0, OK)
       columnValue = A

SQL_Example0: handleSavedResult()

SQL_Util: setStringPlaceholder()
       SQL_ID   = 2
       position = 1
       string   = A

SQL_Util: checkSQL_ID() for setStringPlaceholder()
       SQL_ID   = 2

SQL_Register: getPreparedStatement(2)

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(0)
```

Danach folgt dasselbe analog für den Tuple_Value:

```
SQL_Util: getIntColumnValue()
       SQL_ID   = 0

SQL_Util: checkSQL_ID() for getIntColumnValue()
       SQL_ID   = 0

SQL_Register: getResultSet(0)

SQL_Register: handleResults(0, OK)
       columnValue = -3

SQL_Example0: handleSavedResult()

SQL_Util: setIntPlaceholder()
       SQL_ID   = 2
       position = 2
       value    = -3

SQL_Util: checkSQL_ID() for setIntPlaceholder()
       SQL_ID   = 2

SQL_Register: getPreparedStatement(2)

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(0)
```

Auch das verlief fehlerfrei. Nun kann der INSERT stattfinden, gefolgt vom Inkrementieren des TS3-Zeilenzählers:

```
SQL_Util: executeUpdate()
       SQL_ID   = 2

SQL_Util: checkSQL_ID() for executeUpdate()
       SQL_ID   = 2

SQL_Register: getPreparedStatement(2)

SQL_Register: incExecCalls(2)

SQL_Register: showSQL_Entry(2)
SQL Array Entry 2:
  SQL Statement: INSERT into TS3_0_Table (Tuple_Key, Tuple_Value) values (?, ?)
  Prepared Statement handle exists
  exec calls : 1
  row count  : 0
```

```
  read status:

SQL_Register: updateRowCount(2, 1)

SQL_Register: showSQL_Entry(2)
SQL Array Entry 2:
  SQL Statement: INSERT into TS3_0_Table (Tuple_Key, Tuple_Value) values (?, ?)
  Prepared Statement handle exists
  exec calls : 1
  row count  : 1
  read status:

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(0)
```

Abschließend liest die Methode `copyTS1rowToTS3()` in TS1 nach:

```
SQL_Util: fetchFromCursor()
       SQL_ID   = 0

SQL_Util: checkSQL_ID() for fetchFromCursor()
       SQL_ID   = 0

SQL_Register: getResultSet(0)

SQL_Register: storeFetchResults(0, true)

SQL_Register: showSQL_Entry(0)
SQL Array Entry 0:
  SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
  Prepared Statement handle exists
  Result Set handle exists
  exec calls : 1
  row count  : 2
  read status: D

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(2)
```

Nach der Rückkehr zur `main()` in SQL_Example0 folgt die nächste Abfrage des `readStatus` mittels `hasInputData()` und der Wiedereintritt in `copyTS1rowToTS3()` und so weiter und so fort bis zum Ende der TS1-Ergebnismenge:

```
[…]
 SQL_Util: fetchFromCursor()
```

```
       SQL_ID   = 0

SQL_Util: checkSQL_ID() for fetchFromCursor()
       SQL_ID   = 0

SQL_Register: getResultSet(0)

SQL_Register: storeFetchResults(0, false)

SQL_Register: showSQL_Entry(0)
SQL Array Entry 0:
  SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
  Prepared Statement handle exists
  Result Set handle exists
  exec calls : 1
  row count  : 8
  read status: E

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(1)

SQL_Register: hasInputData(0)
       result = false
```

Damit endet die I1-Schleife. Die `main()` beschafft nun den Zeilenzähler des TS1 Cursor, bevor dieser durch die Op. 23 verloren geht:

```
SQL_Register: getRowCount(0)
       rowCount = 8

SQL_Example0: handleSavedResult()
```

Das Schließen des TS1 Cursor hinterlässt die folgende Protokollstrecke:

```
SQL_Util: closeCursor()
       SQL_ID   = 0

SQL_Util: checkSQL_ID() for closeCursor()
       SQL_ID   = 0

SQL_Register: getResultSet(0)

SQL_Register: resetRowCount(0)

SQL_Register: showSQL_Entry(0)
SQL Array Entry 0:
```

```
   SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
   Prepared Statement handle exists
   Result Set handle exists
   exec calls : 1
   row count  : 0
   read status: E

 SQL_Register: deregister(0, resultSet)

 SQL_Register: showSQL_Entry(0)
 SQL Array Entry 0:
   SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
   Prepared Statement handle exists
   exec calls : 1
   row count  : 0
   read status: E

 SQL_Register: handleResults(0, OK)

 SQL_Example0: handleReturnedResult(0)
```

Der zweite Aufruf von `showSQL_Entry()` zeigt, dass das Query Handle nach dem `deregister()`-Aufruf nicht mehr existiert.

1.5.3 Protokoll der Phase 2

SQL_Example0 fragt nun den `readStatus` des TS2 Cursor mittels `hasInputData()` ab und tritt in die I2-Schleife über `applyTS2rowToTS3()` ein:

