

JDBC

Referenzen

- Dieser Kurs basiert auf der JDBC Spezifikation 2.x und 3.0, Oktober 2001 von Sun. Wenn in diesem Kurs ausschliessliche Eigenschaften von JDBC 3.0 zur Sprache kommen, ist dies entsprechend notiert.
- JDK 1.4 referenziert auf die JDBC-Spezifikation 3.0 mit den AP's `java.sql.*` und `javax.sql.*`
- Viele Datenbanken-Hersteller¹ und die Plattform J2EE 1.3 referenzieren im Moment auf die JDBC-Spezifikation 2.1, mit den Erweiterungen aus dem Optional Package 2.0 `javax.sql.*`
- JDBC 2.x und 3.0 beziehen sich auf SQL 99
- Buch: "Java in Datenbanksystemen"; Petkovic, Brüderl; Addison-Wesley, 2002.

JDBC 3.0 ist im Wesentlichen eine Abrundung und Konsolidierung von JDBC 2.1 und dem Optional Package 2.0. Einige kleinere Änderungen im Funktionsumfang sind hinzugekommen. Ausserdem kann JDBC 3.0 auf die definitive Version von SQL99 abstellen.

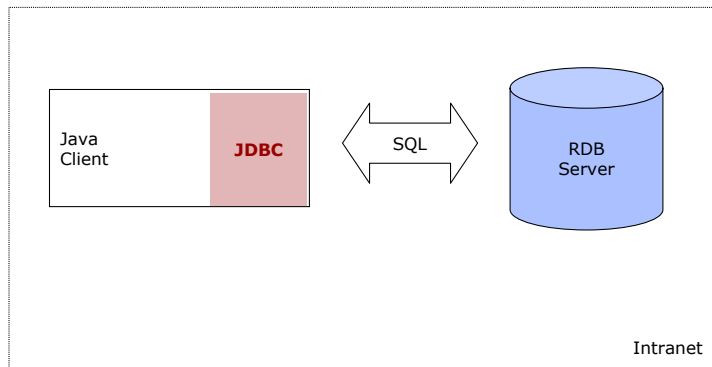
¹ Im Speziellen die bei den Übungen verwendeten Treiber von Cloudscape und jConnect5 für Sybase.

Was ist JDBC ?

- JDBC™ = Java Database Connectivity. Vergleichbar mit ODBC (Open Database Connectivity von Microsoft für C-Programme)
- JDBC ist ein low-level oder call-level API mit drei Funktionen
 - Verbindung zu einer Datenbank herstellen
 - SQL-Befehle absetzen
 - Resultate verarbeiten
- JDBC *ermöglicht* einen produktunabhängigen Datenbankzugriff, *erzwingt* ihn aber nicht.
- JDBC berücksichtigt sowohl die Ansprüche für den Direktzugriff auf eine Datenbank, wie diejenigen von Komponenten-Herstellern, welche gegen die Datenbank und gegen den Client standardisierte Interfaces haben wollen.

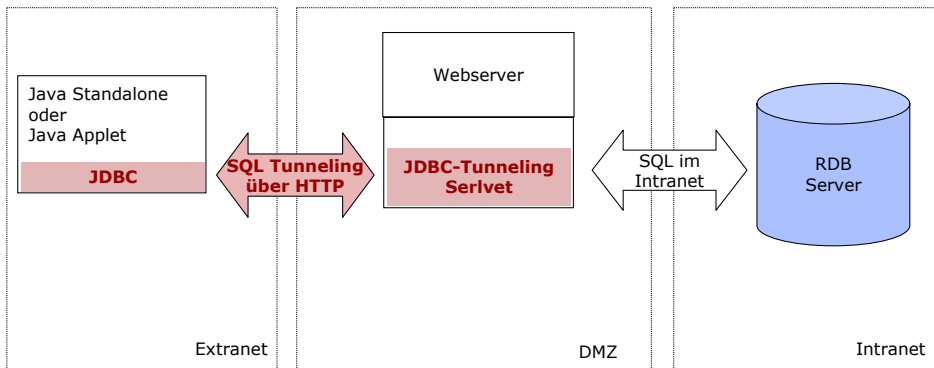
JDBC stellt grundsätzlich nur ein Interface für das "Wie" der Kommunikation mit DB-Servern zur Verfügung. Über den Befehl `Statement.execute()` können im Prinzip beliebige *Strings* an das Datenbanksystem übergeben werden. Allerdings bestehen Abhängigkeiten zwischen DB-System und JDBC z.B. bei der Datentypumwandlung mit `getXXX()` und `setXXX()` Funktionen. JDBC wurde deshalb vor dem Hintergrund von SQL-99 entworfen.

2-Tier Client/Server



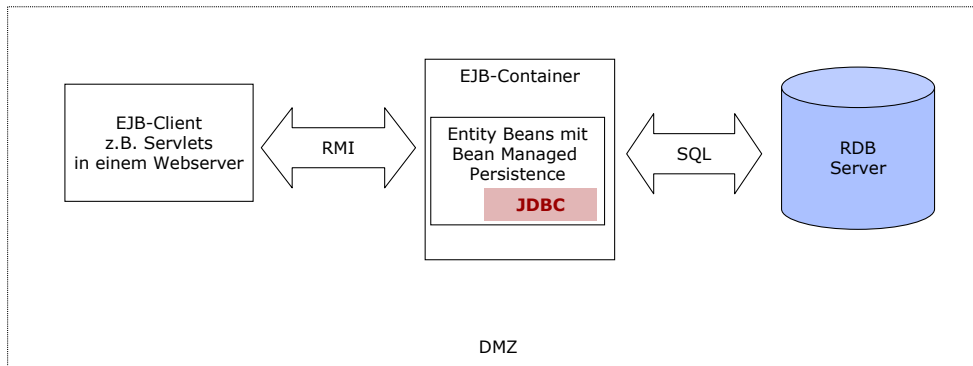
Driver Typ 1, 2 und 4 geeignet

3-Tier Tunneling



Driver Typ 3 geeignet

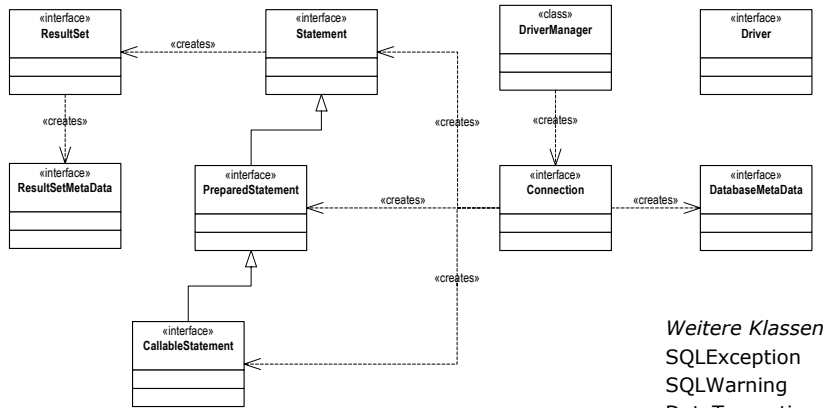
EJB Server



Driver in EJB-Server integriert

JDBC stellt für die Middleware-Entwickler verschiedene DataSource- und Connection-Interfaces zur Verfügung, welche flexibler und applikationsunabhängiger mit NamingServices zusammenarbeiten können als das direkte Arbeiten mit Drivern und einem DriverManager.

Das Package java.sql



Weitere Klassen
SQLException
SQLWarning
DataTruncation
Date
Time
Timestamp
Types
DriverPropertyInfo

Das Package java.sql gehört zum Sprachumfang von Java

Das JDBC API enthält nahezu nur Interface-Definitionen. Die eigentlichen Implementationsklassen werden von einem Datenbank- oder Dritthersteller vertrieben.

Die wichtigste Klasse ist der DriverManager. Bei ihm können die jeweiligen produktespezifischen Treiber, z.B. für Oracle, ODBC, SQL Server, Sybase etc. registriert werden.

Ein Beispiel

```
// Driver registrieren.
    SybDriver d = new SybDriver();
    DriverManager.registerDriver(d);

// Verbindungsaufbau
Connection con = DriverManager.getConnection(
    "jdbc:sybase:Tds:swssun00:1534", "schmd", "workshop" );

// Statementobjekt erzeugen
Statement stmt = con.createStatement();

try {
    // Query absetzen
    ResultSet rs = stmt.executeQuery("SELECT name FROM person");
    // Resultat verarbeiten
    while (rs.next()) {
        String name = rs.getString( 1 );
        System.out.println( name );
    }
} catch ( SQLException e ) {
    System.out.println( e );
    con.rollback();
}
```

Connection-URI
jdbc:name:subname:params
jdbc:odbc:source:params

Typkonversion von
JDBC/SQL nach Java

Es können mehrer Treiber registriert werden. Beim Verbindungsaufbau ruft der DriverManager der Reihe nach die Methode connect(url, props) seiner registrierten Treiber auf. Der Treiber prüft seinerseits, ob er mit dem url etwas anfangen kann. Wenn ja, baut er die Verbindung zur angegeben Datenbank oder zum angegebenen DBMS auf. Wenn nein, gibt er null zurück. Der DriverManager fragt in diesem Fall den nächsten registrierten Treiber an.

