Relationale Datenbanken

© Dr. Arno Schmidhauser
Letzte Revision: Dezember 2006
Email: arno.schmidhauser@sws.bfh.ch
Webseite: http://www.sws.bfh.ch/db

Inhalt

Einleitung

Literatur

[1] "Einstieg in SQL"; Markus Troll, Oliver Bartosch; Galileo Computing, 2004.

[2]"Relationale Datenbanken und SQL"; Günter Matthiessen, Michael Unterstein; Addison-Wesley, 3. Auflage, 2003.

[3] "Database Design for Smarties"; R. J. Muller; Morgan Kaufmann, 1999.

[4] "Datenbanksysteme, Konzepte und Techniken der Implementation"; T. Härder, E. Rahm; Springer 1999.

[5] "Database Systems"; Paolo Atzeni et. al; McGraw-Hill, 2000.

[6] "SQL-3 Complete, Really"; Peter Gulutzan, Trudy Pelzer; Miller Freeman, 1999.

[7] "SQL Performance Tuning"; Peter Gulutzan, Trudy Pelzer; Addison-Wesley, 2003.

[8] "A Guide to the SQL Standard"; C.J. Date; Addison-Wesley 2000.

[9] "Database Administration"; Craig S. Mullins; Addison-Wesley, 2002.

Zu [6] gibt es ein *vollständig SQL-99 konformes* Datenbanksystem namens OCELOT (www.ocelot.ca). Es eignet sich sehr gut für das Studium von SQL als Datenbanksprache.

Die Websites der verschiedenen DB-Hersteller können sehr informativ sein und bieten besonders bezüglich technologischer Fragen oft gute Hilfe.

Die Originalstandards zu SQL erhält man von ANSI:

http://www.ansi.org

Ziele des Kurses

Jeder Teilnehmer ...

- kennt das Relationenmodell.
- kennt die Sprache SQL und kann sie interaktiv oder in Applikationen eingebettet anwenden.
- kann aus einer praktischen Problemstellung ein Datenmodell in UML ableiten.
- versteht den Begriff der Transaktion und kann ihn anwenden.
- hat eine Vorstellung über die innere Organisation eines Datenbanksystems.

Eigenschaften einer Datenbank

- Verwaltung kleiner bis grösster Datenbestände
- Einfacher, standardisierter Zugriff auf Daten mit SQL
- Weit verbreitete Programmier API's (ODBC, JDBC)
- ausgefeilte und hocheffiziente Zugriffsmechanismen
- Wiederherstellung nach Server- oder Client-Crash
- Integritätsregeln stellen Korrektheit der Daten sicher
- Replikations- und Verteilungsmechanismen
- Zugriffskoordination
- Benutzerauthorisierung

Aufgeführt sind hier hauptsächlich *technologische* Gründe für den Einsatz eines Datenbanksystems. Daneben - oder sogar schwergewichtig - gibt es auch organisatorische und modellorientierte Gründe. Für relationale Datenbanken existiert beispielsweise eine enorme Vielfalt an Drittprodukten, welche die Implementation einer Anwendung unterstützen. Auch die Werkzeuge für das Tuning des Zugriffs und das Monitoring von zugreifenden Applikationen sind von enormer Bedeutung beim Betrieb.

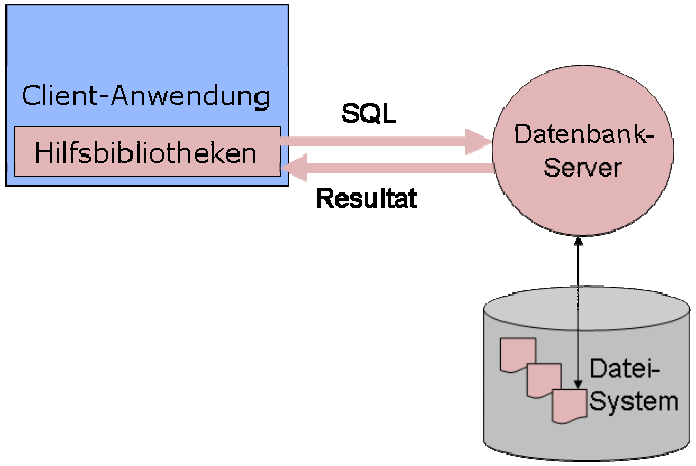
Die Relationale Datenbank ist eines der erfolgreichsten Konzepte in der Informatik. Auf der Basis eines einfachen und sauberen theoretischen Modells aus den 70er Jahren, des Relationenmodelles und der Relationenalgebra, wurde die Sprache SQL entwickelt (Strucured Query Language). SQL dient sowohl der Strukturdefinition einer Datenbank (hauptsächlich in Form von Tabellen) wie auch der Abfrage und Manipulation der Daten darin. SQL definiert damit faktisch die äussere Sicht auf die Datenbank und die funktionalen Möglichkeiten für den Entwickler oder den Benützer. SQL hat seit seiner Entstehung in den 80er Jahre drei grössere Standardisierungen durch ANSI und ISO durchlaufen (SQL-1 von 1989, SQL-2 von 1992, SQL-3 von 1999) und ist weitestgehend von der Software-Industrie anerkannt. Die Produkthersteller bemühen sich klar um eine Annäherung an den Standard.

Relationale Datenbank beherrschen gegenüber anderen Datenbankmodellen volumenmässig den Markt. Die 5 grössten Produkte sind Oracle, DB2, SQL Server, Informix und Sybase. Wichtig ist auch Interbase (Borland) und MySQL (sehr populäres Open Source Produkt). Eine Vielzahl weiterer Systeme mit speziellen Eigenschaften stehen zur Verfügung. Beispielsweise reine Java-Datenbanken wie PointBase.

Die relationalen Systeme werden in verschiedene Richtungen weiterentwickelt. Beispielsweise wird versucht, mit dem Einbau von Java als Programmiersprache und Java-Klassen als Datentypen eine Vereinheitlichung zwischen dem Typsystem einer populären Programmiersprache und demjenigen von SQL zu erreichen. Andere Hersteller haben sich um sprachneutrale, objektorientierte Erweiterungen bemüht, z.B. Oracle. Man spricht von objektrelationalen Datenbanken.

Neben der Familie der relationalen Datenbanken und ihrer Ausläufer gibt es auch rein objektorientierte Datenbanken und XML-basierte Produkte.

Client-Server Architektur



Eine Datenbank ist ein Server-Programm, welches von seinen Client-Anwendungen SQL-Befehle entgegennimmt. Der Datenbank-Server läuft meist auf einem dafür vorgesehen Rechner. Das Server-Programm kontrolliert und verwaltet die zu ihm gehörigen Daten*banken*.

Die Kommunikation zwischen Client und Server findet meist über eine TCP/IP Verbindung statt, über welche die Datenbank SQL-Befehle entgegennimmt und Abfrageresultate ausliefert.

Der Server überwacht die Verbindung zu jedem Client und kann bei dessen Absturz einen Rollback unfertiger Arbeiten (Transaktionen) durchführen.

Der Server steuert den Zugriff mehrerer Clients auf dieselben Daten, indem er während einer SQL-Operation die betroffenen Daten zuhanden *eines* Client sperrt.

Programm-Beispiel

```
...
try {
  Connection con = DriverManager.getConnection( ... );
  Statement stmt = con.createStatement();
  String q1 = "SELECT saldo FROM Konto WHERE idKonto = 4711";
  ResultSet rs = stmt.executeQuery( q1 );
  if ( rs.next()) {
    kontostand = rs.getInt( "saldo" );
    kontostand = kontostand – 1000000;
  }
  String q2 = "UPDATE Konto SET saldo = ? WHERE idKonto = 4711";
  PreparedStatment pstmt = con.prepareStatement( q2 )";
  pstmt.setInt( 1, kontostand );
  pstmt.executeUpdate();
  con.commit();
}
catch ( SQLException e ) { con.rollback(); }
...
```

Gegeben sei folgende SQL-Tabelle

```
create table Konto (
   idKonto varchar(32),
     saldo numeric (10,2),
     primary key ( idKonto ),
     check ( saldo > 0 ) initially deferred
)
```

Das Durchführen des *select-Befehls* beinhaltet für die Datenbank folgende Aufgaben:

1. Transaktion öffnen ('Begin Transaction'-Eintrag im Logfile)
2. Syntax parsen
3. Ausführungsrecht für 'select' auf die Tabelle 'Konto' prüfen
4. Ausführungsplan (Algorithmus) für die Abfrage ermitteln
5. Lese-Sperren auf die gelesenen Datenelemente setzen
6. Daten ausliefern

Das Durchführen des *update-Befehls* beinhaltet für die Datenbank folgende Aufgaben:

1. Syntax parsen
2. Ausführungsrecht für 'update' auf die Tabelle 'Konto' prüfen
3. Schreibsperre auf die zu ändernden Datenelemente setzen (genauer: Lesesperre in Schreibsperre umwandeln)
4. Alte und neue Datenwerte im Logfile protokollieren
5. Änderung durchführen

Das Durchführen des *commit-Befehls* beinhaltet für die Datenbank folgende Aufgaben:

1. check-Bedingung prüfen, im Fehlerfall Rollback durchführen
2. 'End Transaction'-Eintrag im Logfile festhalten
3. Sperren und damit die Daten für andere Benutzer freigeben

Datenmodellierung

- 3-stufige Modellierung
- Konzeptionelles Modell

Das Relationmodell ist ein *formales* Datenmodell.

3-stufige Modellierung

Konzeptionelles Modell

Benötigte Informationen und Zusammenhänge darstellen. Wichtig: Übersichtlichkeit, informell. Werkzeug: Grafische Notation nach UML

Formales Modell

Festlegung eines bestimmten Datenbanktyps (Relationale DB) und Befriedigung von dessen Ansprüchen (Normalisierung).
Darstellung mit UML oder tabellarisch.

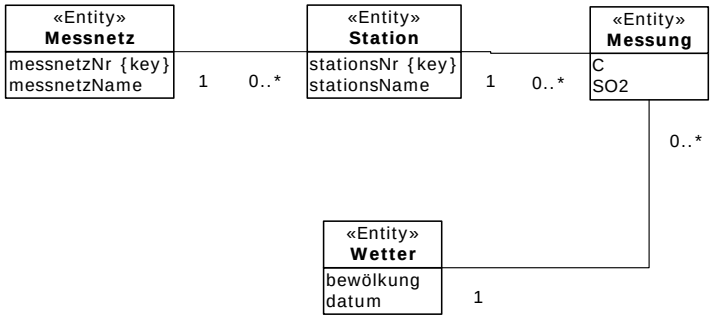
Physisches Modell

Implementation des Datenmodelles (Tabellen mit SQL-Befehlen erzeugen, Hilfsstrukturen aufbauen)

Das Formale Modell heisst auch Logisches Modell

Konzeptionelles Modell mit UML

- Das *konzeptionelle* Modell dient der Darstellung
  - der benötigten Information
  - von Zusammenhängen
- Beispiel: Datenmodell zur Erfassung von Luftschadstoffen an verschiedenen Standorten.



Vorgaben für dieses Datenmodell:

1. Messungen und Messstationen sind die zentralen und wichtigen Elemente für dieses Datenmodell.
2. Jede Messung ist einer Station zugeordnet, resp. wurde dort durchgeführt.
3. Jede Messung bezieht sich auf bestimmte Wetterangaben.
4. Jede Station ist einem bestimmten Messnetz (Beispielsweise das Netz des Kantons, das Netz der Uni, das Netz des VCS usw.) zugeordnet.

Bemerkungen:

1. Im Rahmen von UML kann die Eindeutigkeit eines Attributes ganz allgemein als Zusicherung ausgedrückt werden. Zusicherungen werden in geschweiften Klammern dargestellt, hier mit {key}.
2. Im Rahmen des UML-Modelles ist nicht spezifiziert, wie Assoziationen implementiert werden. Bei einer Programmiersprache würde man mit Referenzen oder Pointern arbeiten, bei relationalen Datenbanken mit Fremdschlüsseln welche sich auf Primärschlüssel beziehen.

UML-Elemente für das konzeptionelle Datenmodell

- Für die *Daten*modellierung sind folgende Elemente wichtig:
  - Klasse: Definiert Name und Attribute von physischen oder konzeptionellen Dingen.
  - Assoziation: Statuiert einen Zusammenhang (eine Abhängigkeit) zwischen zwei Klassen.
  - Multiplizität: Definiert, wieviele Objekte der einen Klasse mit wievielen Objekten der anderen Klasse in Zusammenhang stehen.

