**Select:**

SELECT[*, Attribut1 [AS Name], Attribut2 [AS Name], ...] FROM [Tabellenname, SELECT]
[WHERE Bedingung(>, >=, >=, <, =, NOT IN, NOT LIKE, NOT, IS NULL] [AND] [OR] [FROM SELECT]
[HAVING Bedingung]
[JOIN]
[ORDER BY Attribut [DESC, ASC], Attribut2 [DESC, ASC], ...] //DESC = Z-A ASC=A-Z
[GROUP BY Attribut]

Alternative:

SELECT alias.Attribut1, alias.Attribut2
FROM tabellenname alias;
SELECT Attribut1 FROM Tabellenname alias

Aggregatfunktionen:

COUNT(Attribut [+, -, *, /, %] [KONSTANTE]) [AS Name]//Zählen
AVG(Attribut [+, -, *, /, %] [KONSTANTE]) [AS Name]//Durchschnitt
MAX(Attribut [+, -, *, /, %] [KONSTANTE]) [AS Name]//Maximum
MIN(Attribut [+, -, *, /, %] [KONSTANTE]) [AS Name]//Minimum
MEDIAN(Attribut [+, -, *, /, %] [KONSTANTE]) [AS Name]//Mittelwert
STDDEV(Attribut [+, -, *, /, %] [KONSTANTE]) [AS Name]//Stadardabweichung
SUM(Attribut [+, -, *, /, %] [KONSTANTE]) [AS Name]//Summe
VARIANCE(Attribut [+, -, *, /, %] [KONSTANTE]) [AS Name]//statische Varianz
ROUND(Attribut [+, -, *, /, %] [KONSTANTE]) [AS Name]/Rundet
UPPER(Attribut)//Gross

SELECT Attribute1, Attribut2 [*, /, +, -][KONSTANTWERT] FROM Tabellenname;

**Update:**

UPDATE Tabellenname
SET Attribute1 = Value1, Attribute2 = Value2, ...
[WHERE Bedingung(>, >=, >=, <, =, NOT IN, NOT LIKE, NOT, IS NULL] [AND] [OR] S

**Delete:**

DELETE FROM Tabellenname
[WHERE Bedingung(>, >=, >=, <, =, NOT IN, NOT LIKE, NOT, IS NULL] [AND] [OR]
DELETE FROM Tabellenname //löscht alle Einträge der Tablelle
TRUNCATE TABLE Tabellenname //löscht alle Einträge der Tabelle effizienter

**Insert:**

INSERT INTO Tabellenname (Attribut1, Attribut2)
VALUES (Value1, Value2)
INSERT INTO Tablellenname (Attribut11, Attribut2)
SELECT .... //Selectierte Daten werden eingefüllt

**Join:**

SELECT * FROM Tabelle1 [LEFT JOIN][RIGHT JOIN] Tabelle2
ON Tabelle1.Attribut1 = Tabelle2.Attribute2
//Zusammenführung von verschiedenen Tabellen
LEFT JOIN zeigt die ganze Tabelle1 auf (Nulleinträge bei Tabelle2)
RIGHT JOIN zeigt die ganze Tabelle2 auf (Nulleinträge bei Tabelle1)
Bsp:

SELECT * FROM Abteilungen LEFT JOIN Personen
ON Abteilungen.AID = Personen.AID

| Abteilungen.AID | Abteilungen NAME | Personen.AID | Personen.Name |
|---|---|---|---|
| 1 | Z | 1 | A |
| 1 | Z | 1 | B |
| 4 | W | NULL | NULL |
| 5 | V | NULL | NULL |

**Group:**

SELECT Attribut1, COUNT(Attribut2)
FROM Tabellenname
GROUP BY Attribut1 //Attribut1 wird „gruppiert“ (nicht einzeln aufgeführt)

| Attribut1 | COUNT(Attribut2) |
|---|---|
| bsp1 | 35 |
| bsp2 | 56 |

**Sonstiges:**

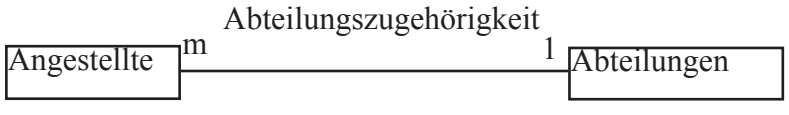
CREATE TABLE Tabellename//erstellt Tabelle
GRANT SELECT, INSERT, UPDATE, DELETE ON Datenbankname.* TO Username IDENTIFIED BY „Passwort“//Berechigun

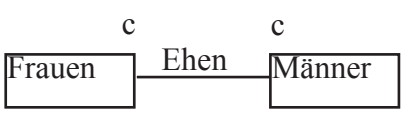
**0.Normalform** Daten in Exccelformat
**1.Normalform**
Keine Schachtelung
PID
11,12
wird zu:
PID
11
12
neu Tabelle bei NULL Einträgen. Identifikationsschlüssel pro Relation bestimmen!
ID | Name | K | J
105| Peter
wird zu:
ID | Name K | J
105| Peter |
**2.Normalform**
Tabelle muss Kriterien von 1. Normalform erfüllen
Jedes nicht zum Identitätsschlüssel gehörge Attribut muss vom ID-Schlüssel abhängig sein.
Achtung:Bei nicht verwendeten Schlüsseln -> neue Tabelle

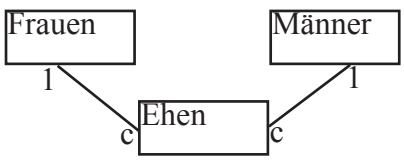
| IDSchlüssel | Attribut |
|---|---|
| Pid | Attribut1, Attribut4, Attribut7 |
| Pjid | Attribut2, Attribut3 |
| Pid, Pjid | ... |

**3.Normalform**
Tabelle muss Kriterien von 2. Normalform erfüllen
Keine transitiven Abhängigkeiten, d. h alle Attributte müssen direkt (und voll) vom ID_Schlüssel abhängig sein.
PID->AbtID AbtID->AbtName
PID->->AbtName
PID | PName | AbtID | AbtName
wird zu:
PID | PName | AbtID AbtID | AbtName
**Globale Normalisierung**
Nur noch globale(muss in mind. einer Relation als Identifikationsschlüssel vorkommen)
und lokale(darf nur in einer Relation, und nicht als Identifikationschlüssel, vorkommen) Attribute
Achtung : Name egal -> Bedeutung wichtig!!
**Generalisierung**
Student(SNr, Name, Adresse, Jahr)
Fussballer(FNr, Name, Adresse, LGA)
wird zu:
Personen(PNr, Name, Adresse)
Studenten(Snr, Jahr)
Fussballer(FNr, Liga)
Global Normalisiert!->referentielle Integrität -> 3.NF & nur globale & lokale Attribute & alle Fremdschlüssel definiert

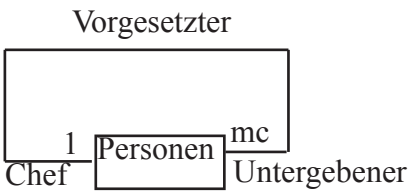
**c-c, c-m, c-cm und m-m, mc-mc, m-mc, mc-c Beziehungen auflösen:**
1 : einfache Assoziation genau eine Entität
c: konditionelle Assoziation 0 oder 1 Entität
m: multiple Assoziation (1..M)
mc: multiple konditionelle Assoziation (0..M)
1-c und 1-mc Beziehungen sind gut 1-c und 1-m sind gut mit Applikationslogik