Die Methode getString() ist eine von vielen getXXX() Methoden zum Abholen von Daten in verschiedenste Java Basistypen und Java Klassen. Siehe API Doc der Klasse ResultSet.

Die Klasse Statement stellt verschiedene Methoden für das Absetzen von SQL-Befehlen zur Verfügung:

executeQuery("SQL String") ist für select-Abfragen vorgesehen.

executeUpdate("SQL String") ist für insert-, delete- und update-Befehle, sowie für DDL-Befehle vorgesehen.

execute("SQL String") ist für beliebige SQL-Befehle gedacht, insbesondere stored-procedure, welche mehrere SQL-Befehle unterschiedlichster Art enthalten können.

Der Connection-URI muss immer aus 3 Teilen bestehen:

jdbc:*subprotocol:params*

Der erste Teil ist fix. Der zweite Teil ist meist der Name eines Datenbankproduktes oder eines Middleware-Herstellers z.B: **sybase, oracle, openlink**. Der zweite Teil kann auch das Schlüsselwort **odbc** sein. Der dritte Teil kann vom Hersteller selbst strukturiert werden. i.a. kann dort der Name einer Datenbank oder einer Datenquelle stehen. Username und Passwort werden häufig nicht über den URI, sondern über das Property Objekt an den Treiber weitergeleitet. Die Treiber-Hersteller sind jedoch in der Definition der Syntax und der Semantik für den dritten Teil des URL frei.

Wenn der URI eine ODBC Datequelle bezeichnet, besteht der dritte Teil aus dem Namen der Datenquelle (logischerName für eine Datenbank) und allfälligen Parametern zum Aufbau der Verbindung wie Benutzername und Passwort. Beispiel:
jdbc:odbc:testdatenbank;UID=meyer;PWD=geheim

Data Source

- Ermöglicht den abstrakten Zugriff auf eine Datenquelle.
- Eingeführt ab JDBC 2.0, Im Optional Package `javax.sql`.
- Anstelle der Adressierung über einen Connection URL.

```
public Interface DataSource {  
    Connection getConnection()  
    Connection getConnection(String user, String passwd)  
    int      getLoginTimeout()  
    void      setLoginTimeout(int secs)  
    PrintWriter getLogWriter()  
    void      setLogWriter( PrintWriter out )  
}
```

Eine `DataSource` repräsentiert im Wesentlichen eine Verbindung zu einer bestimmten Datenbank. Daneben gehört zu einer `DataSource` ein `LogWriter`, das heisst ein `OutputStream` für Meldungen des Datenbanksystems. Ausserdem kann ein `Connection Timeout` definiert werden.

Die im folgenden beschriebenen `ConnectionPoolDataSource` und `XADataSource` werden dem Client gegenüber wenn möglich nicht sichtbar gemacht, sondern die dazugehörenden `Connections` werden in gewöhnliche `DataSource` Objekte verpackt und dem Client zur Verfügung gestellt. Siehe Demo-Beispiel `PoolManager.java`

Data Source

- Die implementierende Klasse des DataSource Interface hat verschiedenste (im Standard nicht vorgegebene) Methoden zur Adressierung der Datenquelle. Beispiel von Sybase:

```
SybDataSource ds = new SybDataSource(); // implements DataSource
ds.setUser("sws24");
ds.setPassword("swsworkshop");
ds.setDatabaseName("studenten_db");
ds.setServerName("swsun00.hta-be.bfh.ch");
ds.setPortNumber(1534);
ds.setDescription("");
Properties props = new Properties();
props.put("REPEAT_READ", "true");
// zusätzlich durch Middleware-Anwendungen ...
ds.setConnectionProperties(props);
Context ctx = new InitialContext();
ctx.bind("jdbc/myDataSource", ds);
```

Die im obigen Beispiel verwendeten Methoden von SybDataSource sind proprietär. Aus Sicht des DataSource Benutzers gilt die Interface-Definition aus `javax.sql.DataSource`.

Ein Datenbank- oder Middleware-Hersteller kann einer DataSource einen Connection-Pool hinterlegen. Die Connection wird beim Aufruf von `getConnection()` aus dem Pool ausgetragen, und beim Aufruf von `Connection.close()` wieder in den Pool zurückgegeben. Der Aufbau des Pools findet zur Startup-Zeit der Middleware statt, oder allenfalls durch eigene Dienst-Klassen im Rahmen der Applikation.

Data Source

- Das DataSource-Objekt kann von einer Applikation selber erzeugt oder via Kontext abgeholt und verwendet werden. Beispiel:

```
// Abholen aus Context, zum Beispiel für BMP in EJB-Klasse
Context cntx = new InitialContext();
DataSource ds = (DataSource) cntx.lookup("jdbc/myDataSource");
// Connection abholen
Connection con = ds.getConnection();
// Arbeiten mit Connection-Objekt wie gehabt z.B
Statement s = con.createStatement();
s.executeUpdate( ... );
con.commit();
```

Die Angabe, wie eine DataSource referenziert wird, hängt vom verwendeten Naming Service ab. Das kann beispielsweise ein LDAP Service sein, oder spezifischer für eine JAVA EJB Umgebung der ENC (Environment Naming Context).

DataSource ist ein Interface, das durch Klassen eines JDBC-Anbieters implementiert wird. DataSource-Objekte können im Applikationskontext zuhanden anderer Nutzniesser abgelegt werden.

Connection Pooling - Performance

- Der Aufbau einer Datenbank-Verbindung ist eine extrem zeitintensive Aufgabe. Einige Messbeispiele:
 - Durchführen einer einfachen Abfrage mit Verbindungsaufbau benötigt im LAN grob 100 msec, im WAN grob 400 msec
 - Durchführen einer einfachen Abfrage über eine bestehende Verbindung benötigt im LAN grob 10 msec, im WAN grob 60 msec
- Connection-Pooling ist eine wichtige Massnahme zur Performance Verbesserung.

Die getestete 'einfache' Abfrage ist:

```
SELECT name FROM person WHERE id = 1
```

Hardware

Server: Sun Sparc Station (1998), Sun Solaris 2.6, Sybase 11.5

Netzwerk LAN: 100 MBit Ethernet

Netzwerk WAN: 100 MBit Ethernet, Firewall, CableCom Modem (512/256)

Client: IBM A22m (2001), Java 1.4, jConnect5

ConnectionPoolDataSource

- JDBC beschreibt einige Interfaces für das Arbeiten mit Connection-Pools
 - ConnectionPoolDataSource
 - PooledConnection
 - ConnectionEventListener
 - ConnectionEvent
- Für die Applikation soll nicht ersichtlich sein, ob sie mit einer normalen oder gepoolten Connection arbeitet, d.h. sie bekommt eine gewöhnliche DataSource zur Verfügung gestellt.
- Der Pool-Manager (z.B. durch den Applikationsserver zu erstellen) soll in standardisierter Weise via JDBC auf die Datenbank zugreifen können.

Das Interface ConnectionPoolDataSource ist sehr ähnlich zu DataSource:

```
public Interface ConnectionPoolDataSource {  
    PooledConnection    getPooledConnection()  
    PooledConnection    getPooledConnection(String user, String passwd)  
    int                  getLoginTimeout()  
    void                  setLoginTimeout(int secs)  
    PrintWriter          getLogWriter()  
    void                  setLogWriter( PrintWriter out )  
}
```

Eine PooledConnection repräsentiert eine physische Verbindung zur Datenbank:

```
public Interface PooledConnection {  
    void                addConnectionEventListener(ConnectionEventListener l)  
    void                removeConnectionEventListener(ConnectionEventListener l)  
    Connection          getConnection(String user, String passwd)  
    Connection          getConnection()  
    void                close()  
}
```

Ein Pool-Manager (innerhalb eines Applikationsservers oder einer entsprechenden, selber gebauten Komponente) fordert über ConnectionPoolDataSource eine physische Datenbank-Verbindungen an (PooledConnection). ConnectionPoolDataSource ist von einem JDBC-Hersteller implementiert. In die PooledConnection trägt sich der Pool-Manager selber als Event Listener ein. verwaltet. Fordert eine Applikation eine Verbindung an, liefert der Pool-Manager eine gewöhnliche Connection via PooledConnection.getConnection() aus.

Die ausgelieferte Connection unterscheidet sich interface-mässig in nichts von einer gewöhnlichen, direkten Verbindung zur Datenbank. In Wahrheit ist sie jedoch nur ein "Handle" auf die PooledConnection, welche die wirkliche Verbindung zur Datenbank beinhaltet. Führt die Applikation ein close() auf ihr Connection-Objekt durch, wird die physische Verbindung nicht geschlossen, sondern nur der PoolManager benachrichtigt, dass die Verbindung wieder für andere Applikationen, resp. andere Applikationsteile zur Verfügung steht.