Im Kurs über UML wurde ebenfalls bereits von Klassen und einem Klassenmodell gesprochen. Im Rahmen der *Daten*modellierung interessieren ausschliesslich die sog. Entity-Klassen, d.h. solche, die langfristige Informationen tragen. Im Rahmen der Applikationentwicklung kommen auch noch viele Klassen zum Zug, die Funktionalität tragen, beispielsweise um bestimmte Informationen für eine Darstellung im Browser aufzubereiten, über ein Netzwerk zu transportieren usw.

Prinzipien für das konzeptionelle Modell

1. Eine Klasse beschreibt ein einzelnes Objekt
2. Bezeichnungen sind anwendungs-, nicht software-bezogen
3. Eine Klasse hat mehr als ein darstellungswürdiges Attribut
4. Attribute einer Klasse haben denselben Lebenszyklus
5. Eine Klasse sammelt eng zusammengehörende Attribute
6. Jedes Attribut hat nur eine Bedeutung

1. Eine Klasse muss einen Namen haben, der ein *einzelnes Objekt* der Klasse beschreibt. In der Regel ist das ein Substantiv in Einzahl. Die Klasse darf nicht die *Menge* aller Objekte bezeichnen. Substantive dürfen *nicht in Mehrzahl* geschrieben sein. Wenn die Menge der Objekte *als Ganzes* wichtig ist, sollte dafür eine eigene Klasse definiert werden (Beispiel: Klasse 'Katalog' *und* Klasse 'Buch', statt nur Tabelle 'Buchkatalog' mit Buch-Datensätzen darin). Eine guter Ansatz für die Syntax von Namen ist Java: Klassennamen mit Grossbuchstabe, Attributnamen mit Kleinbuchstaben beginnen. Einzelne Worte in Wortkombinationen mit Grossbuchstaben abtrennen (Beispiel Klasse 'KursUmfrage' mit Attribut 'startDatum').
2. Namen von Klassen und Attributen müssen *für die Anwendung* der Software von Bedeutung sein und sollten keine software-technischen Ergänzungen haben (Gutes Beispiel: Bücherkatalog, schlechtes Beispiel: BuchCollection)
3. Eine Klasse sollte *mehr als ein* Attribut haben. Wenn nicht, kann das Attribut wahrscheinlich einer anderen Klasse zugeordnet werden. Ausnahme: Wenn zu erwarten ist, dass die Anzahl Attribute in nächster Zeit zunimmt.
4. Die Attribute einer Klasse sollen *denselben Lebenszyklus* wie die ganzen Objekte der Klasse haben. Attribute einer Klasse, die zeitweilig gar keinen Sinn machen, sind anderen Klassen zuzuordnen (Beispiel: Attribut 'Diagnose' für ein Patienten-Objekt ist schlecht, weil der Patient in der Patientenverwaltungs-Software wahrscheinlich mehrere Jahre lebt, die meiste Zeit aber gar nicht krank ist. Zwei Klassen erstellen: 'Patient' und 'Fall').
5. Nur *eng zusammengehörende Attribute* in einer Klasse sammeln (Schlechtes Beispiel: 'EntleiherName' in der Klasse 'Buch' in einer Bibliothekssoftware).
6. Jedem Attribut nur *eine Bedeutung* geben. Beispiel: Die Klasse 'Artikel' habe das Attribut 'Lagerbestand'. Eine Zahl zwischen 0 und ∞ bedeute den tatsächlichen Lagerbestand, die Zahl -1, dass der Artikel nicht mehr verfügbar ist. Die Software zur Ermittlung der gesamten Anzahl Artikel in einem Lager muss nun mit schwerfälligen Konstrukten immer zwei Fälle unterscheiden. Besser: Neues Attribut 'Status' für Klasse 'Artikel einführen.

Das Relationenmodell

- Was ist eine Relation
- Normalisierung

Das Relationmodell ist ein *formales* Datenmodell.

Woher kommt der Begriff Relation?

- Mathematische Basis: Mengenlehre.
- **Domäne** D = Wertemenge [~Datentyp]
- **Relationenschema** = Menge von Attributen A_i mit zugehörigen Domänen D_i [~Tabellendefinition]
- **Tupel** = Wertekombination aus D_1 x D_2 x D_3.... D_i [~Datensatz]
- **Relation** = *Menge* von Tupeln aus einem Relationenschema [~Tabelleninhalt]
- **Relationenmodell** = Menge *aller* Relationenschemas [~Datenmodell]
- Die **Relationenalgebra** definiert Operationen auf Relationen wie Projektion, Selektion, Vereinigung, Differenz, Durchschnitt, Verbund und Produkt [~SQL]

Das Hauptstrukturelement einer relationalen Datenbank ist die Relation. In der Praxis des Entwicklers kann man die Begriffe Relation und Tabelle als identisch behandeln.

SQL ist *eine,* wenn auch die wichtigste Implementation des Relationenmodelles und der Relationenalgebra.

In SQL-Datenbanken sind sowohl die eigentlichen Daten, wie auch Hilfs- und Metadaten (z.B. Tabelle aller Benutzer, Tabelle aller Attributnamen, Tabellen aller Datentypen, Tabelle aller Tabellen usw.) in Tabellen organisiert.

Letzlich hat sich die Einfachheit von Tabellen als Vorteil für das Aufkommen und die Anwendung der relationalen Datenbanken erwiesen. Die Sprache SQL ist damit relativ einfach und überschaubar geblieben.

Die Relationenalgebra und ihre Operation haben folgende Eigenschaften

deskriptiv
Alle Operationen sind mengen- und bedingungsorientiert. Es gibt keine *Ablauf*konstrukte, wie Verzweigungen und Schlaufen.

abgeschlossen
Jede Operation ergibt wieder eine Relation.

optimierbar
Durch algebraische Umformung eines Ausdrucks mit Operationen können ausführungsmässig performantere Ausdrücke erzeugt werden. Beispielsweise ist der Ausdruck (R1 $\cup$ R2) $\cap$ R3 unter Umständen besser bearbeitbar durch den äquivalenten Ausdruck (R1 $\cap$ R3) $\cup$ (R2 $\cap$ R3)

effizient

Die Komplexität einer Operation wächst höchstens proportional zum Produkt der Anzahl Tupel in den beteiligten Relationen.

sicher
Jede Operation ist von endlicher Dauer, wenn der Datenbestand endlich ist.

orthogonal

Alle Operationen können frei und ohne Einschränkungen miteinander kombiniert werden. Die Orthogonalität ist bei SQL nicht vollumfänglich gewährleistet, weil etwa verschachtelte Abfragen (select ... from (select ... from)) nicht möglich sind.

Eigenschaften einer relationalen Datenbank

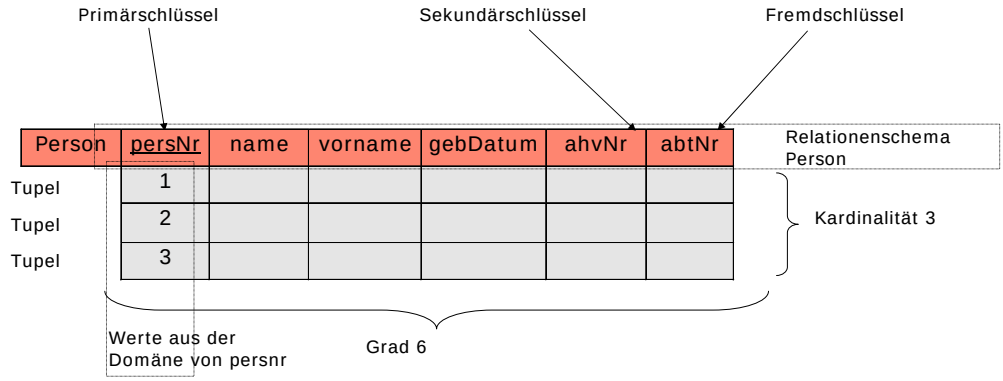
- Die Domäne [Datentyp] jedes Attributes [Feldes] ist **atomar**.
- Elemente [Datensätze] der Relation müssen **unterscheidbar**, das heisst eindeutig **identifizierbar** sein.
- Relationen [Tabellen] sind Mengen, die Elemente daher **nicht geordnet**.

Die Bedingung, dass die Domäne eines Attributes *atomar* sein muss, ist ein entscheidendes Charakteristikum für relationale Datenbanken. Sie bedeutet, dass einzelne Attributwerte in sich nicht wieder ein Tupel oder eine Menge von Tupeln sein dürfen. Dies steht ganz im Gegensatz zur objektorientierten Programmierung, wo Attribute (Member) von Objekten selbst wieder einfache oder komplexe Objekte, Listen, Mengen oder Arrays von Objekten sein dürfen. In der objektorientierten Programmierung können Objekte also eine umfangreiche und komplexe Substruktur haben. In einer relationalen Datenbank ist das nicht möglich. Dies bewirkt einerseits eine sehr *einfache Struktur der einzelnen Relationen* (Tabellen), führt andererseits aber zu einer *grossen Anzahl von Relationen*, welche miteinander in Beziehung stehen.

Eine Relation rsp. eine Tabelle stellt im mathematischen Sinn immer eine *Menge* dar. Die Elemente einer Menge müssen unterscheidbar sein. Dies führt zur Notwendigkeit von Relationen- oder Primärschlüsseln. Enthält eine Tabelle zwei völlig identische Datensätze, repräsentieren sie dasselbe Element und können über SQL-Befehle nicht mehr unterschiedlich behandelt werden.

Die Elemente einer Menge haben nicht von sich aus eine Ordnung, d.h. es gibt keine vorbestimmte Reihenfolge der Elemente. Das Datenbanksystem verwaltet Einträge in einer Tabelle ohne Garantie für eine bestimmte Reihenfolge. Diese kann allenfalls durch Sortieren bei der Ausgabe realisiert werden. Dazu muss aber ein geeignetes Attribut vorhanden sein, oder ein solches *geschaffen* werden. Beispiel: Die Vornamen einer Person haben im Pass eine definierte Reihenfolge, nämlich die, wie die Namen geschrieben stehen. Die Reihenfolge ist nicht alfabetisch. Um die Reihenfolge in einer Datenbank-Tabelle sicherzustellen, muss eine explizite Nummerierung vorgenommen werden, zum Beispiel mit einem Attribut 'VornameNummer'.

Relation, Begriffe



Relationenschlüssel: persNr, name+vorname+gebDatum, ahvNr

Gewählter Primärschlüssel: persNr

Relationenschema
Menge der Attribute einer Relation, "Relation ohne Inhalt".

Relation
Menge von Tupeln. Die Komponenten der Tupel sind definiert durch das *Relationenschema*. Jede Komponente eines Tupels ist ein Wert aus der Domäne des entsprechenden Attributes. Weil Relationen Mengen sind, ist es per Definition nicht möglich, dass zwei identische Tupel vorkommen. Jedes Element kann in einer Menge nur einmal vorkommen. Um die Einzigartigkeit jedes Tupels sicherzustellen, definiert man Relationenschlüsse und daraus wiederum einen speziellen, den Primärschlüssel. Eine Menge ist nicht geordnet und folglich liegen die Tupel einer Tabelle nicht in einer bestimmten Reihenfolge vor. Anwendungsprogramme dürfen nicht von einer bestimmten Reihenfolge der Tupel ausgehen.

Attribut
Name einer Eigenschaft, für die jedem Element der Relation ein Wert zugeordnet werden kann. Jedes Attribut hat eine zugeordnete Wertemenge (Domäne).

Relationenschlüssel
Eine minimale Menge von Attributen, deren Werte jedes Tupel in der Relation eindeutig identifizieren, heisst Relationenschlüssel. Minimal heisst: Wenn man eines der Attribute des Relationenschlüssels entfernt, bilden die übrigen keinen Schlüssel mehr. Es kann mehrere Relationenschlüssel geben, aber immer gibt es mindestens einen (sonst liegt keine Relation vor), nämlich die Menge aller Attribute der Relation. Ein Relationenschlüssel heisst manchmal auch *Candidate Key*.
Ob ein bestimmtes Attribut oder eine Attributkombination einen Relationenschlüssel darstellt ist nicht naturgegeben, sondern eine Definitionsfrage. Eine Telefonnummer könnte beispielsweise je nach Annahmen ein Personelement eindeutig identifizieren oder aber von mehreren Personen benutzt werden.

Primärschlüssel

Die Wahl des Primärschlüssels ist ein Designentscheid. Aus den Relationenschlüsseln wird einer ausgewählt und als Primärschlüssel bezeichnet. Die übrigen Relationenschlüssel sind dann *Sekundärschlüssel*. Sehr häufig wird die Relation um ein künstliches Schlüsselattribut ergänzt und dieses zum Primärschlüssel erklärt. Der Primärschlüssel sollte aus möglichst wenigen Attributen mit konstanten Werten zusammengesetzt sein. In der Regel bervorzugt man einzelne Attribute. Primärschlüssel müssen zu jeder Zeit definiert sein. Sie dürfen keine Nullwerte (siehe unten) enthalten.
Schlüssel definieren in keiner Weise eine Speicherreihenfolge oder eine bestimmte Zugriffsmethode auf die Daten. Dies ist Sache der Systemoptimierung der Datenbank bei der Durchführung einer Abfrage.
Ein Primärschlüssel kann grundsätzlich ein *einzelnes* Attribut oder eine *Kombinationen* von Attributen sein. In der Regel bervorzugt man einzelne Attribute.