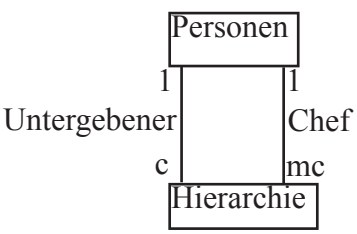



**rekursive Beziehungen auflösen:**



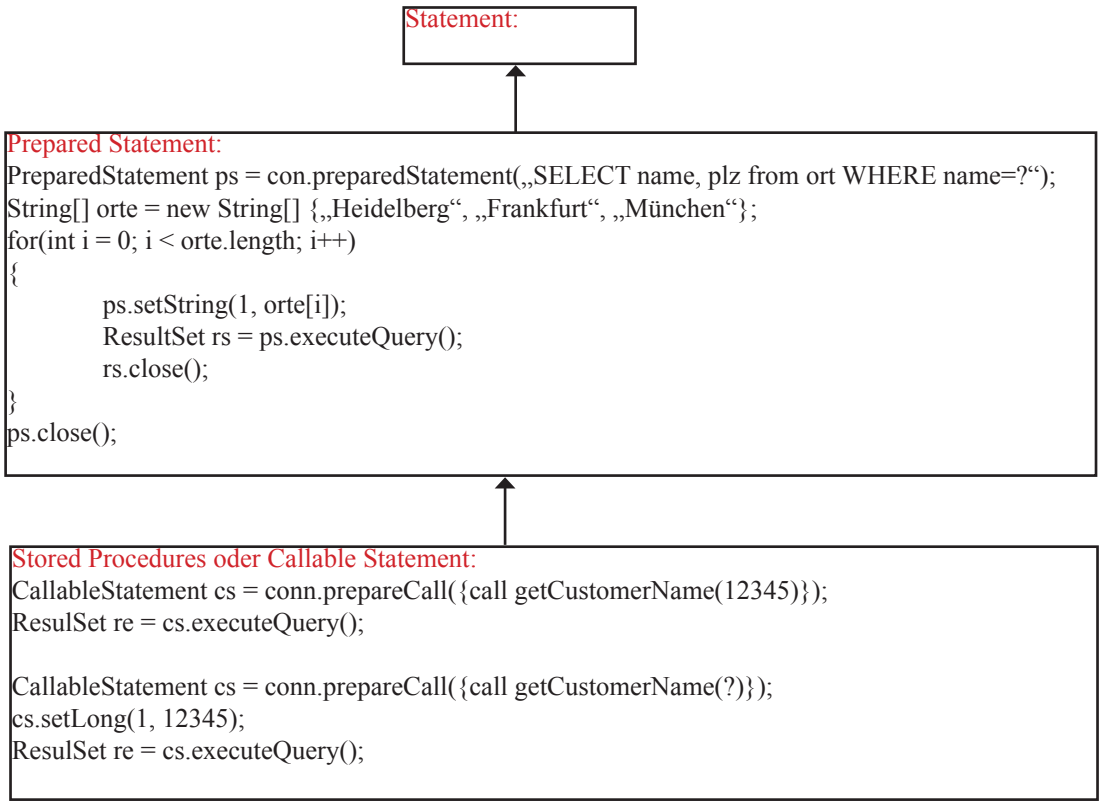
**NULL Werte:**
Definition : NULL Werte sind undefinierter Speicherbereich oder unbekannt
Abfrage : IS NULL
NULL = NULL = 0 oder weder true noch false
NULL Werte Joinen nicht
NULL in arithmetischen Ausdrücken = ganzer Ausdruck NULL
Im Relationenmodell dürfen nur lokale Attributte NULL aufweisen

**4.Generationssprachen:**
Vorteile: Standardapplikationen schnell erstellt (ändern, einfügen), braucht wenig Programmierkenntnisse,
Nachteile : Keine komplette Buisnesslogik (nur mit 3.Generation Sprache), ...

**JDBC 1:**
aktuell : Version 3.0 (Version 4.0 ist in Entwicklung)
Packages : java.sql, javax.sql
Struktur : Driver Manager
-> Connection
-> Database Metadata (Info zur DB)
->Statement (kapselt SQL Befehle)
->Resultset (kapselt Select Ergebnisse) (Mappt Datentypen)
Nach Transfer wird Ressource wieder freigegeben!

Schnittstellen, gültig für beliebiges DMS:
-JDBC(Java Database Connektivity)
-ODBC(Open Database Connektivity) (Systemsteuerung->Verwaltung->DatenQuellen (odbcad32.exe))
-OLEDB
-und weiter abstrakte Schnittstellen (Java Data Objects, Hibernate, ADO (Active Data Objects) LINQ)

**Spezielle Statements:**
Stored Procedures sind Datenbank-Prozeduren, die in einer speziellen Sprache des DBMS Herstellers implementiert sind (PL/SQL bei Oracel, SQLPM bei DB2), un düber JDBC aufgerufen werden können.
Ähnlich wie bei einem PreparedStatement kann die Datenbank hier den Statement-Cache verwenden, um den Access-Plan nicht bei jede Aufruf erneut berechnen zu müssen. Ein CallableStatement mit Platzhaltern hat oftmals Performance-Vorteile.
über Performance gibt es keine allgemeingültige Aussage, aber:
-Metadata-Methoden 8getTypeInfo(), getTables()) sind teuer -> Cachen lohn sich
-es ist besser auf einem ResultSet die Methode getMetaData() aufzurufen als getColums(„Tabellename“) auf einer Connection
-fordere möglichst wenig Daten an (dont select *), denn sie müssen alle übers Netzwerk transportiert werden.



```
// In Java muss zum Kompilieren das JDBC-API importiert werden.
import java.sql.*;
class JDBCManipulate
{
	// Diese statische Methode wird beim Starten des Programms ausgeführt.
	// Starten Sie das Programm mit: java JDBCTest. Im String-Array args
	// stehen die der Kommandozeile übergebenen Argumente. args[0]
	// enthält bereits das erste Argument und nicht den Klassennamen wie in C.
	public static void main(String[] args)
	{
		if (args.length != 2)
		{
			System.out.println(„Bitte geben Sie ein SQL-Manipulationsstatement und den Transaktionscode c oder r an!“);
			System.exit(1);
		}
		try
		{
			// JDBC-Treiber laden: In diesem Fall arbeiten wird mit der JDBC-ODBC- Bridge
			Class.forName(„sun.jdbc.odbc.JdbcOdbcDriver“);
			// Connection herstellen. Als Argument wird eine URL angegeben:
			// Protokoll: jdbc; Subprotokoll: odbc; ODBC-Datenquelle: transactions
			Connection con =DriverManager.getConnection(„jdbc:odbc:transactions“);
			//für Netzwerkdatenbank mySQL
			Class.forName(„com.mysql.jdbc.Driver“).newInstance();
			Connection con = DriverManager.getConnection(„jdbc:mysql://Pfad_zu_Server/Datenbankname?user=name&password=pass“);
			// Statement herstellen, um ein Tabelle lesen zu können
			Statement s = con.createStatement();

			// Das SQL-Statement unter Transaktionskontrolle ausfuehren
			con.setAutoCommit(false);
			try
			{
				s.executeUpdate(sql1);
				s.executeUpdate(sql2);
				s.executeUpdate(sql3);
				con.commit();
			}
			catch(SQLException e)
			{
				con.rollback();
			}
			//Mit ResultSet arbeiten
			ResultSet rs = s.executeQuery(„Select * from „ + args[0]);
			// Zunächst wird die Kopfzeile mit den Spaltennamen ausgegeben.
			// Nun mit Metadaten
			ResultSetMetaData rsmd = rs.getMetaData();
			int i;
			for (i = 1; i <= rsmd.getColumnCount(); i++)
			{
				System.out.print(rsmd.getColumnName(i));
				System.out.print(‚\t‘);
			}
			System.out.println();
			// Das ResultSet kann nun ausgelesen werden. In JDBC beginnt das erste
			// Element mit 1. (Sonst ist 0 als erster Index üblich.)
			int id; String comment;
			while (rs.next())
			{
				for (i = 1; i <= rsmd.getColumnCount(); i++)
				 {
					System.out.print(rs.getObject(i));
					//oder rs.getString(1); rs.getInt(2); rs.getString(„feldname“);
					System.out.print(‚\t‘);
				}
				System.out.println();
			}

			//Generierte Schlüssel zurückgeben lassen
			s.executeUpdate(„INSERT INTO Tabelle (Attribut) VALUES (‚test‘)“, Statement.RETURN_GENERATED_KEYS);
			ResultSet keys = s.getGeneratedKeys();

			//Blobs
			Blob data;
			File f2 = new File(„c:/Pfad.jpg„);
			FileOutputStream fis = new FileOutputStream(f2);
			ResultSet res = s.executeQuery(„select * from Blobs“);
			while(res.next())
			{
				data = res.getBlob(2)
				fis.write(data.getBytes(1, (int)data.length()));
			}

			// Am Schluss werden noch alle Ressourcen im JDBC-Treiber und der
			// Datenbank freigegeben. Diese können nicht über den Java-Garbage-Collector
			// gesammelt werden.
			rs.close();
			s.close();
			con.close();
		}
		catch (Exception ex)
		{
			// Jede Datenbankoperation kann einen Fehler verursachen. Diese Fehler werden hier
			// abgefangen. Der Return-Wert 1 gibt die Fehlersituation nach aussen weiter.
			ex.printStackTrace();
			System.exit(1);
		}
		 System.exit(0);
	}
}
```