Siehe Demo-Beispiel PoolManager.java

Siehe JDBC 3.0 Kapitel 11

XADataSource

- Eine XADataSource repräsentiert eine von mehreren Datenquellen, die an einer *verteilten* Transaktion (XA) teilnehmen.
- Für die Applikation soll transparent sein, dass ihre Datenzugriffe im Rahmen einer verteilten Transaktion stattfinden. Sie arbeitet funktional mit einer gewöhnlichen Connection.
- Die Transaktionsabwicklung findet durch einen Transaktions-Manager statt, bei dem die beteiligten Datenquellen registriert sind.

Das Interface XADataSource ist sehr ähnlich zum Interface DataSource:

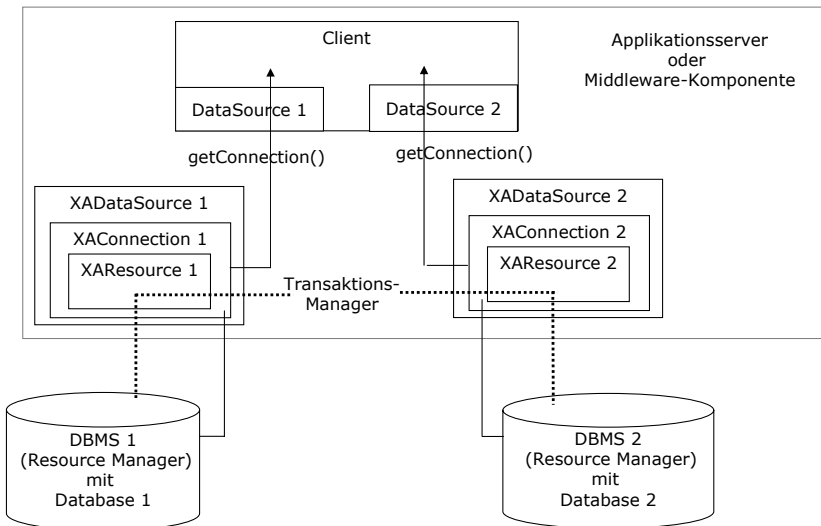
```
public Interface javax.sql.XADataSource {  
    XAConnection    getXAConnection()  
    XAConnection    getXAConnection(String user, String passwd)  
    int              getLoginTimeout()  
    void             setLoginTimeout(int secs)  
    PrintWriter     getLogWriter()  
    void             setLogWriter( PrintWriter out )  
}
```

Eine XAConnection repräsentiert eine physische Verbindung zur Datenbank:

```
public Interface javax.sql.XAConnection extends PooledConnection {  
    XAResource    getXAResource()  
}
```

Ein Pool-Manager (innerhalb eines Applikationsservers oder einer entsprechenden, selber gebauten Komponente implementiert) fordert über die XADataSource eine physische Datenbank-Verbindungen an (XAConnection). Das XADataSource-Interface ist von einem JDBC-Hersteller implementiert. In die XAConnection trägt sich der Pool-Manager selber als Event Listener ein. Fordert eine Applikation eine Verbindung an, liefert der Pool-Manager eine gewöhnliche Connection via XAConnection.getConnection() aus. Die XAConnection wird zusätzlich in einem globalen Transaktionsobjekt (siehe javax.transaction.Transaction und javax.transaction.TransactionManager) registriert. Über dieses Transaktionsobjekt kann dann zu den geforderten Zeitpunkten (zum Beispiel durch einen Applikationsserver gemäss Transaktions-Angaben für die Methoden eines Session-Beans) die verteilte Transaktion durchgeführt werden.

Architekturschema für XA



Implementationen für XADataSource, XAConnection und XAResource werden von JDBC-Hersteller bereitgestellt. Der "Transaktions-Manager" ist eine Implementation von JTA (Java Transaction API), typischerweise von einem Middleware-Hersteller. Dieser implementiert auch das Interface DataSource für den Client.

Das Interface XAResource ist über `getXAResource()` in XAConnection zugänglich. Während sich der Client in vielen Fällen nur für die Connection interessiert, ist der Transaktions-Manager an sich nur an den Methoden von XAResource interessiert, um die Transaktion durchführen zu können.

```
Interface XAResource {
    void start(Xid xid, int flags)
    void end(Xid xid, int flags)
    int prepare(Xid xid)
    void commit(Xid xid)
    void rollback(Xid xid)
    Xid[] recover(int flag)
}
```

Der Transaktionsmanager erzeugt ein Xid und startet auf allen XAResources die verteilte Transaktion mit `start()`. Nun kann von Clients mit den XAConnections gearbeitet werden.

Der Transaktionsmanager startet das Two Phase Commit, wenn die beteiligten Connections von den Clients logisch geschlossen wurden. Als erstes schliesst er die Datenbearbeitung auf jeder XAResource mit `end()` ab. Anschliessend schickt allen XAResources ein `prepare()`. Falls dabei keine Fehler aufgetaucht sind, wird allen XAResources ein `commit()` geschickt und die verteilte Transaktion ist beendet. Ist beim `prepare()` bei irgendeiner XAResource ein Fehler aufgetreten, verschickt der Transaktionsmanager ein `rollback()` an alle. Kann das `commit()`, was bei einem Absturz der Fall wäre, nicht bei allen XAResources ordnungsgemäss durchgeführt werden, muss der Transaktionsmanager periodisch versuchen, das `commit` doch noch durchzubringen. Dabei hat er allenfalls auf den Restart des entsprechenden DBMS zu warten, eine neue XAResource aufzubauen und mit `recover()` die pendenten Transaktionen zu ermitteln.

Anmerkung: Im XA Standard sind keine expliziten Befehle für die Beeinflussung von Transaktionen, zum Beispiel Isolation Level oder Savepoints vorgesehen. Entsprechende Möglichkeiten sind allenfalls bei einem konkreten DBMS einzustellen.

Siehe JDBC 3.0 Kapitel 12

Two-Phase-Commit (2PC)

- Garantiert die ACID-Eigenschaften für eine Transaktion, die sich über mehrere (unabhängige) Standorte erstreckt.
- Eine Transaktions-Manager führt die Gesamt-Transaktion durch.
 - Phase 1: Sicherstellung der modifizierten Daten in allen Standorten
 - Phase 2: Bestätigen und Freigeben der neuen Daten in allen Standorten

Während Phase 1 fordert der Koordinator alle Standorte auf, ihre Änderungen sicherzustellen und gegen Abstürze zu sichern (prepare to commit). Dies geschieht durch Protokollierung im Log-File wie bei gewöhnlichen Transaktionen. Sobald von allen Standorten eine Bestätigung eingetroffen ist, wird Phase 2 eingeleitet.

Während Phase 2 sendet der Koordinator allen Standort eine Meldung, dass sie die Änderungen nun freigeben dürfen (definitives commit).

Wie verhält sich das Protokoll im Fehlerfall, d.h. bei Absturz oder Verbindungsabbruch einzelner Teilnehmer.

Definitionen:

- Ein Teilnehmer befindet sich in Phase 1 bis sein prepare abgeschlossen ist.
- Der Koordinator befindet sich in Phase 1 bis die prepare-Meldungen aller Teilnehmer eingetroffen sind.
- Ein Teilnehmer befindet sich in Phase 2 bis das commit abgeschlossen ist.
- Der Koordinator befindet sich in Phase 2 bis die commit Meldungen aller Teilnehmer eingetroffen sind.

Fehlerfälle:

- Teilnehmer Phase 1: Koordinator schickt allen Teilnehmern ein rollback-Befehl. Der verursachende Teilnehmer verliert seine Teiltransaktion ohnehin.
- Teilnehmer Phase 2: Der Teilnehmer wartet nach seinem Restart auf commit- oder rollback-Befehl vom Koordinator. Ein commit wird abgesetzt, falls sich der Koordinator ebenfalls in Phase 2 befindet.
Die übrigen Teilnehmer sind durch den Fehler nicht betroffen, da der Koordinator ihnen ein commit schicken kann, sobald er selber in Phase 2 ist.
- Koordinator Phase 1 oder 2: Teilnehmer in Phase 1 lösen für sich ein rollback aus. Teilnehmer in Phase 2 müssen auf Restart des Koordinators warten, da sie nicht selber entscheiden können, ob ein rollback oder ein commit durchzuführen ist. Ein Rollback wäre durchzuführen wenn der Koordinator in Phase 1 war beim Absturz, ein commit könnte gemacht werden, wenn der Koordinator in Phase 2 war beim Absturz. Der Koordinator verschickt nach dem Restart die entsprechende Meldung an seine Teilnehmer.
- Administratoren von Teilnehmer-Datenbanksystemen können manuell auf eigene Verantwortung eine Transaktion in Phase 2 committen oder rollbacken. Der Transaktionsstatus wird aufbewahrt zuhanden des Koordinators, sobald letzterer wieder verfügbar ist. Der Transaktionsstatus wird gelöscht, wenn der Administrator oder der Koordinator einen forget-Befehl auslösen.