Fremdschlüssel

Der Fremdschlüssel identifiziert nicht ein Tupel *innerhalb* der Relation, wie der Primär- oder ein Sekundärschlüssel. Der Fremdschlüssel bezieht sich auf den Wert eines Primärschlüssel in einer *anderen* Relation. Ein Fremdschlüssel setzt somit Tupel (Datensätze) aus zwei verschiedenen Relationen in Beziehung miteinander. Das Pendant zum Fremdschlüssel in der Programmierung ist der Pointer.

Domäne

Menge der möglichen Werte für ein Attribut, auch Wertemenge genannt. Mehrere Attribute können diesselbe Domäne haben. Domänen sind atomar, das heisst, einzelne Attributwerte dürfen in sich nicht wieder ein Tupel oder eine Menge sein. Dies steht ganz im Gegensatz zur objektorientierten Programmierung, wo Attribute (Member) von Klassen selbst wieder einfache oder komplexe Objekte, Listen, Mengen oder Arrays von Objekten sein dürfen. In der objektorientierten Programmierung können Objekte also eine umfangreiche und komplexe Substruktur haben. In einer relationalen Datenbank ist das nicht möglich. Dies bewirkt einerseits eine sehr *einfache Struktur der einzelnen Relationen* (Tabellen), führt andererseits aber zu einer *grossen Anzahl von Relationen*, welche miteinander in Beziehung stehen.
Anmerkung: Es ist sehr gefährlich, eine Pseudostrukturierung vorzunehmen, z.B. ein Attribut 'adresse' als komma-separierte Liste von 'strasse', 'plz' und 'ort' in einer Relation zu definieren. Die Abfrage- und Verwaltungs-möglichkeiten der Sprache SQL versagen meist ihre Dienste in solchen Fällen. Definitiv schlecht wird die Performance, wenn in pseudostrukturierten Feldern nach Information gesucht werden muss.
In der Praxis gibt es zwei Arten Domänen für ein Attribut. Einerseits kann ein vorgegebener Basistyp (Zahl, String, Datum, Zeit) gewählt werden, andererseits kann ein Basistyp genommen werden, und dieser mit zusätzlichen Bedingungen eingeschränkt werden. Beispielsweise kann eine Zahl auf einen Wert zwischen 1 und 10 eingeschränkt werden, oder ein Ortsname (String) muss in einer Liste vorgegebener Namen vorkommen.

Relationenmodell

Das Relationenmodell im Sinne aller verwendeten Relationenschemas heisst häufig auch Datenbankschema. Ein Datenbankschema ist häufig gleichbedeutend mit einer effektiv erzeugten Daten*bank* (create database-Befehl) oder allen Tabellen eines bestimmten Benutzers.

Datenbank-Slang

Relation = Tabelle = Entität
Relationenschema = Tabellendefinition
Attribut = Spalte = Feld
Tupel =Zeile = Record = Datensatz
Domäne = Wertemenge = Feldtyp
Sehr verpönt, weil Missverständnisse entstehen können: Datenbank = Tabelle

Ableitung des Relationenmodelles

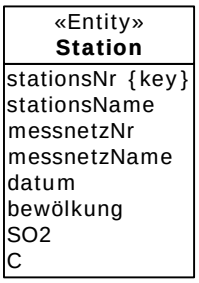
- Das Relationenmodell kann aus dem konzeptionellen Modell wie folgt abgeleitet werden:

1. Für jede UML-Klasse *eine* Tabelle erstellen, *Primärschlüssel* und *Fremdschlüssel* festlegen/wählen.
2. Tabellen mit *atomaren* Domänen erstellen (Erste Normalform).
3. Vermeidung unterwünschter Abhängigkeiten durch Normalisieren in die *zweite* und *dritte* Normalform.

Normalisierung

Ableitung des Relationenmodelles, Ausgangspunkt

- Für das folgende Beispiel soll von einem ganz einfachen, konzeptionellen UML-Modell für das Schadstoff-Messystem ausgegangen werden, um den Ableitungsprozess zu illustrieren

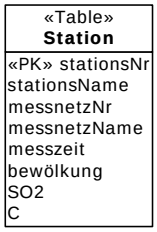


Dies ist ein reines Schulbuchbeispiel. Ein geübter Designer wird sicherlich keine so mit Information überladene Klasse definieren.

Eine Tabelle pro UML-Klasse ...

Primärschlüssel

Station	stations Nr	stations Name	messnetz Nr	messnetz Name	datum	bewölkung	°C	SO_2
	1	Aare	1	Uni	1.6.01	stark	21.3	10
					2.6.01	mittel	18.2	12
					3.6.01	schwach	12.8	14
	2	Rathaus	2	Kanton	1.6.01	stark	21.8	21
					2.6.01	mittel	18.9	20
					3.6.01	schwach	13.8	22
	3	Loeb	2	Kanton	1.6.01	stark	21.0	23
					2.6.01	mittel	17.2	26
					3.6.01	schwach	14.8	28



Bemerkungen:

- Die Primärschlüssel wurden unterstrichen. In der tabellarischen Darstellung ist dies meistens so anzutreffen. In der klassenorientierten Darstellung nach UML werden die Primärschlüssel meist mit «PK» gekennzeichnet.

Folgende Annahmen gelten hier und für die folgende Normalisierung:

- pro Tag (oder pro 'datum') liegt ein Wetterbericht mit Bewölkungsangabe vor.
- Temperatur und SO_2-Konzentration werden pro Station und Tag einmal erfasst.
- Das Wetter ('bewölkung') ist für alle Stationen dasselbe, es wird durch einen zentralen meteorlogischen Dienst zur Verfügung gestellt.

Probleme / Unschönheiten:

- Das Einfügen eines neuen Messnetzes ohne mindestens eine Station ist nicht möglich.
- Sehr viele Daten sind mehrfach erwähnt, z.B. der Zusammenhang zwischen Datum und Bewölkung, zwischen Messnetznummer und Messnetzname etc.
- Das Löschen der Station 1 ist nicht möglich, wenn man die Information über das Messnetz behalten will.

Normalisierung

- Vermeidet ...
  - *Redundanzen* und damit potentielle *Widersprüche.*
  - unerwünschte *Seiteneffekte (Anomalien)* beim Einfügen, Ändern oder Löschen von Daten.
- Ermöglicht ...
  - das "one fact in one place"-Prinzip.
  - ein Datenmodell, das von jedem Entwickler verstanden wird und interpretierbar ist.
  - dass alle vorhandenen Informationen auch tatsächlich abgefragt werden können.

Ein sauberes UML-Modell, korrekt in Tabellen überführt, ergibt meistens ein weitgehend normalisiertes Datenmodell. Die eigentliche Schwierigkeit liegt auch nicht im *Durchführen* der Normalisierung, sondern im *Erkennen* und Definieren der ihr zugrundeliegenden Abhängigkeiten zwischen Attributen ("Gehört ein Mitarbeiter zu einer oder zu mehreren Abteilungen oder manchmal auch zu keiner?"). Die meisten entsprechenden Überlegungen werden aber direkt oder indirekt beim Erstellen des UML-Modelles angestellt, weshalb die Normalisierung vorallem auch ein Werkzeug zur *Verifikation* des Datenmodells ist.

Auch wenn der Ausgangspunkt eines Datenmodells beispielsweise ein bestehendes, unnormalisiertes Papierformular oder Excel-Sheet ist, welches bereits eine *tabellen*artige Struktur aufweist, geht ein sinnvoller Entwurfsweg zuerst über ein UML-Modell und danach über eine Normalisierung der daraus gewonnen Tabellen.

Die Normalformen

0NF Jede Tabelle hat einen Primärschlüssel.

1NF Die Domänen aller Attribute sind atomar.

2NF Bei Tabellen mit mehreren Primärschlüsselattributen muss jedes Attribut, das nicht zum Schlüssel gehört, von *allen* Attributen des Schlüssel abhängig sein. Es darf also keine *partiellen* Schlüsselabhängigkeiten geben.

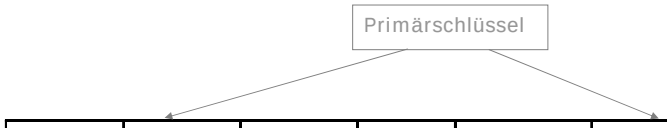
3NF Zwischen zwei Attributen, die nicht zum Schlüssel gehören, darf keine Abhängigkeit bestehen. Es darf also keine *transitiven* Abhängigkeiten geben.

Die nullte Normalform (0NF) ist inoffiziell. Es soll aber nochmals festgehalten werden, dass die Definition eines Primärschlüssels eine zwingende Voraussetzung für die Normalisierung ist.

Diese Definitionen sind nur Daumenregeln. Eine exaktere Definition ist unter dem Kapitel 'Funktionale Abhängigkeiten' zu finden.

Es gibt viele weitere Normalformen, mit denen man unter praktischen Umständen jedoch nie in Konflikt gerät oder deren Auflösung zu umständlich wäre. Gelegentlich gibt es Probleme mit der Boyce-Codd Normalform. Diese ist deshalb am Schluss des Kapitels noch aufgeführt.

1. Normalform (1NF)



Messung	stations Nr	stations Name	mess-netzNr	messnetz Name	datum	bewölkung	°C	SO_2
	1	Aare	1	Uni	1.6.01	stark	21.3	10
	1	Aare	1	Uni	2.6.01	mittel	18.2	12
	1	Aare	1	Uni	3.6.01	schwach	12.8	14
	2	Rathaus	2	Kanton	1.6.01	stark	21.8	21
	2	Rathaus	2	Kanton	2.6.01	mittel	18.9	20
	2	Rathaus	2	Kanton	3.6.01	schwach	13.8	22
	3	Loeb	2	Kanton	1.6.01	stark	21.0	23
	3	Loeb	2	Kanton	2.6.01	mittel	17.2	26
	3	Loeb	2	Kanton	3.6.01	schwach	14.8	28

Atomare Domänen sicherstellen.

Vorgehen für die Überführung von unnormalisierten Tabellen in 1NF

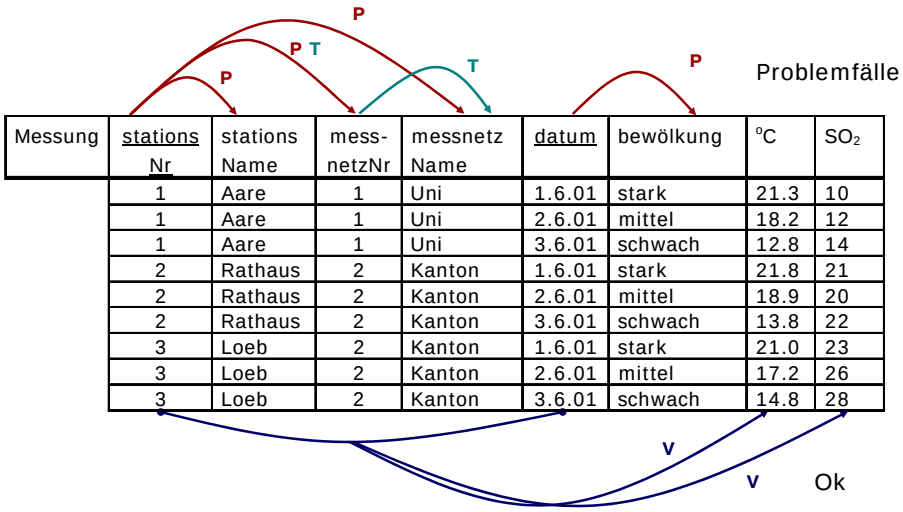
Das Relationenschema wird beibehalten. Weil allerdings nur noch ein Wert für jedes Attribut in einem Tupel zugelassen ist, wird jedes Tupel aus der unnormalisierten Tabelle in ein oder mehrere Tupel der 1NF-Tabelle abgebildet, entsprechend dem durch die Problemstellung gegebenen Zusammenhang (z.B. unabhängige Elemente der mengenwertigen Attribute oder eins zu eins Entsprechungen). Durch die Vervielfältigung der Tupel wird natürlich auch der bisherige Primärschlüssel ungültig und es muss ein neuer definiert werden. In diesem Fall besteht der neue Primärschlüssel aus den Attributen 'stationsNr' und datum'.