**Transaktionen:**

**Motivation:**

Datenbanken stellen ihre Ressourcen mehreren Nutzern gleichzeitig zur Verfügung
Synchronisierungsprobleme und dadurch DB-Konsistenzverletzungen möglich

**Definition:**

„Block von SQL Anweisungen, der nicht unterbochen werden darf“ (Logic unit of work)
„Im Datenbankbereich versteht man unter dem Begriff der Transaktion eine Folge von Operationen, die eine Datenbank von einme konsistenten Zustand in einen möglicherweise neuen, aber wieder konsistenten Zustand überführt und dabei das ACIS-Prinzip einhält.“
„In SQL ist eine Transaktion eine Sequenz von SQL Anweisungen, die atomar ausgeführt wird. Das bedeutet, dass die Transaktion entweder vollständig ausgeführt wird oder keinerlei Effekt auf der Datenbank hinterlässt.
A Atomicity = Transaktion wird als Einheit betrachtet. (Alles oder nichts).
Die Folge der Operationen soll entweder vollständig oder gar nicht ausgeführt werden, d.h beim Abbruch einer Transaktion wird die Datenbank auf den Zustand vor dem Beginn dieser Transaktion zurückgeführt.
C Consistency = Referentielle Integrität bleibt erhalten.( am Ende einer erfolgreichen Transaktion).
Mit dem Ende einer Transakton (also auch beim Abbruch) müssen alle Integritätsbedingungen erfüllt sein - die Datenbank ist wieder in einem konsistente Zustand.
I Isolation = Modifikationen sind erst beim Aschluss einer Transaktion nach aussen sichtbar.
Bei der Ausführung einer Tansakttion in einer Mehrbenutzerumgebung sollte ein Nutzer den Eindruck haben, dass er alleine mit der Datenbank arbeitet. Dies verbietet somit unerwünschte Nebeneffekte durch konkurrierende Zugriffe.
D Durability = Modifikationen einer erfolgr. abgeschlossenen Transaktion sind dauerhaft vorhanden (auch nach Systemabsturz).
Nach Abschluss der Transaktion sollen alle durchgeführten Änderungen dauerhaft in der Datenbank auf dem Externspeicher festgeschrieben sein. Des ist insbesondere im Zusammenhang mit der Pufferverwaltung vn Bedeutung, da Seiten nach Transaktionsende durchaus im Puffer verbleiben.

-Nicht jedes der vier Conrarrency-Problemen benötigt das Serialisierbarkeits Protokoll!!!!
-Jede Transaktion kann eine geeignete Isolationsstufe wählen.
-Immer möglich sind Blockaden und Deadlocks.
-Jede Transaktion hat einen Start und ein Ende. Erfolg = commit(), Misserfolg (Abbruch vom Benutzer) = rollback()
-Start einer Transaktion ist implizit, d.h das Ende einer Transaktion signalisiert den Start einer neuen Transaktion.
-Transaktionen können sequentiell oder parallel durchgeführt werden, sequentiell = immer korrektes Ergenis, parallel = erfordert spezielle Schutzmassnahmen.

**In JDBC:**

Der Beginn einer Transaktion muss nicht extra erwähnt werden, nur Autocommit auf false setzen
Connection c.setAutoCommit(false);
Am Ender entweder c.commit() oder c.eollback()

**2 Phase Locking Protokoll:**

Alle Concurrency-Probleme lassen sich durch Locks und das 2-Phase-Locking-Protokoll lösen, sodass die ACID Eigenschaften eingehalten werden.

ABER: Blockierungen können zu Timeouts führen, und Deadlocks können auftreten.....

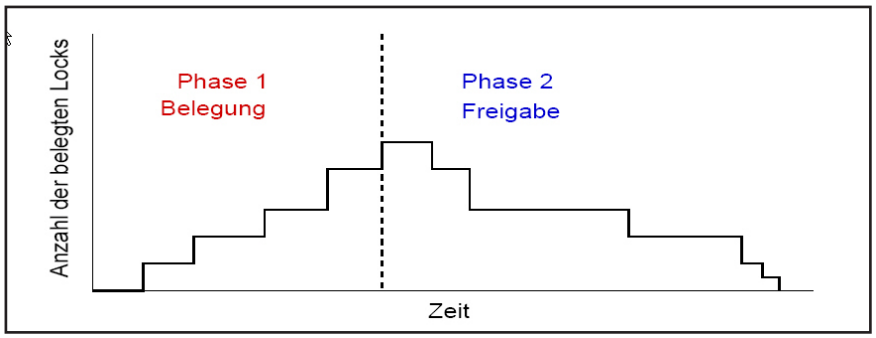
Es gibt Shared Locks (Read Locks, S) und Exclusive Locks (Write Locks, X) sowie Unlock U

| $T_A$ hält \ $T_B$ will | X | S | No Lock |
|---|---|---|---|
| X | Nein $T_B$ wird blockiert | Nein $T_B$ wird blockiert | Ja |
| S | Nein $T_B$ wird blockiert | Ja | Ja |
| No Lock | Ja | Ja | Ja |

T kann einen S-Lock in einen X-Lock umwandeln, sofern keine weiteren S-Locks bestehen!!

Konservaives 2PL:alle Locks direkt am Anfang anfordern.