Zusammenfassung

- Die *DataSource-Philosophie* ermöglicht ein gleichbleibendes Ansprechen unterschiedlicher Datenquellen durch die Applikation.
- JDBC definiert sowohl die applikationsseitigen Interfaces (DataSource, Connection), wie auch die datenbankseitigen Interfaces (DataSource, ConnectionPoolDataSource, XADataSource, Connection, PooledConnection, XAConnection, XAResource).
- Die JDBC-Hersteller liefern Implementation für die datenbankseitigen Interfaces.
- Die Realisierung der Middleware-Logik (Pool-Manager, Transaktions-Manager ist durch JDBC nicht abgedeckt.

Statements

```
...
Statement stat = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE );
ResultSet rs = stat.executeQuery("select * from person");
try {
    while ( rs.next() ) {
        String name = rs.getString( "name" );
        System.out.println( name );
    }
    rs.beforeFirst();
    while ( rs.next() ) {
        String name = rs.getString( "name" );
        rs.updateString( "email", name+"@sws.bfh.ch" );
        rs.updateRow();
        if ( name.length() == 0 ) rs.deleteRow();
    }
    catch( SQLException e ) { System.out.println ( e ); }
}
...
```

Ein Statement ist der Ausgangspunkt einer SQL-Abfrage. Dem Statement-Objekt kann jederzeit ein SQL-Befehl zur Ausführung übergeben werden. Dabei ist für das Statement-Objekt definierbar, wie das zurückgelieferte ResultSet durchlaufen werden kann, und ob die Daten darin modifizierbar sind. Im einfachsten Fall kann ein ResultSet nur vorwärts durchlaufen (TYPE_FORWARD_ONLY) und nicht geändert (CONCUR_READ_ONLY) werden. Im obigen Fall sind sämtliche Durchlaufarten (next(), previous(), first(), last(), absolute(int), relative(int) usw.) möglich (TYPE_SCROLL_INSENSITIVE) und im ResultSet können Daten geändert, gelöscht und eingefügt werden (CONCUR_UPDATABLE). Die Änderungen werden jeweils unmittelbar in der Datenbank nachgetragen. Zu Beachten ist, dass der Modifizierbarkeit durch das Datenbanksystem wesentliche Grenzen gesetzt sind. Join-Abfragen, Abfragen mit Funktionsaufrufen oder Operatoren in der select-Klausel, Abfragen mit group-by-Klauseln dürfen nicht modifiziert werden. Es sind dann neue Statement-Objekte zu verwenden, welche selber die notwendigen SQL-Befehl (insert, delete, update) explizit absetzen. Dieses Verfahren ist in vielen Fällen ohnehin zu bevorzugen, da die genauen Rahmenbedingungen für das Ändern von Daten über Funktionen im ResultSet, und die unmittelbare Sichtbarkeit dieser Änderungen in der Datenbank, vom Datenbank-Produkt abhängig sind.

Weiterhin ist zu beachten, dass gewisse Funktionen eines ResultSet live von einer Datenbank-Connection abhängig sind. Das Zurückgeben von ResultSets an aufrufende Funktionen, im Sinne eines Return-Parameters ist daher mit Vorsicht zu verwenden. Sicherer ist das Umkopieren von Abfrageresultaten in Listen, Vektoren usw.

Positionierte Abfragen

- Ein ResultSet stellt verschiedene Funktionen für die absolute und relative Positionierung auf bestimmte Datensätze zur Verfügung.
- Beispiel: Mechanismus zum Blättern auf Webseiten. Ausgabe der Datensätze zwischen Positionen firstpos und lastpos.

```
String query = " SELECT name, vorname ";
query += " FROM Student ORDER BY name, vorname";

Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_READ_ONLY );

ResultSet rs = stmt.executeQuery( query );
rs.absolute( firstpos );
for( int i = firstpos; i <= lastpos; i++ ) {
    System.out.println( rs.getString(1) + rs.getString(2) );
    rs.relative( 1 );
}
```

ResultSet.TYPE_FORWARD_ONLY heisst, es kann nur vorwärts durch ein ResultSet iteriert werden. Die einzige Positionierungsfunktion ist next().

ResultSet.TYPE_SCROLL_INSENSITIVE heisst, es kann in allen Richtungen durch ein ResultSet iteriert und beliebig positioniert werden.

ResultSet.TYPE_SCROLL_SENSITIVE heisst, beim Positionieren auf einen bestimmten Datensatz, wird automatisch von der Datenbank der aktuellste Zustand dieses Datensatzes geholt.

ResultSet.CONCUR_READ_ONLY heisst, das ResultSet ist Read-Only.

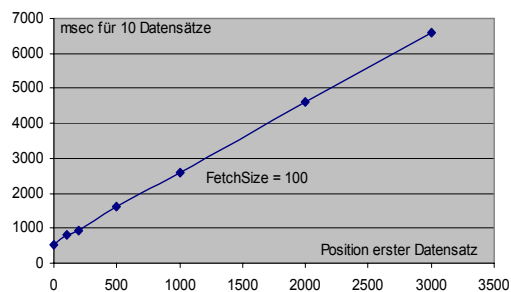
ResultSet.CONCUR_UPDATABLE heisst, via ResultSet kann direkt die Datenbank modifiziert werden.

Die Positionierungsfunktionen sind

```
ResultSet.first()
ResultSet.last()
ResultSet.next()
ResultSet.previous()
ResultSet.beforeFirst()
ResultSet.afterLast()
ResultSet.absolute( int )
ResultSet.relative( int )
```

Positionierte Abfragen - Performance

- Ohne besondere Angaben werden alle Datensätze in die Applikation geladen. Die Positionierung findet in der Applikation statt.
- Mit `Statement.setFetchSize()` kann die Anzahl übertragener Datensätze definiert werden.



Sollen nur die vorderen Datensätze einer SQL-Abfrage verarbeitet werden, ist eine `FetchSize` vorteilhaft, welche gerade der Anzahl zu verarbeitender Datensätze entspricht. Sollen Datensätze in der Mitte oder am Schluss einer SQL-Abfrage verarbeitet werden, ist eine möglichst grosse `FetchSize` sinnvoll.

Hardware

Server: Sun Sparc Station (1998), Sun Solaris 2.6, Sybase 11.5

Netzwerk: 100 MBit Ethernet

Client: IBM A22m (2001), Java 1.4, jConnect5

Eine ideale und gut optimierte Lösung für positionierte Abfragen wären SQL Cursor.

Diese erlauben *datenbankseitig* das Anspringen bestimmter Datensätze. SQL-99 stellt für Cursor unter anderem folgende Befehle zur Verfügung:

```
declare cursorname [options] cursor for  
select... from ...where ... [ order by ... ]
```

```
fetch [from] cursorname [ next | prior | first | last | absolute n |  
      relative n ]
```

Der declare-Befehl definiert eine Menge von Datensätzen, die gelesen oder bearbeitet werden sollen. Der fetch-Befehl erlaubt das Anspringen einer bestimmten Position.

Leider ist der Umgang mit Cursors in JDBC schlecht ausgebaut resp. schlecht definiert.

Einzig durch Aufruf der Methode `Statement.setCursorName()` kann explizit das Arbeiten mit einem Cursor verlangt werden. Gewisse Treiber erstellen von sich aus einen Cursor wenn die Methode `Statement.setFetchSize()` aufgerufen wird (Beispielsweise auch Sybase). Die Implementation der Driver-Klassen ist aber häufig nicht so, dass beispielsweise beim Befehl `ResultSet.absolute( n )` der Cursor-Befehl `fetch absolute n` verwendet wird. Es werden stattdessen alle Datensätze bis zur gewünschten Position abgeholt und im `ResultSet` abgelegt. Das heisst, der positionierte Zugriff wird nur simuliert. Sybase empfiehlt beispielsweise, die Positionierungs-Methoden von `ResultSet` nur bei vernünftig kleinen SQL-Abfragen zu verwenden.

Auch eine manuelle Kontrolle, zum Beispiel das explizite Absetzen eines `fetch`-Befehles, ist nicht möglich. Andere APIs wie ODBC, embedded SQL für C, Stored Procedure-Sprachen lassen ein sehr gezieltes Arbeiten mit Cursors zu.

Cursor hätten eine Reihe von Vorteilen bei positionierten Abfragen:

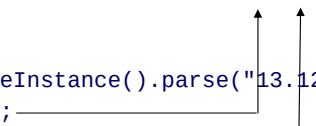
- hohe Optimierung beim Anspringen bestimmter Positionen
- Gutes Locking Verhalten
- Snapshot-Darstellung oder Live-Verknüpfung gegenüber den zugrundeliegenden Datensätzen
- Gezielter Update- oder delete-Befehl auf den aktuellen Datensatz ohne Verwendung eines Primärschlüssels mit
 `update tabelle ... where current of cursorname`
 `delete tabelle where current of cursorname`

Prepared Statements

- Beispiel für das Einfügen von Daten

```
...
PreparedStatement stat = con.prepareStatement( "insert into
                                             person ( name, gebdatum ) values ( ?, ? )" );

try {
    Date date = DateFormat.getDateInstance().parse("13.12.1970");
    stat.setString( 1, "schmid" );
    stat.setDate( 2, new java.sql.Date( date.getTime() ) );
    int count = stat.executeUpdate();
    System.out.println( "Eingefügte Records: " + count );
}
catch( SQLException e ) { System.out.println ( e ); }
...
```



Prepared Statements stellen grundsätzlich nicht mehr Funktionalität als gewöhnliche Statements zur Verfügung, besitzen jedoch eine Reihe von Vorteilen in der Handhabung:

- Auch "schwierige" Parameter, wie Datumsobjekte oder binäre Daten können leicht eingesetzt werden.
- Automatische Ersetzung von Anführungszeichen in String-Parametern
- Sie können vom Datenbanksystem vorkompiliert werden und sind dadurch schneller, wenn Sie mehrmals nacheinander aufgerufen werden

Gebrauch von Prepared Statements:

- Anstelle des Fragezeichens wird mit der jeweiligen setXXX() Methode ein Parameterwert eingefügt.
- Ein Fragezeichen kann überall dort stehen, wo in einem SQL-Befehl ein Datenwert stehen kann.
- Das Fragezeichen ist nicht erlaubt zur Parametrisierung von Attributnamen, Tabellennamen etc.