Alle vorhandene Information kann über SQL-Abfragen der Datenbank entnommen werden. Beispielsweise ist eine Abfrage nach dem maximalen SO_2-Wert oder der vollständigen Menge aller Wetterberichte (Datum mit Bewölkungsangabe) jetzt möglich.

Tabellen in erster Normalform enthalten enorm viel Redundanz. Ein Update-Befehl betrifft sehr viele Einträge.

In obigem Beispiel bestehen immer noch unerwünschte Abhängigkeiten: Alle Daten müssen immer noch einer Station und neu sogar einem Datum zugeordnet werden. Das Einfügen neuer Messnetze oder Wetterberichte (Bewölkungsdaten) ist nur in Zusammenhang mit einer Station möglich.

2. und 3. Normalform, funktionale Abhängigkeiten feststellen



Messung	stations Nr	stations Name	mess-netzNr	messnetz Name	datum	bewölkung	°C	SO_2
	1	Aare	1	Uni	1.6.01	stark	21.3	10
	1	Aare	1	Uni	2.6.01	mittel	18.2	12
	1	Aare	1	Uni	3.6.01	schwach	12.8	14
	2	Rathaus	2	Kanton	1.6.01	stark	21.8	21
	2	Rathaus	2	Kanton	2.6.01	mittel	18.9	20
	2	Rathaus	2	Kanton	3.6.01	schwach	13.8	22
	3	Loeb	2	Kanton	1.6.01	stark	21.0	23
	3	Loeb	2	Kanton	2.6.01	mittel	17.2	26
	3	Loeb	2	Kanton	3.6.01	schwach	14.8	28

Legende
- P = Partielle funktionale Abhängigkeit
- T = Transitive funktionale Abhängigkeit
- V = Volle funktionale Abhängigkeit

Eine funktionale Abhängkeit drückt aus: Ein Attributwert ist eine (diskrete) Funktion von einem oder mehreren anderen Attributwerten. Bezogen auf ein x,y,z Koordinationsystem entspricht eine partielle funktionale Abhängigkeit beispielsweise z = f(x) oder z = f(y), eine volle funktionale Abhängigkeit entspricht z = f(x,y).

Nur die eigentlichen Messwerte °C und SO_2 erfüllen die volle funktionale Abhängigkeit. Die Messwerte sind abhängig von der Station und vom Datum.

Funktionale Abhängigkeit

In einer Relation R(A, B, ...) ist das Attribut B von A funktional abhängig, wenn zu jedem Wert von A genau ein Wert von B gehört. Per Definition ist jedes Nicht-Schlüsselattribut einer Relation vom Schlüssel funktional abhängig. Schreibweise: $A \rightarrow B$. Mit $A \not\rightarrow B$ wird explizit ausgedrückt, dass keine funktionale Abhängigkeit vorliegt.

Volle funktionale Abhängigkeit

In einer Relation R(S_1... S_i, B, ...) ist das Attribut B von den Schlüsselattributen S_1...S_i voll funktional abhängig, wenn B von der Kombination S_1...S_i funktional abhängig ist, nicht aber von einzelnen Attributen S_1...S_i allein. Bei nur einem Schlüsselattribut liegt immer eine volle funktionale Abhängigkeit vor.

Partielle funktionale Abhängigkeit

In einer Relation R(S_1... S_i, B, ...) heisst die funktionale Abhängigkeit $S_k \rightarrow B$ partiell, wenn S_k ein Element der Schlüsselattribute S_1...S_i ist.
Eine partielle funktionale Abhängigkeit drückt das Gegenteil zu einer vollen funktionalen Abhängigkeit aus.

Transitive funktionale Abhängigkeit

In einer Relation R(S, A, B) ist das Attribut B vom Schlüssel S transitiv abhängig, wenn B von A und A von S funktional abhängig ist, nicht aber S von A (mit dieser letzten Bedingung werden die Sekundärschlüssel von transitiven Abhängigkeiten ausgeschlossen).

Anmerkung

In obigen Definitionen darf anstelle von 'Attribut' singemäss 'Attributkombination' stehen und unter A, B, C, S usw. sind dann Attribut*mengen* zu verstehen.

Normalisierungsziel

In normalisierten Tabellen sollen nur noch *volle funktionale Abhängigkeiten* auftreten, alle anderen sind zu eliminieren.

2. Normalform (2NF)

Station	stations Nr	stations Name	messnetz Nr	messnetz Name
	1	Aare	1	Uni
	2	Rathaus	2	Kanton
	3	Loeb	2	Kanton

Partielle Schlüssel-abhängigkeiten entfernt.

Wetter	datum	bewölkung
	1.6.01	stark
	2.6.01	mittel
	3.6.01	schwach

Messung	stations Nr	datum	°C	SO_2
	1	1.6.01	21.3	10
	1	2.6.01	18.2	12
	1	3.6.01	12.8	14
	2	1.6.01	21.8	21
	2	2.6.01	18.9	20
	2	3.6.01	13.8	22
	3	1.6.01	21.0	23
	3	2.6.01	17.2	26
	3	3.6.01	14.8	28

Vorgehen für die Überführung von 1NF in 2NF

Die Überführung in 2NF führt immer zu einer Zerlegung der Ausgangstabelle. Einen eindeutigen Zerlegungsalgorithmus gibt es allerdings nicht. Angenähert kann folgendes Prozedere angegeben werden: Ausgangspunkt sind die Schlüsselattribute. Jedes Schlüsselattribut und alle Attribute, die durch dieses eindeutig bestimmt sind, werden in eine neue Tabelle ausgelagert. In der Originaltabelle verbleiben alle Schlüsselattribute und alle übrigen Attribute, die durch den *ganzen* Schlüssel eindeutig bestimmt sind.

Für obiges Beispiel heisst das:

- Die 'stationsNr' bestimmt eindeutig die Attribute 'stationsName', 'messnetzNr' und 'messnetzName' (umgekehrt allerdings nicht).
- 'datum' bestimmt eindeutig das Attribut 'bewölkung'.
- Die Attribute 'SO_2' und °C' sind nur durch 'stationsNr' und 'datum' zusammen eindeutig bestimmt.
- Das Attribut 'messnetzNr' bestimmt eindeutig das Attribut 'messnetzName'. Weil aber 'messnetzNr' nicht zum Schlüssel gehört, wird diese Abhängigkeit im Moment nicht betrachtet.

Durch je eine neue Tabelle (mit einem neu zu definierenden Namen) für die Attribute 'stationsNr' und 'datum' ergeben sich gesamthaft obenstehende drei Tabellen in zweiter Normalform.

Es ist gut ersichtlich, dass die Zerlegung in 2NF eine Reduktion der Redundanz zur Folge hat. Die Information über Stationen und Messnetze sind nur noch einmal vorhanden.

3. Normalform (3NF)

Transitive Abhängigkeiten entfernt.

Station	stations Nr	stations Name	messnetz Nr
	1	Aare	1
	2	Rathaus	2
	3	Loeb	2

Messung	stationsNr	datum	°C	SO_2
	1	1.6.01	21.3	10
	1	2.6.01	18.2	12
	1	3.6.01	12.8	14
	2	1.6.01	21.8	21
	2	2.6.01	18.9	20
	2	3.6.01	13.8	22
	3	1.6.01	21.0	23
	3	2.6.01	17.2	26
	3	3.6.01	14.8	28

Messnetz	messnetz Nr	messnetz Name
	1	Uni
	2	Kanton

Wetter	datum	bewölkung
	1.6.01	stark
	2.6.01	mittel
	3.6.01	schwach

Vorgehen für die Überführung von 2NF in 3NF

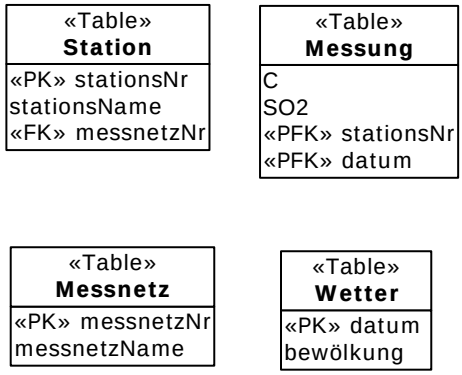
Wie schon die Überführung in 2NF ist auch diejenige in die 3NF eine Zerlegung von Tabellen. Ausgangspunkt für die dritte Normalform sind Abhängigkeiten zwischen Attributen, die nicht zum Schlüssel gehören. Wenn ein Attribut eines oder mehrere andere eindeutig bestimmt, wird eine neue Tabelle mit den beteiligten Attributen erstellt. In der Ausgangstabelle werden die abhängigen Attribute gestrichen.

Für obiges Beispiel heisst das:

Das Attribut 'messnetzName' ist von 'messnetzNr' abhängig. Aus ihnen entsteht die neue Tabelle 'Messnetz'. Damit die Information, zu welchem Messnetz eine Station gehört, erhalten bleibt muss in der ursprünglichen Tabelle 'Station' noch mindestens die 'messnetzNr' vorhanden sein.

Mit dem Erreichen der 3NF ist in aller Regel die Überprüfung des Relationen-modelles beendet. Es exisitieren zwar weitere Normalformen, sie sind aber häufig automatisch erfüllt oder ihre Erkennung ist schwierig. Die früher aus Performance-Gründen vorgenommen Denormalisierung in 2NF ist heute nicht mehr relevant. Relationale Datenbanken sind genügend performant, damit mit einem 3NF Datenmodell gearbeitet werden kann.

UML-Darstellung des Relationenmodelles



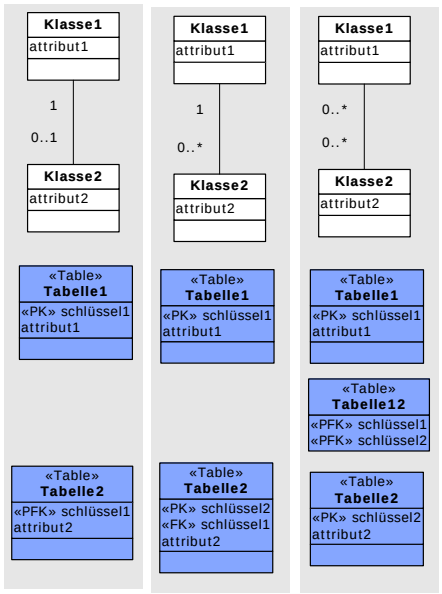
Obiges Diagramm stellt das Relationenmodell für das Schadstoff-Messystem in UML-Form dar. Die Assoziationen sind weggelassen, um zu verdeutlich, dass der Bezug zwischen den Tupeln zweier Relationen (Tabellen) nur noch durch Werte bestimmter Attribut hergestellt wird, nämlich den Primärschlüsseln und den Fremdschlüsseln.

Verwendet man UML zur Darstellung eines Relationenmodelles, so gibt es einige spezielle Notationen:

- Der Stereotyp «Table» verdeutlicht, dass es sich nicht mehr um eine Klasse im OO-Sinn, sondern um eine Relation beziehungweise Tabelle im Datenbanksinn handelt. Häufig trifft man auch auf den Stereotyp «Entity» in UML-Diagrammen. Dieser bezeichnet eine Klasse, deren Objekte (resp. die Attributwerte) in einer Datenbank abgelegt werden sollen. Eine Klasse vom Typ «Entity» muss im Rahmen der Anforderungen 2 und 3 eventuell in mehrere Relationen abgebildet werden müssen. Es können also aus einer «Entity»-Klasse durchaus mehrere «Table»'s entstehen.
- Der Stereotyp «PK» bedeutet, dass es sich beim markierten Attribut oder den markierten Attribut*en* um den Primärschlüssel (Primary Key) handelt. Primärschlüssel werden in anderen Darstellungen sehr häufig durch *Unterstreichen* gekennzeichnet.
- Der Stereotyp «FK» bedeutet, dass es sich beim markierten Attribut oder den markierten Attribut*en* um einen Fremdschlüssel handelt, also um eine Referenz auf einen anderen Primärschlüssel.
- Der Stereotyp «PFK» bedeutet, dass es sich beim markierten Attribut oder den markierten Attribut*en* sowohl um ein Mitglied des Primärschlüssels wie um ein Mitglied eines Fremdschlüssels handelt.

☞ Siehe Skript 'Datenmodellierung mit UML' für eine detaillierte und ausführliche Beschreibung.

Daumenregeln Überführung



Konzeptionelles Modell

Relationen modell

Die Überführung vom UML-Modell in das Relationenmodell RM geschieht unter den Gesichtspunkten:

- Möglichst wenig Relationen (Tabellen) erzeugen.
- Alle Informationen und Zusammenhänge im UML müssen im RM erhalten bleiben.
- Es sind nur atomare Domänen verfügbar.
- Redudanzfreiheit durch Normalisierung.