Striktes 2PL:alle Locks erst ganz am Ende der Transaktion freigeben (alle auf einmal)




**Nicht sperrende Verfahren**

-Serialisierbarkeitsteste
-Optimistische Synchronisation
-Zeitstempelverfahren

**Serialisierbarkeit:**

T1= r1(x) r1(y) w1(x)
T2=r2(y) w2(y)

verschränkter Schedule:
s1= r1(x) r1(y) r2(y)w2(y)w1(x)

serieller Schedule (Serialisierbarkeit):
s2=T1T2=r1(x)r1(y)w1(x)r2(y)w2(y)

## Deadlock & Blocking bei 2 Phases Locking Protokoll:

| **Lost Updates** | | Deadlock | |
|---|---|---|---|
| | **Transaction A** | **time** | **Transaction B** |
| | ... | | ... |
| | Retrieve t | t1 | |
| | acquire S-Lock on t | | ... |
| | ... | t2 | Retrieve t |
| | ... | | acquire S-Lock on t |
| | Update t | t3 | ... |
| | request X-Lock on t | | ... |
| | wait | t4 | Update t |
| | wait | | request X-Lock on t |
| | wait | | wait |

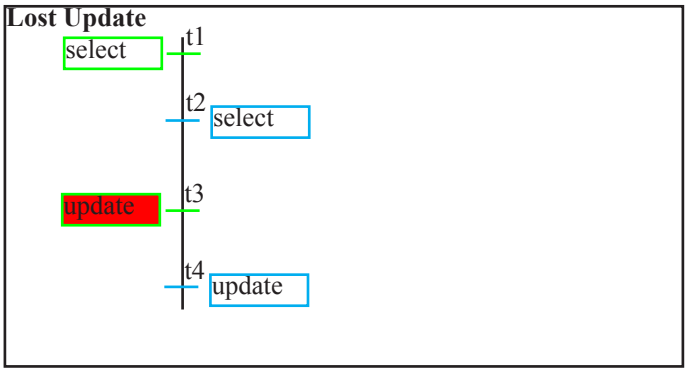
| **Uncommitted Dependency** | | blocking | |
|---|---|---|---|
| a) | | | |
| | **Transaction A** | **time** | **Transaction B** |
| | ... | | ... |
| | Update t | t1 | ... |
| | acquire X-Lock on t | | ... |
| | ... | t2 | Retrieve t |
| | ... | | request S-Lock on t |
| | Rollback | t3 | wait |
| | release-X-Lock on t | | wait |
| | | | resume |
| b) | | | |
| | **Transaction A** | **time** | **Transaction B** |
| | ... | | ... |
| | Update t | t1 | ... |
| | acquire X-Lock on t | | ... |
| | ... | t2 | Update t |
| | ... | | request X-Lock on t |
| | Rollback | t3 | wait |
| | release-X-Lock on t | | wait |
| | | | resume |

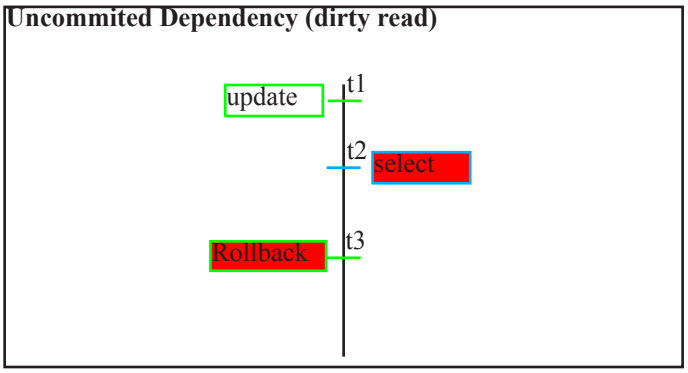
| **Inconsistent Analysis** | | Deadlock | |
|---|---|---|---|
| | acc1:40 | acc2: 50 | acc3: 30 |
| | **Transaction A** | **time** | **Transaction B** |
| | ... | | ... |
| | Retrieve acc1; sum = 40 | t1 | ... |
| | acquire S-Lock on acc1 | | ... |
| | Retrieve acc2; sum = 90 | t2 | ... |
| | acquire S-Lock on acc2 | | ... |
| | ... | t3 | Retrieve Acc3 |
| | | | acquire S-Lock on acc3 |
| | ... | t4 | Update Acc3 (20) |
| | | | acquire X-Lock on acc3 |
| | ... | t5 | Retrieve Acc1 |
| | | | acquiere S-Lock on acc1 |
| | ... | t6 | Update Acc1 (50) |
| | | | request X-Lock on acc1 |
| | ... | t7 | wait |
| | Retrieve acc3; sum = 110 | t8 | wait |
| | request S lock on acc3 | | wait |

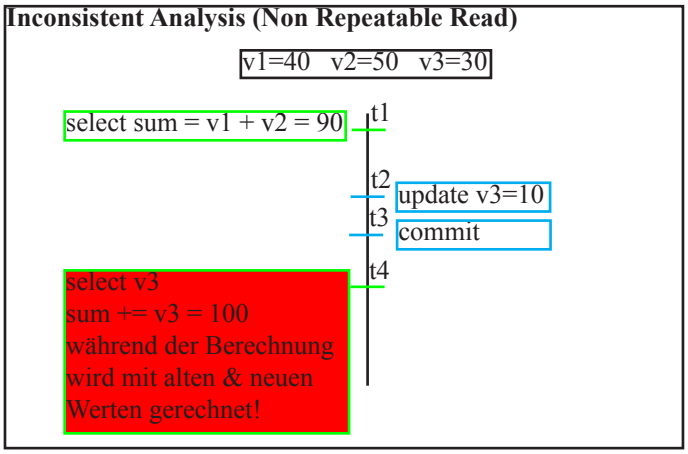
| **Phantoms** | | blocking | |
|---|---|---|---|
| | **Transaction A** | **time** | **Transaction B** |
| | ... | | ... |
| | Retrieve t1, t2, t3 | t1 | ... |
| | acquire S-Lock on t1, t2, t3 | | ... |
| | ... | t2 | Delete t2 |
| | ... | | request X-Lock on t2 |
| | ... | t3 | wait |
| | ... | | wait |
| | Retrieve t1, t2, t3 | t4 | wait |
| | ... | | wait |
| | Commit | t5 | wait |
| | release all locks | | wait |
| | | t6 | resume |

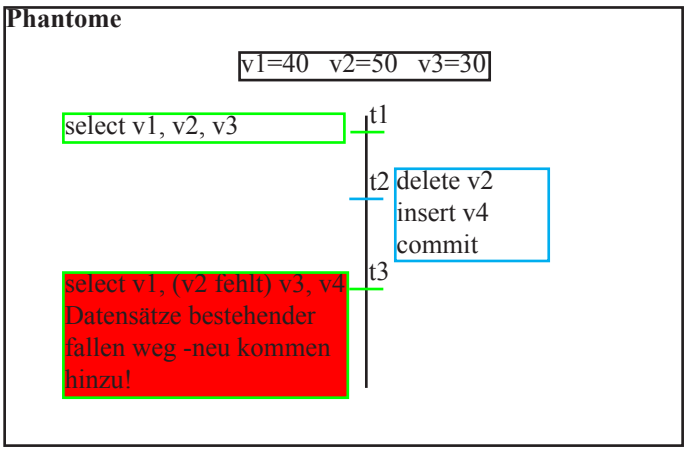
**Concurrency Probleme:**

Nebenläufigkeit bezeichnet das Zusammenspiel mehrerer Transaktionen, die dieselbe Ressourcen (Daten) nutzen.
Die Probleme, die dabei entstehen, sind:



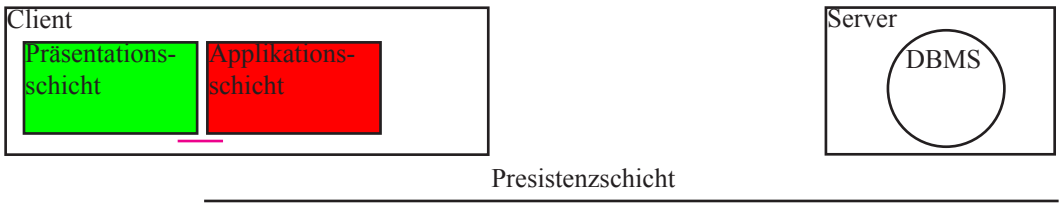

| | |
|---|---|
| Lost Update | Wenn 2 Transaktionen dieselben Daten lesen und schreiben<br>Mit Locks kann ein Lost Update Szenarie zu einem Deadlock führen.<br>Oder:<br>Eine Änderung geht verloren, wenn zwei nebenläufige Transaktionen zunächst auf das Datenobjekt lesen und danach unabhängig voneinander ändern und in die Datenbank zurückschreiben. |
| Uncommited Dependency (Dirty Read) | Dieser Fall tritt auf, wenn die zweite Transaktion ein Tupel liest, das von der ersten geändert, aber noch nicht commited worden ist.<br>Oder:<br>Eine Transaktion liest ein Datenojekt, das zuletzt von einer anderen, noch nicht abgeschlossenen Transaktion geschrieben wurde. Bricht die „schreibende“ Transaktion nach dem Commit der „lesenden“ Transaktion ab, so hat Letztere möglicherweise ungültige Werte gelesen. |
| Inconsistent Analysis (Non Repeatable Read) | Die erste Transaktion bekommt unterschiedliche Resultate, wenn die das Tupel mehrfach liest.<br>Oder:<br>Eine Transaktion liest in zwei aufeinander folgenden Leseoperationen zwei verschiedene Werte desselben Datenobjekts. |
| Phantome | Die erste Transaktion liest mehrere Tupel. Beim zweiten mal fehlen Tupel, oder es sind neue hinzugekommen.<br>Oder:<br>Eine Transaktin liest Dtenobjekte, während nebenläufig eine andere Transaktion Datenobjekte einfügt oder löscht. |

Die Isolationsstufe bezeichnet den Grad der zulässigen Interferenz (Überlagerung) zwischen den Transaktionen. Ziel ist die Steigerung des Durchsatzes, Steigerung der Performance. Die Isolationsstufen beziehen sich immer auf eine Transaktion. Verschiedenen Transaktionen können unterschiedliche Isolationsstufen verwenden. Die Isolationsstufe macht eine Aussage über das Leseverhalten einer Transaktion. Grundsätzlich kann jede Transaktion, egal in welchem Isolationslevel sie läuft, blockiert werden, oder in einen Deadlock geraten!!!!!
Blockaden führen zu Verzögerungen oder zu Timeouts (mit implizitem Rollback).
Bei Deatlocks bestimmt das DBMS ein Opfer (Deadlock Victim), bei der ein Rollback durchgeführt wird.
Es gibt neben den ANSI-Isolationsstufen noch ander (Snapshoot), die eingeführt wurden, um Leseoperationen nicht zu blockieren!
In SQL gibts vier Isolationsstufen:

| Isolationsstufe | Lost Update | Uncommited Dependency (dirty Read) | Inconsistent Analysis (Non Repeatable Read) | Phantome |
|---|---|---|---|---|
| Read Uncommitted | - | X | X | X |
| Read Committed | - | - | X | X |
| Repeatable Read | - | - | - | X |
| Serializable | - | - | - | - |

| | |
|---|---|
| Serializable<br>Level 3<br>(Stärkste) | T kann Tupel mehrfach lesen und die Anzahl der Tupel bleibt gleich.<br>Alle anderen T, die das Nachlesen verhindern, werden blockiert (also, wenn andere T update, delete oder insert ausführen). Auch neu Erstellte Daten werden für andere Transaktionen blockiert!!!<br>Oder:<br>steht für volle Serialisierbarkeit.<br>Oder:<br>Die höchste Isolationsebene garantiert, dass die Wirkung parallel ablaufender Transaktionen exakt die selbe ist wie sie die entsprechenden Transaktionen zeigen würden, liefen sie nacheinander in Folge ab. Auf diese Weise ist sichergestellt, dass keine Transaktion verloren geht und dass sich keine zwei Transaktionen gegenseitig in die Quere kommen. Da die meisten Datenbanksysteme allerdings nur eine Illusion von sequentieller Ausführung aufrecht erhalten, ohne tatsächlich alle Transaktionen nacheinander einzeln abzuarbeiten, kann es hier vorkommen, dass eine Transaktion von der Seite der Datenbank aus abgebrochen werden muss. Eine Anwendung, die mit einer Datenbank arbeitet, bei der die Isolationsebene Serializable gewählt wurde, muss daher mit Serialisationsfehlern umgehen können und die entsprechende Transaktion gegebenenfalls neu beginnen. |
| Repeatable Read<br>Level 2 | T kann mehrfach lesen, eventuell kommen dabei weitere Tupel hinzu.<br>Alle anderen T, die das Nachlesen verhindern, werden blockiert (also, wenn ander T update oder delete ausführen - insert ist erlaubt)<br>Oder:<br>bedeutet ebenfalls, dass die entsprechende Transaktion nur „definitive“ Daten lesen kann. Zudem hält die Transaktion Sperren für alle Tupel, die von ihr gelesen und/oder geschrieben wurden. Sie hält jedoch keine sogenannten Prädikatsperren. Damit kann nur noch das Phantomproblem auftreten.<br>Oder.<br>Bei dieser Isolationsebene ist sichergestellt, dass wiederholte Leseoperationen mit den gleichen Parametern auch die selben Ergebnisse haben. Üblicherweise wird dies sichergestellt, indem eine Transaktion nur Daten sieht, die vor ihrem Startzeitpunkt vorhanden waren. Eine parallele Änderung führt somit auch nach commit nicht zu Inkonsistenzen während einer Transaktion. Dennoch ist es möglich, dass zwei Transaktionen parallel den selben Datensatz modifizieren und nach Ablauf dieser beiden Transaktionen nur die Änderungen von einer von ihnen übernommen wird. Soll die Isolationsebene nicht erhöht werden, können solche Probleme eventuell auch durch Locking umgangen werden. |
| Read Commited<br>Level 1 | T kann Änderungen von anderen T nur lesen, wenn diese T beendet sind.<br>T wird blockiert, wenn andere T Daten verändern (also insert, update oder delete ausführen)<br>Oder:<br>bedeutet, dass die entsprechende Transaktion nur „definitive“ Daten lesen kann. Die Transaktion hält keine Sperren für Tupel, die von ihr gelesen wurden, wohl aber fü r solche, die von ihr modifiziert wurden. Da keine Sperren fü r gelesene Tupel gehalten werden, ist es möglich, dass bei wiederholtem Lesen eines bestimmten Tupels dieses zwischenzeitlich verändert wurde. Auch das Phänomen des inkonsistenten Lesens sowie das Phantom-Problem kann auftreten.<br>Oder:<br>Unterschied zur vorhergehenden Ebene sind hier Änderungen einer parallel ablaufenden Transaktion erst nach einem commit sichtbar. Das bedeutet, dass Transaktionen lediglich vor Daten geschützt sind, die am Ende einer anderen Transaktion nicht übernommen werden. Solche Daten sind üblicherweise fehlerhaft (sonst würden sie ja am Ende der Transaktion committed werden). Diese Isolationsebene ist bei vielen Datenbanksystemen die Voreinstellung, da sie - nach dem vollständigen Fehlen jeder Isolation - am einfachsten zu implementieren ist. |
| Read Uncommited<br>Level 0<br>(schwächste) | T kann Änderungen von anderen T lesen, sobald sie gemacht worden sind. T weiss allerdings nicht, ob Änderungen Bestand haben werden.<br>Oder:<br>bedeutet, dass die entsprechende Transaktion Daten, die von einer anderen noch im Gang befindlichen Transaktion geschrieben wurden, lesen kann (“ dirty read”). Die Transaktion hält keine Sperren. Diese Isolationsstufe ist nur für READ ONLY Transaktionen gestattet.<br>Oder:<br>Bei dieser Transaktionsisolationsebene findet praktisch keine Isolation statt. Änderungen aus anderen Transaktionen sind sofort sichtbar, auch wenn diese noch nicht committed sind. |