Beispiele für PreparedStatement:

- insert into tabelle values (?)
- update tabelle set feld = ?
- delete from tabelle where feld = ?
- select * from tabelle where feld = ?

Prepared Statements ff.

- Beispiel für das Lesen von Daten

```
public void myQuery () throws Exception
{
    try {
        int id = 3;
        PreparedStatement stat = con.prepareStatement(
            "select name, gebdatum from person where id = ? ");
        stat.setInt( 1, id ) _____↑
        ResultSet rs = stat.executeQuery();
        while ( rs.next() ) {
            System.out.println( rs.getString( 1 ) );
            Date gebdatum = rs.getDate( 2 );
            if ( ! rs.wasNull() ) {
                System.out.println( gebdatum.toString() );
            }
        }
    }
    catch( SQLException e ) {
        System.out.println ( e );
    }
}
```

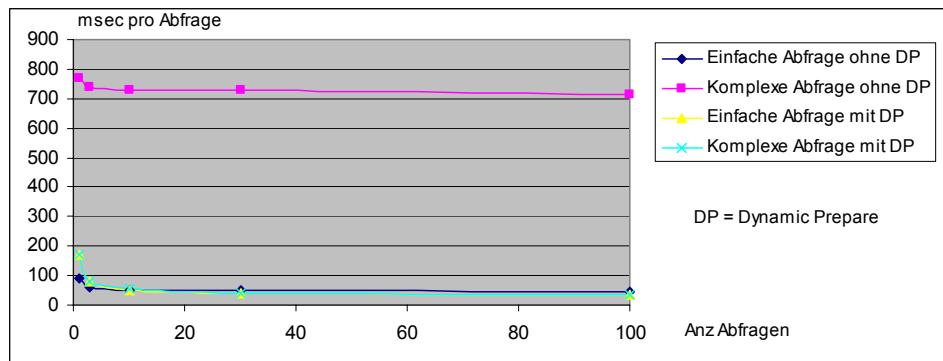
Ein Fragezeichen kann überall dort stehen, wo in einem SQL-Befehl ein Datenwert stehen kann.

Das Abfrageresultat in einem Objekt der Klasse ResultSet wird häufig mit der Funktion next() durchlaufen.

Der erste Aufruf von next() positioniert das ResultSet auf den ersten Eintrag.

Der Aufruf von wasNull() testet, ob das letzte abgefragte Feld einen Null-Wert enthalten hat. Die Methode wasNull() bezieht sich immer auf die aktuelle Zeile im ResultSet und das letzte abgefragte Feld. Beachte, dass bei Objekttypen im Fall von Null-Werten in der Datenbank auch eine null-Referenz erzeugt wird. Bei Basistypen (int, float, char etc.) ist jedoch die Funktion wasNull() für die Vermeidung von Missverständnissen und die Unterscheidung von 0 und 'Null' zwingend notwendig.

Prepared Statements - Performance



- Prepared Statements sparen massiv Zeit bei komplexen SQL-Abfragen und mehrmaliger Durchführung!

Obige Zeitmessungen beinhalten den Aufruf von `prepareStatement()`. Die Prepared Statements werden der Datenbank zur Vorkompilierung übergeben. Bei Sybase heisst das, aus der Abfrage wird temporär eine Stored Procedure erzeugt. Damit die Vorkompilierung durchgeführt wird, muss beim Verbindungsaufbau eine entsprechende Property gesetzt werden (produktspezifischer Name):

```
props.put("DYNAMIC_PREPARE", args[3] );
Connection con = DriverManager.getConnection( args[0], props );
```

Die getestete 'einfache' Abfrage ist:

```
SELECT name FROM Student WHERE idStudent = ?
```

Die getestete 'komplexe' Abfrage ist:

```
SELECT count(*), s.name, k.titel
FROM Rating r, Fragebogen f, Student s, Kriterium k
WHERE r.idFragebogen = f.idFragebogen AND f.idStudent = s.idStudent
AND r.idKriterium = k.idKriterium AND s.idStudent = ?
GROUP BY s.name, k.titel, k.idKriterium
ORDER BY k.titel
```

Tabellengrößen

Student 4000, Rating 200000, Fragebogen 20000, Kriterium 1000.

Hardware

Server: Sun Sparc Station (1998), Sun Solaris 2.6, Sybase 11.5

Netzwerk: 100 MBit Ethernet, Firewall, CableCom Modem (512/256)

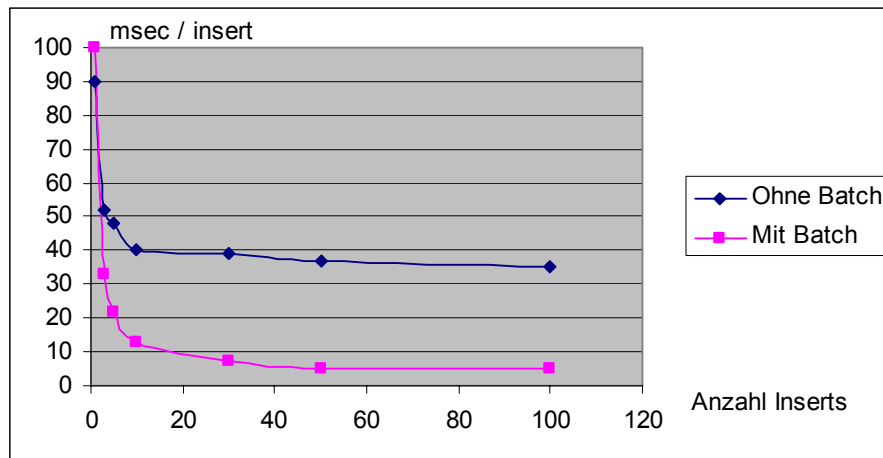
Client: IBM A22m (2001), Java 1.4, jConnect5

Batch Updates

- Batch Updates sind eine Neuerung ab JDBC 2.0
- Es können mehrere SQL insert-, update- oder delete-Befehle in einem Paket an die Datenbank geschickt werden.
- Beispiel:

```
...
String query = "INSERT INTO person ( name, email, gebdatum )
               VALUES ( ?, ?, getdate() )";
PreparedStatement pstmt = con.prepareStatement( query );
while ( ... ) {
    pstmt.setString( 1, "name " + i );
    pstmt.setString( 2, "email " + i );
    pstmt.addBatch();
}
pstmt.executeBatch();
...
```

Batch Updates - Performance



- Batch Updates sparen massiv Zeit für die Durchführung mehrerer SQL-Befehle in einem Schritt!

Batch Updates optimieren den Netzwerk-Verkehr.

Prepared Statements optimieren die Durchführungszeit in der Datenbank.

Die beiden Konzepte können daher mit Gewinn kombiniert werden.

Hardware

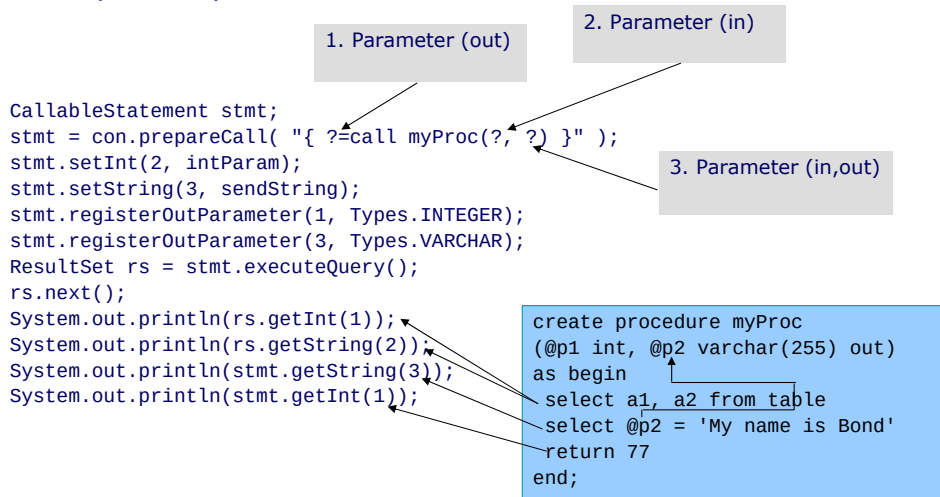
Server: Sun Sparc Station (1998), Sun Solaris 2.6, Sybase 11.5

Netzwerk: 100 MBit Ethernet, Firewall, CableCom Modem (512/256)

Client: IBM A22m (2001), Java 1.4, jConnect5

Callable Statements

- Für den Aufruf von stored procedures
- Beispiel in Sybase



Transaktionskontrolle

- JDBC enthält Methoden für
 - das Setzen des Commit-Modus (Autocommit)
 - das Setzen des Isolation Levels
 - das Setzen von Save Points
 - commit und rollback von Transaktionen
 - die Kontrolle von verteilten Transaktionen (XAResources)

Im Rahmen von EJB-Applikationen ist insbesondere der Isolation Level entweder auf Container-Ebene (Deployment Deskriptor) oder auf Bean-Ebene (mit expliziter JDBC-Programmierung) zu definieren.