Zur Erklärung von «PK» «FK» und «PFK» siehe vorhergehende Folien.

Zusammenfassung

- Das konzeptionell Modell definiert den Informationsgehalt einer Datenbanken. Es ist auf das *WAS* ausgerichtet.
- Das Relationenmodell ist auf die Abbildung der Information in eine *relationale* Datenbank ausgerichtet. Es ist auf das *WIE* ausgerichtet. Aus einer UML-Klasse resultieren eine bis mehrere Tabellen.
- Das Relationenmodell *muss* normalisiert sein.

Das Relationenmodell enthält letzlich nicht nur die aus dem konzeptionellen UML-Modell abgeleiteten und normaliserten Tabellen, sondern meist auch noch verschiedene

technische Hilfstabellen, beispielsweise für das Aufbewahren des zuletzt gebrauchten Primärschlüsselwertes, oder für das Mitführen von Session-Information in einem Web-Shop etc.

SQL I

- Übersicht
- Tabellen erzeugen
- Datentypen
- Tabellen modifizieren
- Tabellen abfragen

Die folgenden Beispiele und Erklärungen sind auf Sybase (MS SQL) bezogen, welches in vielen Teilen mit SQL-3 Entry-Level konform ist. Abweichungen vom Standard SQL-3 wurden soweit wie möglich kenntlich gemacht.

Was ist SQL?

- SQL ist eine deklarative Sprache, d.h. sie definiert *was* getan werden soll, nicht *wie*.
- Jeder Datenbankbenutzer und jede Applikation kommuniziert in einer relationalen Umgebung ausschliesslich über SQL mit der Datenbank.
- SQL-Befehle können interaktiv oder eingebettet in verschiedene Programmiersprachen ausgeführt werden.
- SQL ist ein ANSI/ISO Standard in drei grossen Schritten: 1989, 1992, 1999

Im Juni 1970 veröffentlichte Dr. E.F. Codd im Association of Computer Machinery Journal den Artikel "A Relational Model of Data for Large Shared Data Banks". Das dort beschriebene Modell ist als das definitive theoretische Modell für RDBMS akzepiert.

SQL ist eine (nicht ganz perfekte) Implementation der Relationenalgebra.

Die Sprache "Structured English Query Language" (SEQUEL) wurde bei IBM entwickelt, um Codd's Modell zu implementieren. SEQUEL wurde später zu SQL (Structured Query Language). 1979 wurde die erste kommerzielle SQL-Implementation von Oracle auf den Markt gebracht. Der erste Standard von SQL wurde 1989 herausgegeben.

SQL wird laufend weiter entwickelt, der neueste Standard von ANSI (ISO) ist SQL-3 (auch SQL-99 genannt). Vorhergende sind SQL-92 rsp. SQL-89. SQL-3 ist mit SQL-92 fast ausnahmslos konform, enthält jedoch wesentliche Erweiterungen, beispielsweise objektorientierte Ansätze und Anbindungen für die Sprache Java in SQL-J.

Da Daten in der Regel längerlebig sind als Applikationscode, kommt der Kontinuität von SQL eine wichtige Bedeutung zu. Die Hersteller von Datenbanksystemen sind bemüht, ihren SQL-Dialekt in Richtung Standard weiter zu entwickeln und gleichzeitig die proprietären Erweiterungen oder Spezialitäten zu pflegen. Die Entwickler haben damit die Wahl, nur mit Konstrukten aus dem Standard-Umfang zu arbeiten, oder die speziellen Produkteigenschaften mit den entsprechend proprietären Konstrukten voll zu nutzen. Typisch proprietäre Eigenschaften sind in den Bereichen Datentypen, Defaultwerten, Abfrage-Befehle, Concurrency Control und physische Datenorganisation zu finden.

SQL-Befehle, Beispiele

```
create table Person (
   idPerson integer,
   name varchar(64)
)
```

```
insert Person (idPerson, name)
values ( 3, 'Muster' )
```

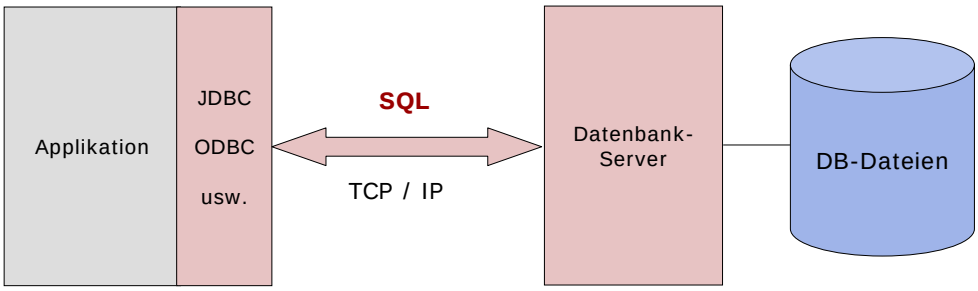
```
select *
from Person
where name = 'Muster'
```

```
update Person
set name = 'Muster-Müller'
where idPerson = 3
```

```
delete Person
where idPerson = 3
```

SQL als Protokollsprache

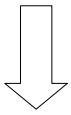
- SQL kann im Netzwerksprachgebrauch als Kommunikationsprotokoll auf Layer 7 aufgefasst werden.
- SQL - Befehle werden über eine Netzwerkverbindung an den Datenbank-Server geschickt.
- Der Datenbank-Server schickt das Abfrage-Resultat an die Applikation zurück.



SQL vs. Programmiersprache

SQL

```
select name
from Person
where persNr = '6'
```



C

```
prs = fopen ("Person.dat", mode );
while(
fscanf(prs, "%s%s ",persnr, name)> 0 ) {
   if( strcmp(persnr, "6") == 0 ) {
       printf( "%s %s", name );
   }
}
fclose ( prs );
```

- Lesbarere Statements
- Zugriff durch DBMS optimierbar
- Abfrage wird beim DBMS durchgeführt
- Grosser Overhead für Parsing und Durchführung

Funktionsweise von SQL

SQL lässt den Benutzer die Daten auf der logischen Ebene benutzen, er braucht grundsätzlich keine Kenntnisse darüber, was im Hintergrund abläuft. Alle Tupel, welche die Selektionskriterien erfüllen, werden als Einheit dem User Interface oder dem Applikationsprogramm oder auch einer weiteren Abfrage übergeben.

Man unterscheidet drei Gruppen von SQL-Statements:

1. Data Definition Language DDL: Definition von Datenbankobjekten (Tabellen) erstellen, verändern und löschen.
2. Data Control Language DCL: Zugriffsrechte auf die Datenbankobjekte verwalten, Konsistenz unter den Datenbankobjekten definieren.
3. Data Manipulation Language DML: Daten einfügen, verändern, löschen. Daten abfragen (dies ist der komplexeste Teil von SQL).

SQL-Tabellendefinition, Beispiel

```
create table Person
(
  idPerson      numeric(8,0) not null
                default autoincrement,
  name          varchar(30)  not null,
  vorname       varchar(15)  not null,
  gebDatum      date         null,
  anzKinder      integer      null,
  ahvNr         varchar(14)  not null,
  primary key   (idPerson),
  unique        (ahvNr),
  check         (ahvNr like '___.__.___.___' )
)
```

Eine SQL-Tabelle definiert ein Relationenschema, d.h. die Name und Domäne der zum Relationenschema gehörigen Attribute. Ausserdem können in einer SQL-Tabelle der Primärschlüssel, allfällige Sekundärschlüssel, sowie allfällige Referenzen auf andere Tabellen (Fremdschlüssel) definiert werden. Mit der Definition der SQL-Tabelle wird in der Datenbank auch alle notwendige Verwaltungs-Infrastruktur für die Datensätze (Tupel) vorbereitet. Dazu gehört beispielsweise Information über den Besitzer, Indices für den schnelleren Zugriff, vorreservierter Platz usw.

SQL ist eine Realisierung des Relationenmodelles. Es gibt daher nur einfache, atomare Datentypen für die Attribute. Möchte man beispielsweise für eine Person *mehrere* Vornamen in der Datenbank festhalten, wird eine *zweite Tabelle* (Relation) notwendig, in welcher pro Datensatz (Tupel) ein Vorname festgehalten werden. Jeder Datensatz der zweiten Tabelle enthält eine Referenz (Fremdschlüssel) auf die erste Tabelle.

Tabellen werden mit `drop table` *`tablename`* wieder gelöscht. Generell gilt: `drop` ist das Gegenteil von `create` bezüglich Datenstruktur-Befehlen.

Die Angabe `default autoincrement` als automatischer Zähler für den Primärschlüssel ist Sybase-spezifisch und soll hier als Beispiel für eine typische Produkterweiterung genommen werden. Die automatische Schlüsselvergabe wird in SQL-3 nicht geregelt, jedoch in den meisten Produkten angeboten.

Der like-Operator ist testet den Ausdruck links auf ein bestimmtes Pattern. Ein Pattern kann echte Zeichen enthalten, sowie die Wildcards '_' und '%'. '_' bezeichnet ein einziges, beliebiges Zeichen. '%' bezeichnet ein 0 bis n beliebige Zeichen. In der Praxis existieren verschiedenste Erweiterungen des like-Operators. In SQL-3 Standard sind diese Erweiterungen im `similar`-Operator zusammengefasst, der den Vergleich eines Strings mit einem allgemeinen regulären Ausdruck ermöglicht.

Angaben für die Tabellendefinition beinhalten:

- Attributname, Datentyp, Nullwert, Defaultwert, automatische Schlüsselwerte
- Primärschlüssel, Sekundärschlüssel (`unique` Angabe)
- Referenzielle Integriätsbedingungen
- Semantische Integritätsbedingungen

SQL-Tabellendefinition, Beispiel ff

```
create table Adresse (
   idAdresse  numeric(10,0) not null
              default autoincrement,
   idPerson   numeric(10,0) not null,
   strasse    varchar(200)  not null,
   primary key( idAdresse ),
   foreign key( idPerson ) references Person( idPerson )
)
```

Das Bezeichnen von Primär- und Sekundärschlüsseln kann zur automatischen Erzeugung entsprechender Indices führen.

Häufig kennen Datenbanksysteme auch temporäre Tabellen. Deren Lebensdauer kann auf die Transaktion, die Session oder den DBMS-Prozess beschränkt sein. Sie sind immer im Hauptspeicher angelegt, müssen nicht gesichert werden, unterliegen keinem gleichzeitigen Zugriff durch mehrere Benutzer und sind daher sehr schnell.

In Zusammenhang mit der Tabellendefinition stehen:

- Rechte für Lesen, Ändern, Löschen, Einfügen
- Zusätzliche Indices
- Views
- Triggers

Weitere Angaben bei der Tabellendefinition sind möglich, aber sehr produktspezifisch. Es sind meist Angaben zu:

- Speicher-Ordnung, zum Beispiel sequentiell ungeordnet, B-Tree mit Schlüssel, Hash mit Schlüssel (beeinflusst nur die Performance, nicht das Resultat eines SQL-Befehles).
- Füllgrad (Platzreserve bei Änderungen an Daten)
- Speicherort (Tablespace, Datenbank-Device)
- Locking-Verhalten (Tabelle, Page, Record)
- Logging (Mit oder ohne Logging)
- Caching (Grösse, Cache-Strategie LRU oder MRU)
- Replikationsangaben

SQL-3 Datentypen

- Zahlen: `INTEGER`, `SMALLINT`, `NUMERIC(p,s)`, `DECIMAL(p,s)`, `FLOAT(p)`, `REAL(p)`, `DOUBLE PRECISION(p)`
- Bitwerte: `BIT(n)`, `BIT VARYING(n)`
- Binärdaten: `BLOB(n)`
- Zeichen: `CHAR(n)`, `VARCHAR(n)`, `NCHAR(n)`, `NCHAR VARYING(n)`, `CLOB(n)`, `NCLOB(n)`
- Zeit: `DATE`, `TIME(p)`, `TIMESTAMP(p)`, `INTERVAL`
- Logik: `BOOLEAN`

Der SQL-Standard macht, ganz genau betrachtet, keine Angaben über die Art der Implementation der Datentypen. Beispielsweise ist nicht definiert, wieviele Bytes und damit einen wie grossen Wertebereich ein `INTEGER` oder ein `FLOAT` umfasst. Aus praktischer Sicht gibt es jedoch Datentypen, welche sich faktisch überall gleich verhalten, während bei anderen recht grosse Differenzen zwischen den Herstellern bestehen.

`INTEGER` und `SMALLINT` werden überall angeboten und sind in 4 Bytes resp. 2 Bytes abgelegt.