**Client/Server Applikationen:**



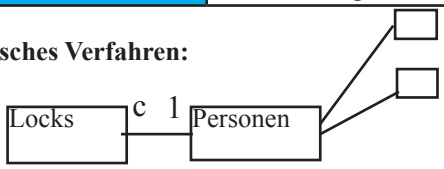

Modell Viewer Controller (MVW)

**Hauptthemen:**
-Applikatorische Sperrverfahren (Transaktionen (ACID-Eigenschaften) nicht geeignet, da zu viele Locks und kein Schutz vor Deadlocks)
-Performance Optimierung / Durchsatz (bei mehreren Clients)
-Abgleich von Datenbeständen
-OO in Relationaler Datenstruktur

**Applikatorische Sperrverfahren:**
Motivation:
-lange Transaktionsdauer verhindern, Leerlauf von DB-Transaktionen verhindern
-DB Transaktionen sind kein effektiver Schutz (Deadlock, blocking)

| | |
|---|---|
| pessimistisches Verfahren | Konfliktvermeidung (Lösung Sperren). 1 Applikation kann schreiben, alle andern können lesen (evt. mit Warnung) |
| optimistisches Verfahren | Konflikterkennung beim Speichern. Konfliktbehebung in Applikation, nicht im DBMS. Ev. Änderungen verwerfen, Mergen) |

**pessimistisches Verfahren:**




Schreiblocks werden in einer Hilfstabelle geführt, in welcher der Primary Key des zu ändernden Datenbankfelds eingetragen wird. Zudem muss darin auch eine Sperrdauer eingetragen werden für den Fall, dass der Client die Verbindung un-freiwillig abbricht (z.B. Absturz). Die Dauer kann Client-seitig aktualisiert werden, solange der Lock gebraucht wird. Die Aktualisierung läuft in einem separatem Thread und darf nicht von der aktiven Userinteraktion abhängig sein. Stichwort Robustheit.

-Applikation öffnet eine Personendatenbank
-Lock beziehen
-Daten lesen + manipulieren
Locks Relation
PID ¦ Kennung der Applikation ¦ Sperrzeitpunkt
Mac Addresse
IP Nummer
User ID
-Sperrzeitpunkt bring Robustheit ins System
-Datensatz maximal x Sekunden sperren
-Heartbeat : Aktualisierung eines Sperrzeitpunktes
-Applikation schliesst den Personendatensatz
-1.Lock entfernen
-Einsatz verwenden, wenn Wahrscheinlichkeit eines Konfliktes gross und die Kosten des Datenverlustes hoch ist, falls Merging aufwendig un dteuer ist!

**optimistische Verfahren:**
Konflikterkennung Vergleich zwischen aktuellem Datenbestand und ursprünglich gelesenem. Bei Konflikt -> Mergeverfahren anbieten!
Listing 1: Pseudocode Save
Tabelle um eine Spalte RowVersion mit Ganzzahlwert erweitern, welcher bei jedem Update inkrementiert wird
HilfstabelleErweitert:
PID ¦ Client ¦ Uhrzeit/Dauer ¦ RowVersion
. . . . . .

Pseudocode Save2:
save2 ( )
Select from Personen
where PID = 4411
and RowVersion = RowVersion // Konflikterkennung
Hat man die Möglichkeit, Konflikte schnell/einfach/günstig zu beheben, so kann man das optimistische Sperrverfahren anwenden. Ist die Konfliktbehebung jedoch zeitaufwändig/kompliziert/ teuer so ist das pessimistische Sperrverfahren vorzuziehen.

Listing 1: Pseudocode Save:
```
save( )
Select from Personen
        where PID = 4411
        and alterName = name
        and . . .
. . .
if(result)
        update
else
        Konflikt
```

**Client / Server Applikationen:**

-Durchsatz auf dem DBMS verbessern
-möglichst wenig Aktionen (SQL Statements abwickeln)
-lese + schreibverhalten optimieren

Transaktionen genügen nicht für Client-Server, wiel:
-Transaktionen dürfen nicht so lange Daten sperren
-Transaktiontn bieten keinen Schutz vor Blockaden

**Leseverhalten „Lazy Load“ / „Load on Demand“:**

-Daten nicht im Konstruktor laden, sonder beim 1.Zugriff (erster Getter). Dazu ein Bool „dataLoades“ verwenden
->Kapselung aller Felder im „getter“

**Schreibverhalten „Lazy Save“ / „Dirty Managment“ -> nur geänderte Daten schreiben:**

Bei einem syve() nicht immer speichern, sonder nur, wenn der Datensatz „dirty“ ist. Dazu ein Bool „isDirty“ verwenden, der bei jedem setter() Aufruf auf „true“gesetzt wird.

**Beziehung zwischen Datensätzen (Fremdschlüssel):**

```
Person.loadDataFromDB(){
        //nur Daten aus Rel „Person“ laden
}
Person.getAllTelNr(){
        //loadDataFromDB() ist hier nicht nötig
        vTel = new Vector();
        //DB zugriff: SELECT * FROM TelNr WHERE pid = this.pid
        for each r in resultSet{
                vTel.add(new TelNr(r.values());
        }
}
Person.save(){
        for each t in vTel{
                t.save();
        }
}
```

**Transaktionskontroller:**

```
public void save(boolean doCommit){
        if(doCommit){
                //begin Transaction
        }
        //DB Aktion
        if(doCommit){
                //Commit / rollback
        }
}
```

**Lazy Load:**

```
Person ( int pid) {
        id = pid ;
}

String getName ( ) {
        if (!dataLoaded) {
                loadData ( ) ;
        }
        return name ;
}

(fuer jedes Property)
private void loadData ( ) {
        select from Personen where id =. . .
        name = . . . // f u e r a l l e P r o p e r t i e s
        dataLoaded = t r u e ;
}
```

**Lazy Save:**

```
void save( ){
        i f (!isDirty){
        return;
        }else{
        update Personen set. . . //fuer alle Properties
        isDirty=false;
        }
}
void setName (String newName) {//fuer alle Properties
        i f (name.equals( newName)){
        return;
        }
        name = newName ;
        isDirty=true;
}
```

**Erweiterungen für Insert, Delete, ...:**

Vereint „Lazy Load“ und „Lazy Save“ in einer State Machine:
Die States in einer DB-Applikation können wie im Diagramm dargestellt werden.