Commit-Modus

- Eine Transaktion wird immer durch den Datenbank-*Client* bestätigt oder verworfen (commit oder rollback)
- Die Methode `Connection.setAutoCommit()` definiert, ob der Driver von sich aus nach jedem SQL-Befehl ein commit absetzt.
- Ein SQL-Befehl gilt aus Sicht JDBC als beendet
 - unmittelbar nach der Ausführung eines `executeUpdate()`-Befehles
 - wenn das `ResultSet` eines `executeQuery()`-Befehles geschlossen wird

Transaktionen bestehen in der Regel aus mehreren SQL-Befehlen. Autocommit sollte daher in der Regel auf `false` gesetzt werden, damit erst nach dem letzten zu einer Transaktion zugehörigen Befehl die Änderungen an der Datenbank bestätigt oder allenfalls verworfen werden können.

Ein `ResultSet` wird mit der Methode `close()` explizit geschlossen oder wenn über das zugehörige `Statement`-Objekt ein neuer SQL-Befehl abgesetzt wird, das `Statement`-Objekt oder die `Connection` geschlossen wird.

Wenn eine `execute()`-Methode mehrere `ResultSets` zurückliefert, gilt der SQL-Befehl aus Sicht JDBC abgeschlossen, wenn alle zugehörigen `ResultSets` geschlossen sind.

Einige Datenbanksysteme bieten ein serverseitiges Autocommit an. Faktisch ist damit ein SQL-Befehl abgeschlossen, sobald alle Datensätze, resp. die Ausführungsbestätigung an den Client geschickt wurden. Führt ein Datenbanksystem im Rahmen einer `Stored Procedure` mehrere SQL-Befehle aus, wird bei serverseitigem Autocommit jeder einzelne dieser SQL-Befehl committed.

Der Driver weiss, ob bei `setAutoCommit( true )` ein clientseitiges oder serverseitiges Autocommit durchzuführen ist. Serverseitiges Autocommit ist identisch mit dem ANSI Transaktionsmodus `CHAINED`.

Isolation Level

- Im Grundsatz sollte eine laufende Transaktion logisch unbeeinflusst von jeder anderen laufenden Transaktion sein -> Transaktion sollten sich serialisierbar verhalten.
- Die Serialisierbarkeit hängt insbesondere davon ab, wie weit eine Transaktion Änderungen von anderen *laufenden oder eingeschobenen* Transaktionen sieht oder beeinflusst (Isolationsgrad).
- In SQL99 und JDBC 3.0 sind 4 Isolation Levels definiert

0	READ UNCOMMITTED
1	READ COMMITTED
2	REPEATABLE READ
3	SERIALIZABLE

```
Connection.setTransactionIsolation( int level )
```

```
levels:  Connection.TRANSACTION_READ_UNCOMMITTED
         Connection. TRANSACTION_READ_COMMITTED
         Connection. TRANSACTION_REPEATABLE_READ
         Connection. TRANSACTION_SERIALIZABLE
```

Die Realisierung eines Isolation Levels geschieht in den meisten Fällen über Sperr-Mechanismen:

Schreibsperren (exklusive Sperren) werden von einer Transaktion auf Datensätze gesetzt, die von ihr geändert, neu eingefügt oder gelöscht werden. Schreibsperren dauern immer vom Moment ihres Setzens bis zum Ende der Transaktion an.

Zum Lesen von Daten werden meist Lesesperren gesetzt. Dabei gibt es verschiedene Variationen: Eine Lesesperre wird gar nicht gesetzt (READ UNCOMMITTED), eine Lesesperre wird nur für den kurzen Moment des Lesevorganges gesetzt (READ COMMITTED), eine Lesesperre wird ab dem Moment ihres Setzens bis zum Ende der Transaktion gesetzt (REPEATABLE READ), eine Lesesperre wird mit Range Locks ergänzt, welche das Einfügen von neuen Datensätzen verhindern (SERIALIZABLE). Im Weiteren gibt es oft auch die Möglichkeit, in ganz kritischen Fällen für das Lesen von Daten bereits exklusive Sperren anzufordern. Noch restriktiver kann auch eine ganze Tabelle *zum Vornherein* mit einem lock table-Befehl gesperrt werden.

Neben Sperr-Mechanismen existieren für die Realisierung eines Isolation Levels je nach Datenbank-Produkt weitere Konzepte: Das Zeitstempel-Verfahren ermöglicht es festzustellen, ob ein Datensatz zwischen dem Lesen und dem Zurückschreiben in die Datenbank von anderen Transaktionen geändert wurde. Eine Datensatz-Versionierung (Oracle) ermöglicht es, bis zum Isolation Level REPEATABLE READ ganz ohne Sperren auszukommen.

READ UNCOMMITTED

- Eine Transaktion kann jederzeit noch laufende und unbestätigte Änderungen anderer Transaktionen sehen.
- Vorteil: Die Transaktion wird nie durch andere blockiert
- Nachteil: Es werden ev. Datenwerte gelesen, die durch die schreibende Transaktion wieder verworfen werden. Eingefügte, aber noch nicht bestätigte Datensätze werden ebenfalls bereits gelesen.
- Beispiel: Anzeige- und Visualisierungsprozesse, die nicht im Detail auf konsistente Daten angewiesen sind.

READ UNCOMMITTED heisst auch Dirty Read.

Fügt eine Transaktion Daten ein, die später aufgrund einer Verletzung von Integritätsbedingungen oder durch den Benutzer wieder gelöscht werden, sieht eine Transaktion im Modus READ UNCOMMITTED diese Daten bereits. Dasselbe gilt für Änderungen an bestehenden Daten.

READ COMMITTED

- Eine Transaktion sieht jeweils den letzten *bestätigten* Zustand eines Datensatzes.
- Vorteil: Es werden nur Daten gelesen, die mindestens einmal korrekt waren.
- Nachteil: Je nach Implementation kann es bereits zu Wartesituationen kommen.
- Beispiel: Lies den aktuellen Saldo aller Konto-Datensätze. (Snapshot-Zustand).
- Achtung: Aufgrund der gelesenen Daten dürfen aber keine Updates in der Datenbank durchgeführt werden.

In vielen Datenbanksystemen ist der Modus READ COMMITTED so implementiert, dass für das Lesen eines Datensatzes eine kurze Lesesperre gesetzt wird. Diese Lesesperre kann jedoch mit einer bestehenden Schreibsperre einer anderen Transaktion kollidieren, die den Datensatz geändert hat, einen Datensatz neu eingefügt oder (je nach Implementation) einen Datensatz gelöscht hat.

Bei Oracle wird von geänderten Datensätzen jeweils eine Kopie des zuletzt gültigen Zustandes aufbewahrt. Diese Kopie steht Transaktionen im Modus READ COMMITTED zur Verfügung. In diesem Fall blockiert die lesende Transaktion nie, und der Modus READ UNCOMMITTED wird überflüssig.

Folgendes Szenario ergibt ein falsches Ergebnis, wenn mit READ COMMITTED gearbeitet wird: Transaktion T1 liest den Kontostand von Konto 1. Transaktion T2 tut dasselbe. T2 ändert den Kontostand um den Betrag +X und committed. T1 ändert den Kontostand ebenfalls um +X. Die Änderung von T2 geht damit verloren!

REPEATABLE READ

- Mit dem Lesen eines Datensatzes wird dessen Zustand in der Datenbank eingefroren. Jedes nachfolgende Lesen ergibt wieder denselben Zustand. Dieselbe Abfrage liefert dasselbe Resultat.
- Vorteil: Der Zustand (Wert) der in die Applikation gelesen wurde, ist mit dem Zustand in der Datenbank identisch.
- Nachteil: Es ist immer noch möglich, Datensätze neu einzufügen, welche der Abfragebedingung genügen (Phantome). Diese können unter Umständen Berechnungen in der Applikation verfälschen.
- Beispiel: Lies den aktuellen Kontostand und ändere ihn um den Betrag X.

Die Implementation geschieht in vielen Fällen mit Lesesperren, welche auf den gelesenen Datensätzen (denjenigen welche die where-Klausel erfüllen) für die Dauer der Transaktion gesetzt werden. Für das Lesen und Testen, ob ein Datensatz überhaupt der where-Bedingung genügt, wird eine kurze Lesesperre gesetzt.