`NUMERIC` und `DECIMAL` werden synonym gehandhabt und sind nützlich für Fix-Punkt Arithmetik. Berechnungen mit Eingaben im Rahmen der angegebenen Präzision p und der angegebenen Kommastellen s ergeben ein *exaktes* Resultat im Rahmen einer Ausgabe mit p Stellen und s Kommastellen. Der Standard macht keine Angaben zu den maximalen Angaben für p und s, liegt jedoch bei vielen DBMS weit über 30.

`FLOAT`, `REAL` repräsentieren Zahlen mit Mantisse/Exponent Darstellung. `FLOAT` und `REAL` werden in der Regel in 4 Bytes abgelegt (gemäss IEEE Spezifikation 745-1985) mit einer Mantisse von 7 Stellen und einem Exponent von -38 bis +35. Wird eine Präzision p angegeben, benützt das Datenbanksystem letzlich entweder eine 4 Byte oder eine 8 Byte Darstellung.

`DOUBLE PRECISION` wird in der Regel in 8 Bytes abgelegt mit einer Mantisse von 15 Dezimalstellen und einem Exponent von -304 - + 308.

Bitwerte werden als Strings eingegeben, beispielsweise mit `B '1001'`. Arithmetische Operationen sind nicht möglich.

Binärdaten (`BLOB` heisst Binary Large Object) können beliebig grosse Datenmengen aufnehmen. Gegenüber Zeichen-Datentypen haben sie keine lexikalische Semantik (zum Beispiel beim Vergleich von Buchstaben mit oder ohne Umlauten). Auf Binärdaten gibt es in der Praxis meist weniger Operationen und Vergleiche als bei Zeichendaten. Die interaktive Ein-/Ausgabe kann in Form von Hexadezimalen Strings erfolgen, beispielsweise `X 'FF00FF'`. Je nach Hersteller heissen sich auch ganz anders.

Der Zeichentyp CHAR nimmt Strings fester Länge auf. Bei Bedarf wird mit Blanks aufgefüllt. Der Zeichentyp VARCHAR nimmt Strings variabler Länge auf und merkt sich neben den Zeichen auch die exakte Länge des Strings. Die Maximale Länge von CHAR und VARCHAR ist je nach Produkt eventuell deutlich eingeschränkt. Die Empfehlung im SQL-Standard nennt 1000 Zeichen als Minimalempfehlung. Häufig liegt die Maximale Länge aber bereits bei 255 Zeichen. Der Trend der Hersteller geht dahin, nicht mehr zwischen CHAR und VARCHAR zu unterscheiden. Wichtig ist, dass CHAR und VARCHAR Datentypen Zeichen aus einem *bestimmten Zeichensatz* und einer *bestimmten Sortierordnung* aufnehmen. Dieser Zeichensatz ist je nach Hersteller systemweit oder pro Tabellen-Attribut einstellbar.

Die Zeichentypen NCHAR und NCHAR VARYING unterscheiden sich bezüglich SQL-Standard nicht wesentlich von CHAR und VARCHAR. Der einzige Unterschied liegt darin, dass bei NCHAR und NCHAR VARYING ein systemweit definierter Zeichensatz verwendet wird.

Klassischerweise können CHAR und VARCHAR Zeichentypen technisch nur mit 1-Byte Zeichensätzen umgehen, beispielsweise ISO 8859-1. Immer mehr ist jedoch der UNICODE Zeichensatz mit dem Speicherformat UTF-8 gefragt. Hersteller implementieren deshalb NCHAR und NCHAR VARYING so, dass diese Mehr-Byte Zeichensätze aufnehmen können.

Achtung: Das Resultat von Vergleichsoperationen ist Zeichensatz- und Sortierordnungs-abhängig. Die Sortierordnung heisst auch Collation-Sequence und definiert einen Satz von Regeln für einen bestimmten Zeichensatz. Die Regeln bestimmen, ob ein Zeichen grösser, kleiner oder gleich wie ein anderes Zeichen ist. Dementsprechend können beispielsweise die Zeichen ä a und à identisch sein, sie können unmittelbar aufeinander folgen, oder sie können (bei binärer Sortierordnung) sehr weit auseinander liegen. Die Sortierordnung beeinflusst sowohl das Resultat von Vergleichsoperationen wie auch die Ausgabe mit ORDER BY in select-Befehlen.

Der Typ DATE hat eine Wertebereich von 1. Januar 0001 bis 31.12.9999. Das Standardformat für die Ein-/Ausgabe ist 'yyyy-mm-dd'. Der Typ Time hat das Standardformat 'hh:mm:ss.nnnnnn'. Eine genauere Präzision als 1 Sekunde (p > 0) ist optional. Typische Speicherplatz-Belegung für DATE ist 4 Byte, für TIME 8 Byte.

Der Typ TIMESTAMP ist eine Kombination von DATE und TIME und belegt i.a. 8 Byte Speicherplatz. Das reicht aus für eine Auflösung von 1 Mikrosekunde über den ganzen Datumsbereich von DATE.

Einige Hersteller stellen Mechanismen zur Verfügung stellen, damit jeder vergebene Zeitstempel systemweit eindeutig ist, beispielsweise indem einem Zeitwert ein fortlaufender Zählerwert hinzugefügt wird. Hochaufgelöste Zeitstempel als Primärschlüssel zu verwenden ist allerdings heikel. Es ist zu testen, ob der Zeitstempel auch mit der richtigen Auflösung in die Applikation resp. zurück in die Datenbank gelangt. Soll ein Zeitstempel verwendet werden, um zu prüfen, ob ein Datensatz nach dem letzten Lesen durch einen anderen Prozess verändert wurde, kann auch mit einer fortlaufenden Zahl gearbeitet werden.

Nullwerte

- Null ist eine spezielle Markierung, die anstelle eines Wertes stehen kann.
- Null wird syntaktisch wie eine Konstante behandelt und wird `null` geschrieben.
- Ob `null` zulässig ist für ein Attribut, wird im Rahmen der Tabellen- oder Domänendefinition angegeben.
- Gutes Beispiel: Geburtsdatum einer Person (Daten existieren, sind aber eventuell nicht bekannt)
- Schlechtes Beispiel: Autonummer in einem Personendatensatz. Der Nullwert ist mehrdeutig: Person hat kein Auto oder die Nummer ist nicht bekannt.

Defaultwerte

- Der Defaultwert definiert den Wert für ein Attribut, wenn dieses beim erstmaligen Einfügen eines Datensatzes in die Tabelle weggelassen wird.
- Defaultwerte können beispielsweise sein

```
CURRENT { DATE | TIME | TIMESTAMP | USER }
konstanter Wert | konstanter Ausdruck
NULL
```

- Anwendungsbeispiele

```
create table Person (
    name varchar(30)       default CURRENT USER,
    homepage varchar( 64 ) default 'http://localhost'
... )

create table Fall
     eingegangenAm timestamp default CURRENT TIMESTAMP,
... )
```

Obige Beispiele sind in der Syntax spezifisch für Sybase ASA 9.0, decken aber in etwa die im SQL-Standard definierten Möglichkeiten ab.

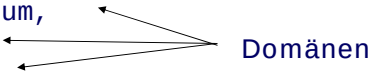
Ein Defaultwert kann im Rahmen einer Domänen-Definition oder direkt bei einem Attribut im Rahmen der Tabellen-Definition angegeben werden. Gibt es sowohl in der Domäne wie in der direkten Defintion eines Attributes einen Default-Wert, hat die Angabe bei der Attribut-Definition Vorrang (Lokalitätsprinzip).

Die Gross-/Kleinschreibung für die Schlüsselworte ist meist nicht relevant.

Domänen

- Eine Domäne definiert einen Wertebereich für ein Attribut:

```
create table Person (
     gebDatum      GebDatum,
     homepage      Url,
     land          Land
)
```



- Eine Domäne schränkt einen bestehenden Basistyp weiter ein:

```
create domain GebDatum date null
       check (value <= getdate() )
create domain Url varchar(128)
       check (value like 'http://%' )
create domain Land varchar(2)
       check (value in (select land from Land ) )
```

Domänen dienen nicht dem *Erzeugen* von Werten, sondern der Definition, welche Werte *erlaubt* sind.

Obige Syntax genügt dem SQL-3 Standard. In In Sybase ASA 9.0 wäre das Schlüsselwort `value` durch eine Variable mit vorabgestelltem @ zu ersetzen, beispielsweise `@wert`.

Achtung: Die Bedingung in der Domänendefinition gilt als erfüllt, wenn sie entweder als `true` oder `null` evaluiert wird.

Es können mehrere check-Klauseln nacheinander aufgeführt werden. Genausogut kann aber auch innherhalb einer Bedingungen mit AND- oder OR-Verknüpfungen gearbeitet werden. In SQL-3 kann jede Bedingung auch einen Namen tragen (Constraint Name), über den sie später gelöscht werden kann.

In SQL-3 kann ein Domain geändert werden mit `alter domain`. Es ist möglich, den Defaultwert oder die Bedingung zu löschen, respektive neu hinzuzufügen. Die bestehenden Datensätze müssen einer allfällig geänderten Domain-Bedingung genügen. In Sybase ASA 9.0 kann die Domäne über alter table geändert werden. Datensätze, welche eine allfällig neuen Bedingung genügen, werden ohne Fehlermeldung beibehalten.

Die explizite Definition einer Domäne ist sehr nützlich zur strikten Kontrolle von Zustandsattributen mit einer meist geringen Zahl von Werten, beispielsweise:

Zivilstand -> ledig, verheiratet

Geschlecht -> m, f

Bestellung -> laufend, ausgeliefert, bezahlt

Ausleihung -> laufend, gemahnt, zurueck

Hotline-Fall -> entgegengenommen, in Bearbeitung, abgeschlossen, abgebrochen.

Randbemerkung: Die zweistelligen Ländercodes sind in ISO 3166 definiert.

Automatische Werte

- Automatische Werte werden bei *jeder* Modifikation resp. jeder Einfügung neu berechnet.
- Die Definition von automatischen Werten findet häufig im Rahmen von Domänen statt.
- Beispiele aus Sybase ASA 9.0:

```
create domain ID integer default AUTOINCREMENT

create table Person (
  idPerson ID,
  geaendertAm   timestamp  default TIMESTAMP,
  geaendertVon varchar(32) default LAST USER
)
```

Die Definition von automatischen Werten im Rahmen von Domänen ist konzeptionell nicht ganz richtig: Die Domäne ist ja nicht zuständig für die *Vergabe* von Werten, sondern legt nur die *möglichen* Werte eines Attributes fest. Trotzdem arbeiten viele Produkte auf diese Weise.

Der SQL-Standard macht gar keine Aussagen zu automatischen Werten, hingegen gibt es in den verschiedenen Produkten einige schöne Möglichkeiten. Obige Beispiele sind in der Syntax spezifisch für Sybase ASA 9.0.

Ein häufiger Automatismus ist ein Zähler für den Primärschlüssel einer Tabelle.

Die Angaben TIMESTAMP und LAST USER haben zur Folge, dass ein entsprechendes Attribut bei jeder Modifikation (inkl. Einfügung) auf den aktuellen Wert von TIMESTAMP oder LAST USER gesetzt wird. Natürlich können die entsprechenden Attribute auch explizit gesetzt werden. Zu beachten ist der Unterschied zu den Angaben CURRENT TIMESTAMP oder CURRENT USER bei den Beispielen zu Domänen, welche lediglich bei der *ersten* Einfügung eines Datensatzes den Wert ihres zugeordneten Attributes setzen. Die Angabe CURRENT USER ist weniger nützlich als es den Anschein hat, weil viele Applikationen gegenüber der Datenbank nur noch als *ein* Benutzer auftreten, unabhängig davon, wer effektiv vor dem Bildschirm sitzt.

Einfach SQL-Modifikationsbefehle

```
insert into Person
 ( name, vorname, gebDatum, anzKinder, ahvNr )
values
 ( 'Muster', 'Franz', '1-jan-2003', 2, '822.59.268.113' )

delete from Person
where idPerson = 3

update Person
set ahvNr = '822.59.268.113'
where persnr = 4
```

Der Unterhalt der Daten geschieht mit den drei SQL-Befehlen `insert`, `delete` und `udpate`.

Zeilenumbrüche haben keine Relevanz für SQL-Befehle. Die Befehle selbst dürfen gross oder klein geschrieben werden. Bei Tabellennamen, Attributnamen usw. wird in vielen System nicht zwischen Gross- und Kleinschreibung unterschieden, es sei denn, die Namen sind in doppelte Anführungszeichen eingeschlossen (quoted identifiers). Ob bei Stringdaten zwischen Gross- und Kleinschreibung unterschieden wird, kann bei vielen Datenbanksystem beim Erstellen der Datenbank definiert werden. Gemäss SQL-Standard muss wählbar sein, ob Gross-/Kleinschreibung für Stringdaten relevant ist.