**Dirty:** Daten in Applikation und BD sind unterschiedlich
**Clean:** Daten der Applikation entsprechen den Daten in der DB.
**Hollow:** Daten auf der Applikation noch unvollständig geladen
**New:** Daten sind erst in der Applikation erstellt worden, aber noch nicht in DB existent.
**Deleted:** Daten sind auf der Applikation gelöscht worden, aber in der DB noch vorhanden.

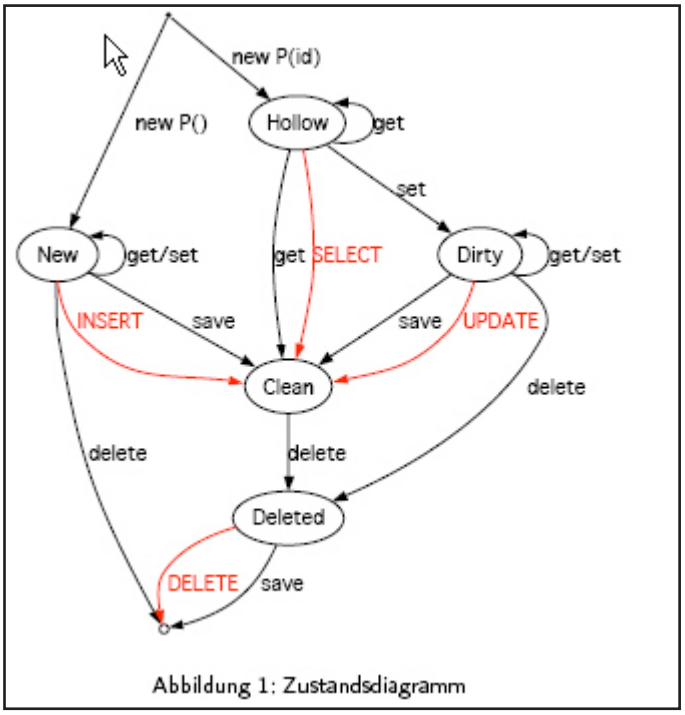


Abbildung 1: Zustandsdiagramm

**JDBC 2:**

API für Applikationsentwickler im Paket java.sql.*

**Treibertypen:**

-Java Native Driver für alle möglichen Datenbanken (mit proprietärem Treiber)
-Java Native Interface zur Anbindung von C-Programmen, etc.
-JDBC-ODBC-Bridge für Microsoft-Datenbankanbindung (langsamer als JND, nur als Übergangslösung gedacht)

**Mit JDBC können drei Dinge erledigt werden:**

1. Eine DB-Verbindung aufbauen oder auch auf Daten aus Tabellen in einzelnen Files zugreifen
2. SQL-Statements verschicken
3. Die Ergebnisse verarbeiten

**Architektur:**

**Relevant sind folgende Klassen und deren Verknüpfungen:**

**Driver** (ein Interface) wird speziell für das DBS geladen, zum Beispiel: Class.forName(„com.mysql.jdbc.Driver“).newInstance();. Der Driver wird vom DriverManager benötigt.

**DriverManager** kann angefragt werden, er soll eine Connection-instanz rausrücken (DriverManager.getConnection(url, user, passwd ) )

**Connection** liefert Statements und DatabaseMetaData

**Statement** wird zur Vorbereitung einer SQL-Abfrage angefordert. Hat zwei Ableitungen: PreparedStatement und davon wieder CallableStatement. PreparedStatement dient dazu, SQL-Abfragen vorzukompilieren und dann effizient auszuführen. CallableStatement wird benutzt, Stored Porcedures aufzurufen.

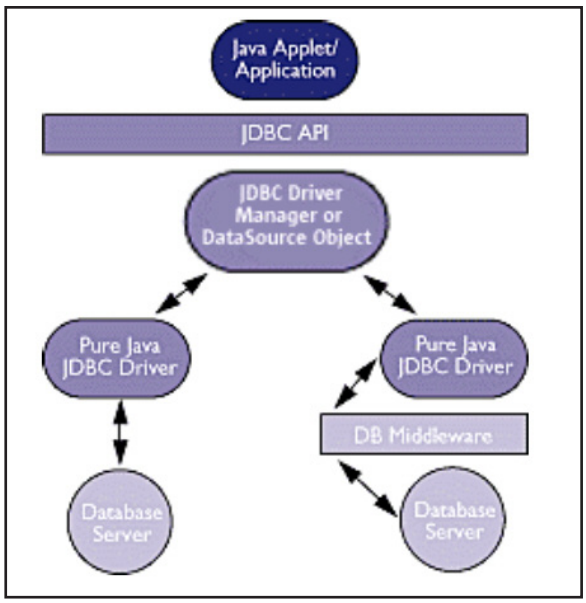
**Statement** beinhaltet die Ergebniswerte eines Statements.

**DatabasesMetaData** -> Rückgabewert von Connection.getMetaData(), beinhaltet Metadaten zur DB-Verbindung und zum DBMS

**ResultSetMetaData** -> Rückgabewert von ResultSet.getMetaData(), beinhaltet Metadaten zum ResultSet.

**Grundsätzlich findet in etwa folgender Ablauf statt:**

-Applikation erzeugt Connection-Instanz
-Applikation weist Connection-Instanz an, eine Statement-Instanz zu erzeugen
-Statement-Instanz wird mit SQL-Befehlen bestückt
-Statement-Instanz wird ausgeführt
- ResultSet-Instanz wird von Statement-Instanz angefordert
- Iteration über die ResultSet-Instanz
-Alle Instanzen wieder schliessen (ResultSet, Statement, Connection)

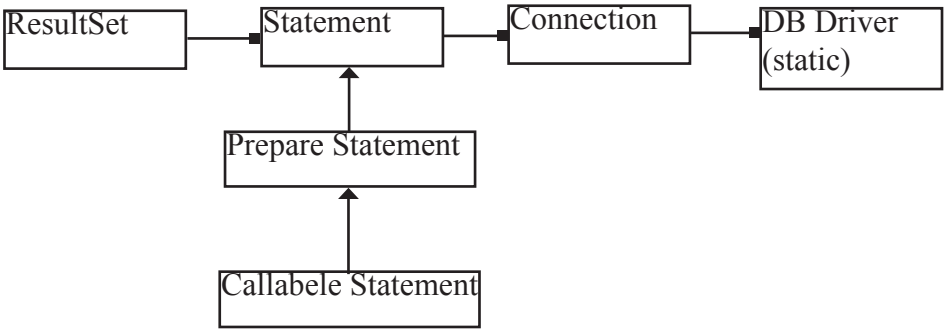



**Vorteile von JDBC:**

-Freie Wahl des DB-Systems durch Abstraktion
-Vereinfachte Entwicklung durch JDBC-API
-Vereinfachte Konfiguration in Netzwerk-Umgebungen

**Schlüsselfunktionen:**

-Voller Zugriff auf Metadaten
-Keine Installationsroutinen nötig
-DB-Verbindung über URL identifziert
-In Java enthalten



**LINQ:**

**Grösster Vorteil gegenüber anderen Datenbankschnittstellen liegt darin, dass „Strongtyping“ angewendet wrd. Die Prüfung zu kompilierzeit, ob das Statement korrekt ist (nicht erst zur Laufzeit).**

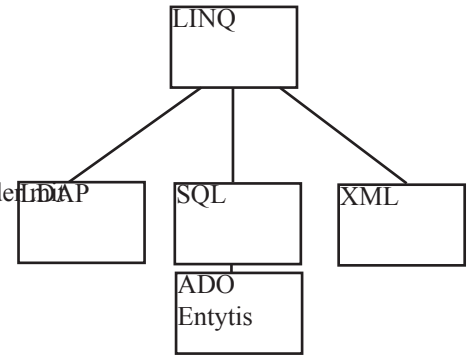
**Datenbanksysteme stellen zur Lösung dieses Problemes bereits die Lösung der „Stored Procedures„ zur Verfügung.**