Das unter READ COMMITTED beschriebene, fehlerhafte Szenario kann mit dem Modus REPEATABLE READ nicht mehr ablaufen. Wenn eine Transaktion T1 einen Datensatz gelesen hat, kann keine andere diesen Datensatz ändern. T1 kann also beispielsweise den Kontostand lesen, um den Betrag X erhöhen und zurückschreiben. Für das Zurückschreiben muss sie allerdings die Lesesperre, die sie bereits besitzt, in eine Schreibsperre umwandeln. Dies gelingt nur, wenn keine andere Transaktion eine Lesesperre besitzt. Besitzt eine andere Transaktion eine Lesesperre und möchte diese eventuell sogar auch in eine Schreibsperre umwandeln, entsteht eine Deadlock-Situation. Dies ist unangenehm, wird aber immerhin bemerkt, weil das Datenbanksystem dem Deadlock-Verursacher einen Rollback aufzwingt und eine Fehlermeldung schickt. Die betroffene Transaktion kann dann nochmals von der Applikation gestartet werden. Möchte man Deadlocks grundsätzlich verhindern, bleibt nur das Arbeiten mit expliziten Tabellensperren.

Folgendes Szenario ergibt ein falsches Ergebnis, wenn mit REPEATABLE READ gearbeitet wird: In einem ersten Durchgang liest Transaktion T1 die Anzahl Konto-Datensätze in einer Konto-Tabelle. Anschliessend fügt eine Transaktion T2 einen neuen Konto-Datensatz hinzu und committed. Danach liest Transaktion T1 in einem zweiten Durchgang wieder alle Datensätze und berechnet die Summe der Kontostände. Die Berechnung des mittleren Kontostandes durch Transaktion T1 liefert ein falsches Ergebnis!

SERIALIZABLE

- Eine Transaktion sieht jeweils den letzten bestätigten Zustand eines Datensatzes. Zusätzlich können keine Phantome mehr auftreten.
- Vorteil: Eine Transaktion arbeitet in jedem Fall korrekt.
- Nachteil: Starke Behinderung anderer Transaktionen.
- Beispiel 1: Lies die Summe aller Kontostände, Lies die Anzahl Kontos. Berechne durchschnittlichen Kontostand.
- Beispiel 2: Teste, ob eine Zimmerreservation zwischen Datum A und B möglich ist. Wenn ja, Füge die Reservation ein.

Die Implementation geschieht häufig durch Setzen von Lesesperren auf gelesenen Daten und zusätzlichen Einfügesperren: Für jeden gelesenen Datensatz wird eine Einfügesperre gesetzt, welche verhindert, dass vor (und nach) dem gelesenen Datensatz neue Datensätze eingefügt werden. Die Einfügesperren heißen auch Range Locks.

Beispiel: Die Transaktion mit der Abfrage `select * from Account where balance between 1000.00 and 2000.00` soll im Modus `SERIALIZABLE` ablaufen. Auf der Tabelle 'Account' besteht ein Index über dem Attribut 'balance'. Das Datenbanksystem sperrt die gelesenen Datensätze mit einer Lesesperre und setzt zusätzlich im Index für jeden gelesenen Wert sowie den nächst höheren Wert über 2000 von 'balance' einen Range Lock. Der Range Lock verhindert, dass im Index vor einem Eintrag mit einem Range Lock ein neuer Wert eingefügt wird.

Range Locks haben den immensen Vorteil, dass nicht ganze Tabellen, sondern nur bestimmte Bereiche gesperrt werden. Ältere Implementation arbeiten für die Realisierung von `SERIALIZABLE` mit einer Sperre (Lese- oder Schreib-) auf der Tabelle *als Ganzes*. Dies führt dazu, dass die beteiligten Transaktionen faktisch, nicht nur virtuell, nacheinander ablaufen.

Siehe auch Beispiele unter `X:\...\Demos\Isolation-Levels\`

Savepoints

- Statt im Fehlerfall eine ganze Transaktion zu verwerfen, kann es eventuell sinnvoll sein, nur auf bestimmte Sicherungspunkte zurückzufahren. SQL kennt hierzu die Befehle

`savepoint name`

`rollback to savepoint name`

- Datenbanken kennen das Savepoint-Konzept bereits länger. Neu wurde auch das JDBC-API um entsprechende Funktionen in der Connection-Klasse ergänzt:

`Savepoint setSavepoint( String name )`

`rollback( Savepoint savepoint )`

Mit Hilfe von Savepoints können relativ elegant Undo-Funktionen gebaut werden. Pro möglichem Undo-Schritt wird ein Save-Point gesetzt. Voraussetzung ist, dass während der Datenbearbeitung in einem GUI die entsprechende Transaktion offen bleiben kann. Eine Redo-Funktion ist einfach zu realisieren, indem alle SQL-Befehle, die zu einem Redo-Schritt gehören, in einer entsprechenden Liste aufbewahrt und beim Redo-Aufruf abgespielt werden.

Fehlerbehandlung

- Da SQL-Befehle ad hoc ausgeführt und interpretiert werden, bestehen grundsätzlich sehr viele Fehlermöglichkeiten:
 - Inkorrekte Syntax eines SQL-Befehls
 - Konvertierungsfehler (z.B. Data Truncation)
 - Kein Zugriffsrecht auf eine der beteiligten Tabellen
 - Verletzung eines Constraints
 - Deadlock-Situation zwischen mehreren Prozessen
 - Kommunikationsfehler mit dem Datenbanksystem

Das Absetzen eines SQL-Befehles kann *mehrere* Fehler *gleichzeitig* zur Folge haben.

Ein Fehler kann an die Datenbank-*Verbindung*, an das SQL *Statement* an sich oder an einen einzelnen *Datensatz* des Resultates geknüpft sein.

Ein Fehler kann das Ausführen eines SQL-Befehles verunmöglichen (Exception) oder nur behindern (Warning).

Der Exception-Mechanismus im Paket java.sql muss diesen Aspekten Rechnung tragen.

Fehlerklassen

- SQLException
- SQLWarning
- DataTruncation
- BatchUpdateException

SQLException

Basisklasse aller Fehlermeldungen in Zusammenarbeit mit SQL. Eine SQLException wird ausgeworfen, wenn ein Datenbankbefehl *nicht durchgeführt werden konnte*, weil z.B. ein Tabelle nicht vorhanden ist, nicht die nötigen Zugriffsrechte vorhanden sind, die Syntax einer Abfrage falsch ist usw.

SQLWarning

Abgeleitet von SQLException. Eine SQLWarning wird erzeugt, wenn ein Datenbankbefehl zwar durchgeführt wird, aber *ungewöhnliche Resultate* liefert, z.B. ein update-Befehl, der keine Einträge verändert. Warnings werden nicht ausgeworfen, sondern an das Objekt angehängt, das die verursachende Methode enthält.

Warnungen sind an einem Connection-, Statement- oder ResultSet-Objekt angehängt. Beispielsweise sind DataTruncation Warnungen beim Einlesen von Daten am ResultSet angehängt, weil die Warnung vom gelesenen Datensatz abhängig ist. Eine Warnung, welche meldet, dass ein Update-Befehl keinen Datensätze getroffen hat, ist demgegenüber am entsprechenden Statement-Objekt angehängt.

Warnungen können mit getWarnings() und getNextWarning() abgeholt werden. getWarnings() und getNextWarning() können nur aufgerufen werden, wenn das jeweilige Connection-, Statement- oder ResultSet-Objekt noch nicht geschlossen ist.

DataTruncation

Abgeleitet von SQLWarning. Falls Daten beim Schreiben in die Datenbank Daten abgeschnitten werden müssten, wirft der JDBC Treiber eine DataTruncation Exception. Falls Daten beim Lesen abgeschnitten werden, weil das aufnehmende Objekt aus irgendeinem Grund zu klein ist (z.B. Array Grösse), wird eine DataTruncation Exception erzeugt und an das verursachende Objekt, beispielsweise an das Re *angehängt*. Warnings und DataTruncation mit getWarnings() abgeholt werden.

BatchUpdateException

Abgeleitet von `SQLException`. Wenn bei der Ausführung eines SQL-Batches einzelne Statements des Batches fehlschlagen wird eine `BatchUpdateException` geworfen. Es ist Sache des Drivers und der Datenbank, nach einem ersten Fehler die übrigen Statements des Batches noch auszuführen oder den Batch abubrechen. In jedem Fall kann aber über die ausgeworfene `BatchUpdateException` und darin mit Hilfe der Methode `getNextException()` genau nachvollzogen werden, welche Statements des Batches durchgeführt wurden und welche einen Fehler geliefert haben. Ausserdem kann über die Methode `BatchUpdateException.getUpdateCounts()` nachvollzogen werden, welches Statement zu einer Änderung in der Datenbank geführt hat und welches nicht.

Eine `BatchUpdateException` wird auch ausgeworfen, wenn innerhalb eines Batches ein `select`-Befehl abgesetzt wird. `Select`-Befehle (resp. solche die ein `ResultSet` erzeugen) dürfen nicht in einem Batch stehen.