Attribute, welche automatische Schlüssel enthalten, dürfen in der Liste der einzufügenden Namen nicht vorkommen. Für Attribute mit zugelassenem Null-Wert darf die Konstante `null` verwendet werden. Sind bestimmte Attributnamen nicht aufgeführt, wird für sie automatisch ein Null-Wert oder ein eventuell definierter Default eingefüllt.

Je nach Datenbanksystem oder Einstellung desselben werden einfache oder doppelte Anführungszeichen für Strings und Datumswerte verwendet. SQL-3 definiert einfache Anführungszeichen, die doppelten sind für quoted identifiers (z.B. Attributnamen mit Spezialzeichen darin) reserviert.

Ein `delete`-Befehl ohne `where`-Teil löscht den ganzen Tabellen-Inhalt, jedoch nicht die Tabelle an sich. Lezteres muss mit `drop table` geschehen.

Das Standard-Format von Datumswerten ist gemäss SQL- und ISO-Standards `'YYYY-MM-DD'`. Jedes Datenbanksystem hat zusätzliche Befehle mit dem die Eingabe- und Ausgabeformate von Datumswerten gesetzt werden können. Bei Sybase ASA sind dies die Optionen `DATE_FORMAT` für die Ausgabe, und `DATE_ORDER` für die Eingabe.

Einfache SQL-Abfragen

```
select * from Person

select upper(name), vorname, gebDatum
from Person
order by name, vorname

select * from Person
where idPerson = 3

select name || ' ' || vorname
from Person
where name like 'I%' or name like 'J%'
order by name, vorname

select name, vorname from Person
where gebDatum is null
```

Der select-Befehl ermöglicht einfache, aber auch sehr komplexe Abfragen. Die generelle Struktur einer Abfrage sieht wie folgt aus:

Der select-Teil definiert die Attribute der Resultattabelle (Projektion). Ein * heisst, dass alle Attribute der im from-Teil genannten Tabellen im Resultat erscheinen. Im select-Teil dürfen auch Funktionen auf Attributen und Aliasnamen für die auszugebenden Attributnamen verwendet werden.

Der from-Teil definiert die Ausgangstabellen. Konzeptionell gesehen wird bei einer Abfrage aus allen genannten Tabellen im from-Teil das kartesische Produkt gebildet. Aus diesem werden im select-Teil bestimmte Attribute und im where-Teil bestimmte Datensätze herausgefiltert.

Der where-Teil definiert die Bedingungen an die auszugebenden Datensätze (Restriktion).

Der order by-Teil definiert die Sortierung der Ausgabe. Da relationale Datenbanken mengenbasiert arbeiten, kann grundsätzlich nicht von einer Sortierfolge in den Basistabellen ausgegangen werden. Eine Sortierung ist nur garantiert, wenn sie mit order by erzwungen wird.

Gruppierung

- Gruppierungen sind bei der Bildung von Statistiken und der Analyse eines Datenbestandes ein sehr potentes Hilfsmittel.

```
select anzKinder, count(*)
from Person
group by anzKinder

select datepart(year, gebDatum) as Jahr, avg(anzKinder)
from Person
where gebDatum is not null
group by Jahr

select anzKinder, count(*)
from Person
group by anzKinder
having count(*) > 10
```

Obige Aufzählung der Aggregatfunktionen entspricht dem SQL-3 Standard.

Sehr wichtig ist, dass bei der Gruppierung die *einzelnen* Datensätze, aus der die Gruppe *gebildet* wird, verlorengehen. Da SQL nur Tabellen erster Normalform kennt, ist es nicht möglich, im obigen ersten Beispiel noch gleichzeitig die Namen der Personen in jeder Gruppe zu bekommen.

Zu beachten ist der Unterschied zwischen der `where`-Klausel und der `having`-Klausel. Die `where`-Klausel selektiert Datensätze *vor* dem Gruppieren. Die `having`-Klausel selektiert die Datensätze nach dem Gruppieren, sie filtert also gewisse *Gruppen* heraus.

In der `select`-Klausel können nur folgende Attribute ausgegeben werden:

- Eines oder mehrere der Attribute, die auch in der `group by` Klausel stehen.
- Eine so genannte Aggregat-Funktion, welche einen Wert über die ganze Gruppe berechnet: `count(*)`, `sum(a)`, `avg(a)`, `max(a)`, `min(a)`.

Unterabfragen

- Unterabfragen sind nützlich für Abfragen der Art "Suche alle Datensätze welche in einer anderen Tabellen *nicht* vorkommen".

```
select * from Person
where idPerson not in ( select idPerson
                        from Adresse )
```

```
select * from Person p
where not exists ( select *
                   from Adresse a
                   where p.idPerson = a.idPerson )
```

Jede Unterabfrage wird (mindestens konzeptionell) für jeden Datensatz der übergeordneten Abfrage *einmal* durchgeführt.

Unterabfragen sind häufig dann anwendbar, wenn sich Fragen stellen wie "Suche alle x, die nicht in y vorkommen ..." oder "Gib mir alle x, ausser diejenigen, welche....".

Die Daten der Tabelle in der Unterabfrage können nicht ausgegeben werden. Dafür müsste (zusätzlich) mit einem Join gearbeitet werden. → Siehe nächste Seiten.

Kartesisches Produkt

- Der konzeptionelle Ausgangspunkt für viele Abfragen, die Information aus *mehreren* Tabellen zusammen-stellen, ist das *kartesische Produkt*. Beispiel in SQL:

Person	idPerson	name
	1	A
	2	B
	3	C

Adresse	idPerson	strasse
	1	X
	2	Y

```
select * from Person, Adresse
```

Person.idPerson	person.name	Adresse.idPerson	Adresse.strasse
1	A	1	X
1	A	2	Y
2	B	1	X
2	B	2	Y
3	C	1	X
3	C	2	Y

Das Relationenschema eines kartesischen Produktes besteht aus allen Attributen der beiden Ausgangsrelationen. Die Produktrelation wird gebildet indem jedes Tupel der einen Relation mit jedem Tupel der anderen Relation verknüpft wird.

Das kartesiches Produkt kann durch das Datenbanksystem mit Hilfe einer n-fachen Schleife algorithmisch sehr leicht berechnet werden. Natürlich ist das kartesische Produkt selten das, was man als Schlussresultat einer Abfrage möchte. Aber es bildet (mindestens konzeptionell) den Ausgangspunkt für weitere Operationen, d.h. die Einschränkung der Datensätze mit dem where-Teil (Der genaue Begriff in der Datenbanktechnologie heisst Restriktion) und die Einschränkung auf bestimmte Attribute im select-Teil (Der genaue Begriff in der Datenbank-technologie heisst Projektion).

Join

- Aus dem kartesischen Produkt zweier Tabellen wird ein *Join*, wenn folgende Bedingung gilt: Die Werte der gemeinsamen Attribute müssen gleich sein.

Person	idPerson	name
	1	A
	2	B
	3	C

Adresse	idPerson	strasse
	1	X
	2	Y

```
select * from Person, Adresse
where Person.idPerson = Adresse.idPerson
```

Person.idPerson	person.name	Adresse.idPerson	Adresse.strasse
1	A	1	X
2	B	2	Y

Der Join, auch *Verbund* genannt, ist ein zentrale Operation bei Abfragen mit SQL. Ihm gilt auch besondere Aufmerksamkeit bei der Optimierung von Abfragen.

Natürlich kann eine Join-Abfrage mit weiteren Einschränkungen ergänzt werden, beispielsweise

```
select name, strasse
from Person, Adresse
where Person.idPerson = Adresse.idPerson
and name = "Müller"
```

Einer Tabelle im from-Teil kann ein Alias zugeteilt werden. In vielen Fällen ist dies ein reines Hilfskonstrukt um besser lesbare Abfragen zu erhalten. Beispiel:

```
select p.name, a.strasse
from Person p, Adresse a
where p.persnr = a.persnr
and p.name = "Müller"
```

Natural Join

- Der *Natural* Join von zwei Tabellen ist ein Join über die *gemeinsamen* Attribute beider Tabellen.

Person	idPerson	name
	1	A
	2	B
	3	C

Adresse	idPerson	strasse
	1	X
	2	Y

```
select * from Person natural join Adresse
```

Person.idPerson	person.name	Adresse.idPerson	Adresse.strasse
1	A	1	X
2	B	2	Y

Manchmal ist es notwendig, anzugeben, welche Attribute miteinander verknüpft werden sollen. Dann ist folgende folgende Schreibweise möglich:

```
select *
from Person natural join Adresse on ( Person.idPerson =
Adresse.idPerson)
```

Verhalten des Natural Join im SQL-Standard

Der natural join ist *kommutativ*, das heisst:

A natural join B = B natural join A

Der natural join ist *assoziativ*, das heisst:

(A natural join B) natural join C = A natural join (B natural join C)

Wenn zwei Tabellen A und B, respektive B und C, keine gemeinsamen Attribute haben, wird der Natural Join zu einem Kreuzprodukt (Auch Cross Join oder Kartesisches Produkt genannt).

Der Natural Join verhält sich in Sybase leider nicht assoziativ, sondern vergleicht nur Attribute unmittelbar benachbarter Tabellen im From-Ausdruck, was zu Problemen führt, wenn ein Join gebildet werden soll, bei dem *eine* Tabelle gemeinsame Attribute zu *mehreren* anderen Tabellen hat.

Besser als der Natural Join wäre eine Join-Möglichkeit, die sich an den definierten Fremdschlüssel/Primärschlüssel-Beziehungen ausrichtet. Das ist z.B. bei Sybase möglich, jedoch nicht im SQL-Standard.

Outer Join

- Ein *Outer Join* ist ein Join mit der zusätzlichen Bedingung: Jeder Datensatz der linken Tabelle muss mindestens einmal im Resultat erscheinen.

Person	idPerson	name
	1	A
	2	B
	3	C

Adresse	idPerson	strasse
	1	X
	2	Y

```
select * from Person natural left outer join Adresse
```

Person.idPerson	person.name	Adresse.idPerson	Adresse.strasse
1	A	1	X
2	B	2	Y
3	C	null	null

Outer-Joins sind ein häufiges Bedürfnis in der Praxis. Obige Syntax ist Sybase-spezifisch. Gemäss SQL-3 Standard müsste folgende Syntax verwendet werden:

```
select * from Person left outer join Adresse
```

Falls ein Datenbanksystem keine Outer-Joins anbietet kann er in etwa nachgebildet werden mit folgender Abfrage, die syntaktisch fast immer erlaubt ist:

```
select p.persnr, p.name, a.strasse
from Person p, Adresse a
where p.persnr = a.persnr
union
select p.persnr, p.name, null
from Person p
where p.persnr not in ( select a.persnr from Adresse a )
```

Der Outer-Join ist nicht direkt aus dem kartesichen Produkt von Person und Adresse ableitbar sondern ist eine Vereinigung zweier Abfragen, wie oben dargestellt.

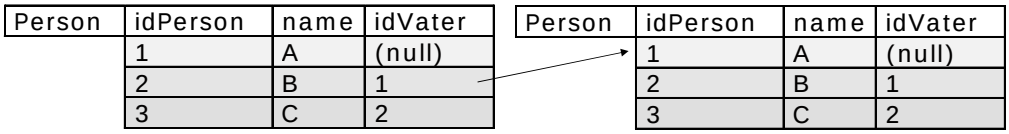
Manchmal ist es notwendig, anzugeben, welche Attribute miteinander verknüpft werden sollen. Dann ist folgende folgende Schreibweise möglich:

```
select *
from Person natural left outer join Adresse
            on ( Person.idPerson = Adresse.idPerson)
```

Diese Schreibweise ist notwendig, wenn die zu verknüpfenden Attribute nicht gleich heissen, oder wenn die zu verknüpfenden Tabelle nicht unmittelbars benachbart sind.

Self Join

- Eine Tabelle kann mit sich selber verbunden werden. Man spricht von einem Self Join. Beispiel: welche Person ist Vater von B?



Person	idPerson	name	idVater
	1	A	(null)
	2	B	1
	3	C	2

Person	idPerson	name	idVater
	1	A	(null)
	2	B	1
	3	C	2

```
select p2.name
from Person p1, Person p2
where p1.idVater = p2.idPerson
and p1.name = 'B'
```

- Die Tabellen gelten durch ihre Umbennung mit einem Alias logisch als zwei verschiedene Tabellen.

Wenn Abfragen auf eine Tabelle durchgeführt werden, die mit sich selbst in Beziehung steht wird die Verwendung von Alias-Namen zwingend.

Join, Natural Join und Self Join werden unter auch unter dem Sammelbegriff Inner Join zusammengefasst.