**Framework für Abfragen von (tabellarischen) Daten**

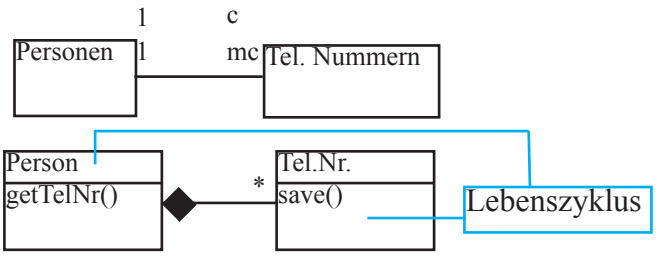
-deklerativ
-erweiterbar
-modular
-Sprachunabhängig
-Einheitliche Abfrage ->strong Typing, Typenkorrektheit zu compile Zeit prüfen, über die ganzen Schichten

**Methoden / Technologien:**

-Extension Methods (Klassen von aussen erweiterbar)
-funktionale Programmierung racers.from(...).where(...).orderBy(...).select(...)
-“Seiteneffektfreie Programmierung“
-racers.from(i)where().select()
-Lambda Expressions : verzögerte Ausführung von anonymen Methoden
-Expresson Trees : „Parse-Trees“, „AST“ (manipulierbarer Code)
-LINQ -> DB/SQL
    -Mapping Klassen müssen vorhanden sein (entweder von Hand schreiben, oder LINQ AQLMetal generieren lassen)
    -SQLMetal generiert aus DB Schema eine Klasse
-Syntactic Sugaring (Syntaktisches versüssen einer Programmiersprache)
    -z.B anonyme Klassen in Java, Generics, Nullable Typen



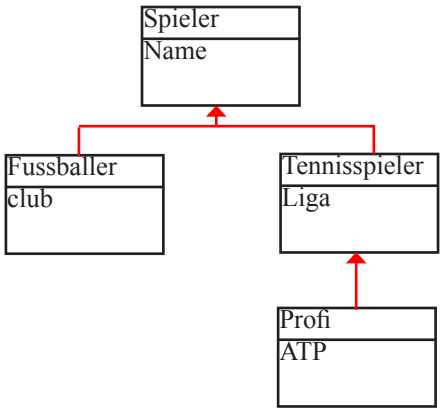
**Beziehungen (Fremdschlüssel):**



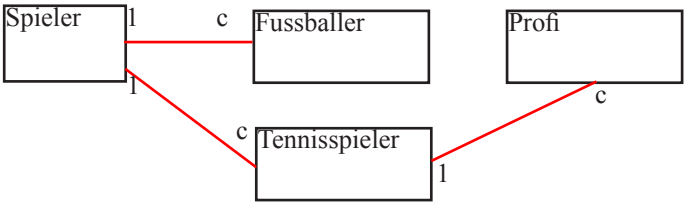

```
loadData(){
        //Nur Daten aus Rel. Person laden!
        //Nicht Tel. Nummern laden!
}
getTelNr(){
        //load Date Aufruf unnötig
        tnr = new Vector();
        //DB Zugriff SELECT * FROM Tel.Nummern WHERE PID = this.id;
        tnr.add(new Telnr());//alle Daten im Vector abfüllen
}
save(){
        //für jede abhängige Tabellen Vectoren
        foreach(v in tnr) v.save();
}
```

Empfehlung:
-1:mc / 1-c
- ◆—— eigene Klassen schreiben
Transaktionskontrolle:
entweder nur in Hautentität oder in intra Klassen
typische Lösung: save( bool docommit)

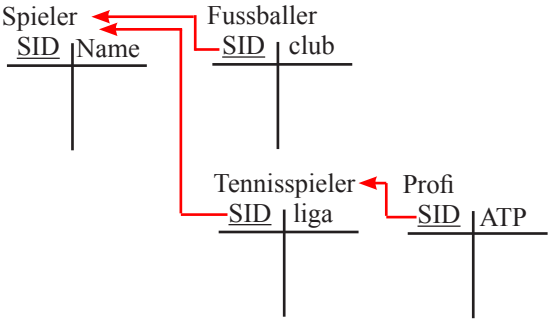
**Abbildung von Vererbungshierarchien:**




**ERM:**




**RM:**




+erweiterbar
+keine Redundanzen
-viele Joins, oder mehrere Selects hintereinaander
-ins / upd / del für jede Tabelle separat

**Welche Klasse muss instanziert werden?**

Beginnend bei den Detailtabellen zu den haupttabellen durcharbeiten, d.h. den Vererbungsbaum Postorder durchlaufen und jeweils nach der ID suchen.

**Alternative Lösung: alles in einer Tabelle:**


-denormalisiert -> Redundanzen!
+schnelle selects, inserts, updates, deletes
aus einer SID Klasse instanzieren:
a)zusätzliches Feld (in Spieler) (Klassenname -> Refelections)
b)Felder / Relationen analysieren
NULL / Datensätze von „unten nach oben“
b) nur Tabellen für alle konkreten Klassen
**Welche Klasse muss instanziert werden?**
zusätzliches Feld mit vollqualifiziertem Klassennamen fahren (und dann mit Refelctions arbeiten)

**OO-RDBMS Mapper:**
(Java Serialisierung)
-Nur Single User Betrieb
-File basiert
JDBC Basisbibliothek
-Bedeutet: eine eigene Architektur hinterlegen
OO-RDBM-Mapper
-Java EE
-JDO(Java Data Objects) (veraltet)
JPA(Java Persitancy API) (löst JDO ab)
-Hibernat, Torque, ...
**Kriterien für die Wahl eines OO-RDBMS-Mappers:**
-OO Klassenmodelle in ein Relationenmodell abbilden
-Vererbung
-Assoziationen / Aggregationen
-RM nach OO abbilden
-Schlüsselgenerierung (Object-ID, über Prozesse hinweg stabil)
-Eingebaute Abfragesprachen (SQL)
-Umgang mit Transaktionen
-Integration in andere Frameworks

**Hibernate:**
Hibernate ist ein Java-basierter Persistenzdienst, der Relationen auf Objekte abbildet.
Der Entwickler kümmert sich nicht um SQL-Routinen. SQL-Ausdrücke werden für die entsprechend konfigurierte DB von Hibernate normalisiert und abstrahiert. Zusätzlich sorgt Hibernates Reflexionsmechanismus dafür, dass für die Persistenz relevante Aspekte zur Laufzeit generiert werden. Die Verbindung zum DB-System kann dabei zum Beispiel über JDBC erfolgen.
**Ablauf:**
-Objektrelationale XML-Darstellung des Domänenmodells
-Konfiguration von Hibernate (DB-Treiber, Verbindungsdaten etc.)
-Generierung von Persistenz-Klassen
-Datenbank-Schema exportieren
-Programmieren in der Hibernate-API
**Reverse-Engineering:**
Alte Datenbestände können mit Hilfe von externen Tools einer Persistenzschicht zugänglich gemacht werden.
Auch existierende Persistenzklassen können Hibernate nachträglich hinzugefügt werden.

**Schlüsselgenerierung:**
I=select max(id) + 1
Extra Tabelle erstellen mit den aktuellen Maximas aller Tabellen