Falls die Bearbeitung eines Batches gar nicht gestartet werden kann, beispielsweise wegen einem Verbindungsfehler, wird eine normale `SQLException` ausgeworfen.

Fehlerklassifizierung

`SQLException.getSQLState()` liefert eine *generische*, von ANSI/XOPEN standardisierten Fehlerbeschreibung. Die entsprechende Variable heisst allgemein `SQLSTATE`.

`SQLException.getErrorCode()` liefert einen hersteller-abhängigen Fehlercode zurück.

Der Standardisierte Fehlercode besteht aus 5 Zeichen. Die ersten zwei bezeichnen eine Fehlerkategorie, die nächsten drei den genauen Fehler.

Alle Stati, die mit **01** beginnen, enthalten Warnungen. Status **01004** ist beispielsweise eine Data Truncation Warnung.

Alle Stati, die mit **23** beginnen, bedeuten eine Exception wegen Verletzung einer Integritätsbedingung.

Alle Stati, die mit **40** beginnen, weisen auf einen Rollback durch die Datenbank hin, z.B. aufgrund eines Deadlocks.

Alle Stati, die mit **42** beginnen, weisen auf einen Syntaxfehler oder eine Verletzung der Zugriffsrechte hin.

Beispiel Fehlerbehandlung

```
public void sendSQL ( Connection con, Statement stmt )
{
    boolean retry = true;
    while ( retry ) {
        try {
            stmt.execute();
            retry = false;
            con.commit();
        }
        catch( SQLException e ) {
            if ( e.getSQLState().equals( "40001" ) ) {
                System.out.println("Deadlock!");
                retry = true;
            }
            con.rollback();
        }
    }
}
```

Dieses Code-Beispiel zeigt eine Fehlerbehandlung für eine Transaktion bestehend aus einem einzigen SQL-Befehl mit Retry-Mechanismus für den Fall eines Deadlocks. Dieses Beispiel kann sinngemäss ergänzt werden, wenn eine Transaktion mehrere SQL-Befehle umfasst. Beispielsweise kann dann mit einem Batch-Statement gearbeitet werden oder einer Liste von Einzelstatements. Der catch-Block muss dann unter Umständen wie folgt ergänzt werden

```
catch( SQLException e ) {
    while ( e != null ) {
        String sqlstate = e.getSQLState();
        // zu unternehmende Aktionen ...
        e = e.getNextException();
    }
}
```

Multithreading

- Das native Datenbank-Protokolle auf *einer* Connection ist i.a. nicht multithreaded.
- Der JDBC-Standard macht *keine Aussage* bezüglich Synchronisation und Thread-Sicherheit der Java-Methoden.
- Die Verantwortung für die Synchronisation von Threads welche diesselben Connection-, Statement- und ResultSet-Objekte benützen, liegt beim Entwickler.
- In der Regel wird man *pro* Thread daher mit *einer* Connection arbeiten. Parallelität ist damit bis hinunter auf das DBMS gewährleistet.
- Beanspruchen mehrere Threads *dieselbe* Connection muss synchronisiert und die Transaktionsverantwortung geregelt werden.

Eine sinnvolle Anwendung von Multithreading ist die Abbruchmöglichkeit für laufende SQL-Befehle über `Statement.cancel()`.

Ob ein Treiber-Paket seine Methoden synchronisiert oder nicht, ist fallweise abzuklären.

Unterbrechbare Queries

- SQL-Abfragen können längere Zeit dauern, weil
 - die Abfrage komplex ist
 - der Datenbestand gross ist
 - der DB-Server hoch belastet ist
 - der Transfer zum Client lange dauert
- Der Benutzer oder der aufrufende Benutzerprozess will die Möglichkeit besitzen eine laufende Abfrage abubrechen oder eine maximale Dauer festzulegen.
- Mit folgenden Methoden kann die Ablaufzeit kontrolliert werden:
`Statement.setQueryTimeout( int secs )`
`Statement.cancel()`

In Web-Applikation ist ein häufiges Verhaltensmuster wie folgt: Aufgrund eines HTML-Submit-Aufrufes eines Browsers wird beim Webserver eine Datenbank-Abfrage gestartet. Wenn diese lange dauert, führt der Benutzer ein "Reload" durch oder startet den Aufruf nochmals und setzt damit eine zweite identische Abfrage bei der Datenbank in Gang. Die erste Abfrage läuft immer noch, der Webserver bemerkt nämlich nicht, dass der Browser nicht mehr auf das Ergebnis des ersten Aufrufes wartet. Ein Webserver (-Servlet) bemerkt das "Fehlen" des Browsers erst beim Versuch, Daten zum Browser zu übertragen, d.h. beim Schreiben auf das `ServletResponse`-Objekt. Dies geschieht in der Regel aber erst, wenn die Datenbank-Abfrage durchgeführt wurde.

Ein verbessertes Szenario wäre nun wie folgt: Das verantwortliche Servlet setzt die Datenbankabfrage in einem eigenen Thread ab, und testet währenddessen periodisch, ob das `ServletResponse`-Objekt noch offen ist (zum Beispiel durch Schreiben von Leerzeichen und/oder eine `flush()`-Operation). Wird das `ServletResponse`-Objekt ungültig, kann das verantwortliche Servlet das laufende SQL-Statement mit der `cancel()`-Methode unterbrechen.

Als einfache, aber grobe Variante, um lang dauernde SQL-Abfragen zu unterbrechen, kann auch ein generelles Timeout auf ein Statement gesetzt werden mit `Statement.setQueryTimeout()`.

Das Unterbrechen von laufenden SQL-Statements mit `setQueryTimeout()` oder `cancel()` bedingt, dass der JDBC-Driver und das native Datenbank-Protokoll das Unterbrechen von Queries unterstützt.

Neuerungen in JDBC 3.0

- Neue Interfaces für SQL99 Datentypen und entsprechende set/get Methoden für Clob, Blob, Array, boolean, URL, Ref
- Abholen von autogenerierten Schlüsseln mit `Statement.getGeneratedKeys()`
- Interface `Savepoint`
- Interfaces für das ConnectionPooling
- Ein `RowSet` als Ableitung von `ResultSet` für den direkten Einsatz als `JavaBean` in einer GUI-Komponente.
- Verknüpfung der Connector Architecture zu JDBC Interfaces.
- Interface `ParameterMetaData` für mehr Information über Parameter in Prepared- und Callable-Statements.

Driver Typ 1: JDBC-ODBC Bridge

- JDBC gibt Datenbankaufrufe an eine ODBC Library weiter. ODBC- und DBLibrary-Code ist *lokal* und *plattformabhängig*. Der ODBC-Code ist in C geschrieben und wird über das Java native API angesprochen.
- Gut geeignet für Java-Programme, die auf lokale Datenbanken ohne DBMS zugreifen, z.B. Microsoft Access. In diesem Fall müssen die Java Applikation, der ODBC-Code(.dll Files) und die DB auf demselben Rechner installiert sein.
- Greift man via ODBC auf SQL-Server zu, können letztere auf einem separaten Rechner laufen.
- Nicht geeignet für den Einsatz mit Applets, da Zugriff auf lokale installierten ODBC-Code nötig (Security Einschränkungen).
- Breit verfügbar für verschiedenste DBMS, z.B. bei Intersolv.

Driver Typ 2: Java - native API

- JDBC gibt Datenbankaufrufe an eine herstellerspezifische C-Bibliothek weiter. Beispiel: OpenClient Library von Sybase.
- Während ODBC häufig für filebasierte DB's eingesetzt wird, werden Typ 2 Driver vorallem für grosse SQL Datenbanken eingesetzt (Oracle, Sybase, Informix, DB2, Ingres, ...).
- Typ 2 Driver sind einfach zu implementieren, benötigen aber wie ODBC Zugriff auf lokale Ressourcen. Sie sind eine Übergangslösung zu Typ 3 und 4 Treibern.
- Nicht geeignet für Applets da lokale Resource benötigt.
- Performance: ca. 3-4 mal schnellerer Verbindungsauf als mit Typ 1 und 2 (Dr. Dobbs Journal, 1/98). Keine wesentlichen Unterschiede für Abfragen zu den anderen Driver Typen.

Driver Typ 3: Pure Java Net Protocol

- Der Client kommuniziert nicht mit der DB, sondern mit einem Gateway-Prozess, resp. Tunneling Servlet, welcher seinerseits der eigentliche Datenbank-Client ist.
- Es wird nur mit Java Code gearbeitet. Kein Zugriff auf lokale, plattformspezifische Ressourcen.
- Von allen grossen DB-Herstellern unterstützt.
- Beste Lösung für Applets, sehr schlank.
- Gateway kann unterschiedlichste DB-Systeme bedienen, z.B. bei Openlink

Driver Typ 4: Pure Java DB Protocol

- Es wird nur mit Java Code gearbeitet. Kein Zugriff auf lokale, plattformspezifische Ressourcen.
- Direkte Verbindung zwischen Client und Datenbank-Server über das native Datenbank-Protokoll.
- Heute häufigste Variante für alle 2-Tier Applikationen.