```
 SQL_Register: hasInputData(1)
        result = true


 Phase2Core: applyTS2rowToTS3()
```

Diese Methode hat die Aufgabe, die TS2-Zeilen auf den TS3-Zwischenbestand anzuwenden. Sie erhöht den TS2-Zeilenzähler und bestückt zunächst den `Tuple_Key`-Platzhalter des UPDATE Statement mit dem dazu korrespondierenden Wert aus TS2:

```
 SQL_Util: getStringColumnValue()
        SQL_ID   = 1

 SQL_Util: checkSQL_ID() for getStringColumnValue()
        SQL_ID   = 1

 SQL_Register: getResultSet(1)
```

```
SQL_Register: handleResults(0, OK)
       columnValue = N

SQL_Example0: handleSavedResult()

SQL_Util: setStringPlaceholder()
       SQL_ID   = 3
       position = 2
       string   = N

SQL_Util: checkSQL_ID() for setStringPlaceholder()
       SQL_ID   = 3

SQL_Register: getPreparedStatement(3)

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(0)
```

Analog dazu folgt die Zuweisung zum `Tuple_Value`-Platzhalter:

```
SQL_Util: getIntColumnValue()
       SQL_ID   = 1

SQL_Util: checkSQL_ID() for getIntColumnValue()
       SQL_ID   = 1

SQL_Register: getResultSet(1)

SQL_Register: handleResults(0, OK)
       columnValue = 8

SQL_Example0: handleSavedResult()

SQL_Util: setIntPlaceholder()
       SQL_ID   = 3
       position = 1
       value    = 8

SQL_Util: checkSQL_ID() for setIntPlaceholder()
       SQL_ID   = 3

SQL_Register: getPreparedStatement(3)

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(0)
```

Unmittelbar danach kann das UPDATE Statement ausgeführt werden:

```
 SQL_Util: executeUpdate()
        SQL_ID   = 3

 SQL_Util: checkSQL_ID() for executeUpdate()
        SQL_ID   = 3

 SQL_Register: getPreparedStatement(3)

 SQL_Register: incExecCalls(3)

 SQL_Register: showSQL_Entry(3)
 SQL Array Entry 3:
   SQL Statement: UPDATE TS3_0_Table set Tuple_Value = Tuple_Value + ? where Tuple_
Key = ?
   Prepared Statement handle exists
   exec calls : 1
   row count  : 0
   read status:

 SQL_Register: updateRowCount(3, 1)

 SQL_Register: showSQL_Entry(3)
 SQL Array Entry 3:
   SQL Statement: UPDATE TS3_0_Table set Tuple_Value = Tuple_Value + ? where Tuple_
Key = ?
   Prepared Statement handle exists
   exec calls : 1
   row count  : 1
   read status:

 SQL_Register: handleResults(0, OK)

 SQL_Example0: handleReturnedResult(0)

 SQL_Register: getRowCount(3)
        rowCount = 1

 SQL_Example0: handleSavedResult()
        Row found ==> UPDATE successful
```

Der vom UPDATE rückgemeldete `rowCount`-Wert 1 signalisiert, dass der Zugriff erfolgreich war. Folglich kann sofort die nächste TS2-Zeile nachgelesen werden. Auch hier kehrt das Programm zur `main()` zurück, prüft den Zugriffserfolg und tritt dann wieder in `applyTS2rowToTS3()` ein. Das muss hier nicht gezeigt werden, ebensowenig wie die Strecke bis zum UPDATE und unmittelbar danach.

Schon bei der zweiten TS2-Zeile mit dem Schlüsselwert `'I'` ist der UPDATE erfolglos:

```
SQL_Register: getRowCount(3)
       rowCount = 0

SQL_Example0: handleSavedResult()
       Row not found ==> INSERT necessary
```

Analog zur Phase 1 überträgt `applyTS2rowToTS3()` anschließend den Zeilenschlüssel auf die erste INSERT-Platzhalterposition und den Zeilenwert auf die zweite Position. Das Protokoll der INSERT-Strecke lautet wie folgt:

```
SQL_Util: executeUpdate()
       SQL_ID   = 2

SQL_Util: checkSQL_ID() for executeUpdate()
       SQL_ID   = 2

SQL_Register: getPreparedStatement(2)

SQL_Register: incExecCalls(2)

SQL_Register: showSQL_Entry(2)
SQL Array Entry 2:
  SQL Statement: INSERT into TS3_0_Table (Tuple_Key, Tuple_Value) values (?, ?)
  Prepared Statement handle exists
  exec calls : 9
  row count  : 1
  read status:

SQL_Register: updateRowCount(2, 1)

SQL_Register: showSQL_Entry(2)
SQL Array Entry 2:
  SQL Statement: INSERT into TS3_0_Table (Tuple_Key, Tuple_Value) values (?, ?)
  Prepared Statement handle exists
  exec calls : 9
  row count  : 1
  read status:

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(0)
```

Aufmerksamen Betrachter(inne)n fällt vermutlich auf, dass die beiden Ausgabeblöcke von `showSQL_Entry()` dieselben Inhalte zeigen. Der bisherige `rowCount`-Wert 1, der vom letzten INSERT der Phase 1 stammt, wird mit 1 überschrieben, weil genau eine Zeile eingefügt wurde.

Danach wiederholen sich diese Schritte mit den weiteren TS2-Zeilen. Am Ende der TS2-Ergebnismenge folgt dieser Protokollabschnitt:

```
SQL_Util: fetchFromCursor()
       SQL_ID   = 1

SQL_Util: checkSQL_ID() for fetchFromCursor()
       SQL_ID   = 1

SQL_Register: getResultSet(1)

SQL_Register: storeFetchResults(1, false)

SQL_Register: showSQL_Entry(1)
SQL Array Entry 1:
  SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS2_0_View
  Prepared Statement handle exists
  Result Set handle exists
  exec calls : 1
  row count  : 7
  read status: E

SQL_Register: handleResults(0, OK)

SQL_Example0: handleReturnedResult(1)
```

Das beendet die I2-Schleife. Die main() beschafft den TS2-Zeilenzähler und schliesst den TS2 Cursor.

```
SQL_Register: hasInputData(1)
       result = false

SQL_Register: getRowCount(1)
       rowCount = 7

SQL_Example0: handleSavedResult()
```

1.5.4 Protokoll der Finalphase

```
SQL_Util: closeCursor()
       SQL_ID   = 1

SQL_Util: checkSQL_ID() for closeCursor()
       SQL_ID   = 1

SQL_Register: getResultSet(1)

SQL_Register: resetRowCount(1)

SQL_Register: showSQL_Entry(1)
SQL Array Entry 1:
```

```
   SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS2_0_View
   Prepared Statement handle exists
   Result Set handle exists
   exec calls : 1
   row count  : 0
   read status: E

 SQL_Register: deregister(1, resultSet)

 SQL_Register: showSQL_Entry(1)
 SQL Array Entry 1:
   SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS2_0_View
   Prepared Statement handle exists
   exec calls : 1
   row count  : 0
   read status: E

 SQL_Register: handleResults(0, OK)

 SQL_Example0: handleReturnedResult(0)
```

Nun folgen nur noch die Statistik-Ausgabe und das Schließen der Connection, welches intern alle noch offenen Datenbankbeziehungen schließt, zu denen Handles gehören:

```
SQL_Example0 Statistics
=== Input (internal variables) ===
TS1 rows read         = 8
TS2 rows read         = 7
=== Output ===
TS3 inserts from TS1 = 8
TS3 inserts from TS2 = 2
TS3 updates from TS2 = 5
=== SQL Service variables ===
TS1 rows read         = 8
TS2 rows read         = 7

 SQL_Register: getExecCalls(0)
       execCalls = 1
TS1 SELECT exec calls = 1

 SQL_Register: getExecCalls(1)
       execCalls = 1
TS2 SELECT exec calls = 1

 SQL_Register: getExecCalls(2)
       execCalls = 10
TS3 INSERT exec calls = 10
```

```
 SQL_Register: getExecCalls(3)
       execCalls = 7
TS3 UPDATE exec calls = 7

 SQL_Util: closeConnection()

 SQL_Register: getConnectionHandle()
Display SQL array contents at closeConnection()

 SQL_Register: showSQL_Array()

 SQL_Register: showSQL_Entry(0)
 SQL Array Entry 0:
   SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS1_0_View
   Prepared Statement handle exists
   exec calls : 1
   row count  : 0
   read status: E

 SQL_Register: showSQL_Entry(1)
 SQL Array Entry 1:
   SQL Statement: SELECT Tuple_Key, Tuple_Value FROM TS2_0_View
   Prepared Statement handle exists
   exec calls : 1
   row count  : 0
   read status: E

 SQL_Register: showSQL_Entry(2)
 SQL Array Entry 2:
   SQL Statement: INSERT into TS3_0_Table (Tuple_Key, Tuple_Value) values (?, ?)
   Prepared Statement handle exists
   exec calls : 10
   row count  : 1
   read status:

 SQL_Register: showSQL_Entry(3)
 SQL Array Entry 3:
   SQL Statement: UPDATE TS3_0_Table set Tuple_Value = Tuple_Value + ? where Tuple_
Key = ?
   Prepared Statement handle exists
   exec calls : 7
   row count  : 1
   read status:
       SQL_Entries = 4

 SQL_Register: getSQL_Entry(0)
```

```
 SQL_Register: deregister(0)

 SQL_Register: handleResults(0, OK)

 SQL_Register: getSQL_Entry(1)

 SQL_Register: deregister(1)

 SQL_Register: handleResults(0, OK)

[…] (dito für die SQL IDs 2 und 3)

 SQL_Register: storeConnectionHandle()

 SQL_Register: resetSQL_Array()

 SQL_Register: handleResults(0, OK)

 SQL_Example0: handleReturnedResult(0)
*** Successful completion ***
```