SQL-Operatoren und Funktionen

- Vergleichsoperatoren (where-Klausel)
  `=, <, >, <=, >=, <>, between, like`
- Boolsche Operatoren (where-Klausel)
  `and, or, not, ( ), is null, is not null`
- Mengenoperatoren (where-Klausel)
  `in, not in, exists,`
  `>any, <any, =any, >all, <all, =all`
- Mathematische Operatoren (where- und select-Klausel)
  `+, -, *, /`
- Systemfunktionen (where- und select-Klausel)
  `upper(), lower(), trim(), substring(), ...`

Alle oben genannten Operatoren und Funktionen sind im SQL-3 Standard auf-geführt. Im SQL-Standard und insbesondere in den meisten produktspezifischen SQL-Dialekten existiert noch eine Vielzahl weiterer Funktionen, vorallem in den Bereichen

- Stringverarbeitung (Zusammensetzen, Einfügen, Ersetzen, Patternmatching)
- System (Datenbankname, Benutzerdaten, aktuelles Datum und Uhrzeit)
- Datentypkonversion
- Datums- und Zeitverarbeitung (Datumsteile extrahieren, Summe, Differenz)
- Mathematische Funktionen (Winkelfunktionen, Runden, Zufallszahlen, Potenzierung)

Nullwerte in SQL-Klauseln

- Suchbedingungen (`where ... )`
  sind erfüllt, wenn die Prüfung TRUE ergibt.
- Constraints (`check ... )`
  sind erfüllt, wenn das Resultat TRUE oder NULL ist.
- Gruppierungen (`group b ... )`
  ergeben eigene Gruppe für den NULL-Wert.
- Einmaligkeit(`unique(...)`)
  Es dürfen beliebige viele NULL-Werte vorkommen.
- Einmaligkeit(`distinct`)
  Der NULL-Wert kommt nur einmal vor.

Zu beachten ist, dass die verschiedenen Produkte oft von diesen Regeln abweichen. In Sybase darf z.B. der NULL-Wert in einem unique-Attribut nur einmal vorkommen.

Der Datenbank-Papst C.J. Date betrachtet die Einführung von Null-Werten als äusserst fragwürdig und plädiert für die Anwendung von gut gewählten Default-Werten, ohne spezielle Behandlung mit einer dreiwertigen Logik.

JDBC

- Übersicht
- Programmierung
- Transaktionskontrolle
- Fehlerbehandlung

Die folgenden Informationen basieren 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-Hersteller (Sybase ASA 9.0) 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.

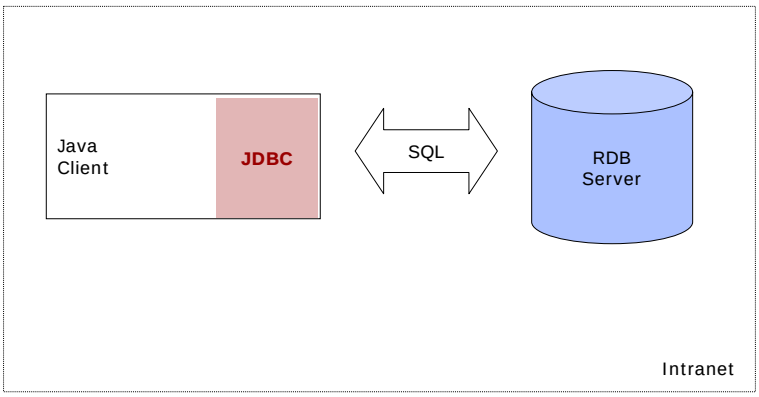
JDBC 3.0 ist im Wesentlichen eine Abrundung und Konsolidierung von JDBC 2.1 und dem Optional Package 2.0. Einige kleinere Änderungen im Funktionsumfang sind hinzugekommen. Ausserdem kann JDBC 3.0 auf die definitive Version von SQL99 abstellen.

Was ist JDBC ?

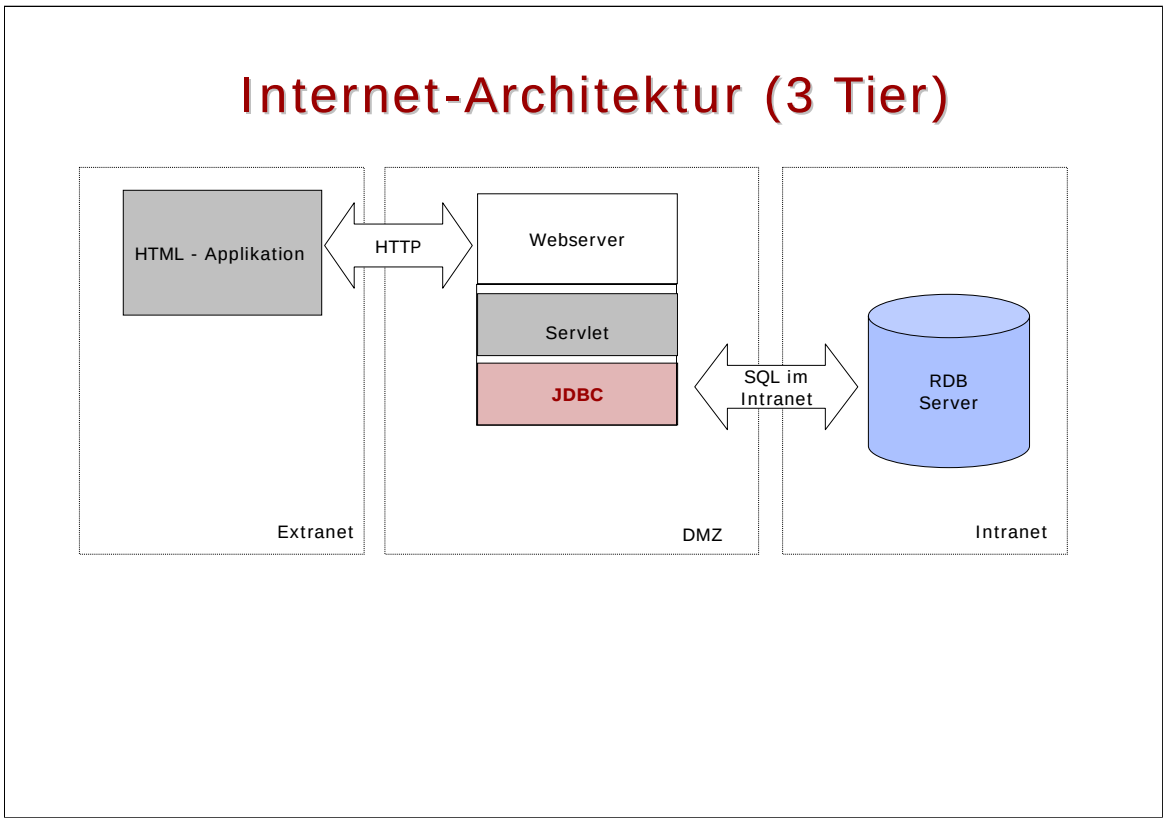
- JDBC™ = Java Database Connectivity. Vergleichbar mit ODBC (Open Database Connectivity von Microsoft für C-Programme)
- JDBC ist ein low-level oder call-level API mit drei Funktionen
  - Verbindung zu einer Datenbank herstellen
  - SQL-Befehle absetzen
  - Resultate verarbeiten
- JDBC *ermöglicht* einen produktunabhängigen Datenbankzugriff, *erzwingt* ihn aber nicht.

JDBC 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.

Client/Server Architektur (2 Tier)

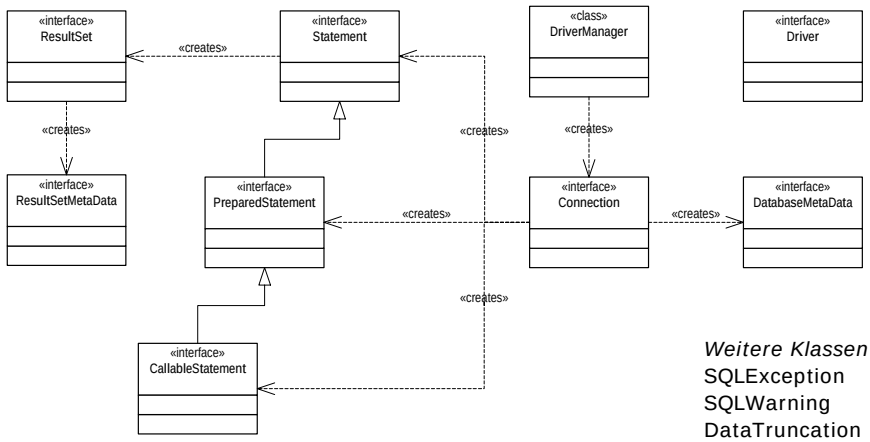


Diese Architektur ist geeignet für lokale Applikationen im Intranet auf einer bestimmten Betriebssystemplattform.



Diese Architektur ist geeignet für einfache Internet-Applikationen.

Das Package `java.sql`



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:swsdb:2638?ServiceName=b31klass", "dba", "sql" );

// 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 );
  }
  con.commit();
} catch ( SQLException e ) {
  System.out.println( e.toString() );  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.

Von der Methode `DriverManager.getConnection()` existieren verschiedene Varianten. Neben der oben dargestellten wird sehr häufig die Form `getConnection( String url, Properties p )` verwendet. Der Parameter `url` enthält minimale Angabe zum Verbindungsaufbau, alle übrigen Angaben wie Datenbank, Username, Passwort etc. werden über das `Properties`-Objekt mitgegeben. Das hat den Vorteil, dass zahlreiche, produkt-spezifischen Angaben im gleichen `Properties`-Objekt mitgegeben werden können.

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

Detail-Beispiel für die Vewendung eines Statement-Objektes:

```
...
 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 ); }
 ...
```

Das Statement-Objekt ist der Ausgangspunkt der SQL-Abfrage. Dem Statement-Objekt wird ein SQL-Befehl zur Ausführung übergeben. Dabei ist im ersten Parameter des obigen Aufrufes definiert, wie das zurückgelieferte ResultSet durchlaufen werden kann, und im zweiten Parameter, ob die Daten darin modifizierbar sind. Weil im obigen Fall TYPE_SCROLL_INSENSITIVE gesetzt ist, kann das ResultSet mehrmals und in allen Richtungen durchlaufen, oder jederzeit auf einen beliebigen Datensatz positioniert werden. Mit der Einstellung CONCUR_UPDATABLE können Daten über das ResultSet direkt geändert werden. Die Änderungen werden unmittelbar in der Datenbank nachgetragen. Zu Beachten ist dabei, dass der Modifizierbarkeit durch das Datenbanksystem Grenzen gesetzt sind. Join-Abfragen, Abfragen mit Funktions-aufrufen oder Operatoren in der select-Klausel und Abfragen mit group by-Klauseln können nicht modifiziert werden.

Der Aufruf der meisten Funktionen des ResultSet ist davon abhängig, dass eine Datenbank-Connection aktiv ist. Das Weitergeben eines ResultSet an andere Teile einer Applikation, im Sinnne eines Return-Parameters, ist daher mit Vorsicht zu verwenden. Sicherer ist das Umkopieren von Abfrageresultaten in Listen, Vektoren usw.

Für den den ersten Parameter von Connection.createStatement können folgende Werte eingestellt werden:

ResultSet.TYPE_FORWARD_ONLY heisst, es kann nur vorwärts durch ein ResultSet iteriert werden. Die einzig erlaubte Postionierungsfunktion 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. Wenn gelesene Datensätze aufgrund der Transaktions-einstellungen in der Datenbank ohnehin gesperrt bleiben, ist diese Einstellung überflüssig.

Für den den zweiten Parameter von Connection.createStatement können folgende Werte eingestellt werden:

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 )
```

Prepared Statements (1)

- Beispiel für das Lesen von Daten

```
...
try {
  int id = ... // id von irgendwoher bekommen
  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 ) );
    String name = rs.getString( 1 );
    Date gebdatum = rs.getDate( 2 );
    System.out.println( name );
    System.out.println( gebdatum.toString() );
  }
}
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 mit den entsprechenden set()-Methoden.
- Anführungszeichen in String-Parametern werden automatisch in eine korrekte Escape-Folge umgewandelt.
- Ein prepared Statement kann vom Datenbanksystem vorkompiliert werden und ist dadurch schneller, wenn es mehrmals nacheinander aufgerufen wird.

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 ein Datenwert stehen kann.
- Das Fragezeichen ist nicht erlaubt zur Parametrisierung von Attributnamen, Tabellennamen etc.

Beispiele für PreparedStatements:

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

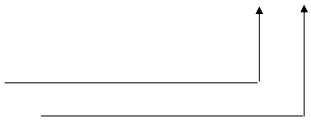
Ein Fragezeichen kann überall dort stehen, wo in einem SQL-Befehl ein Datenwert stehen kann.

Prepared Statements (2)

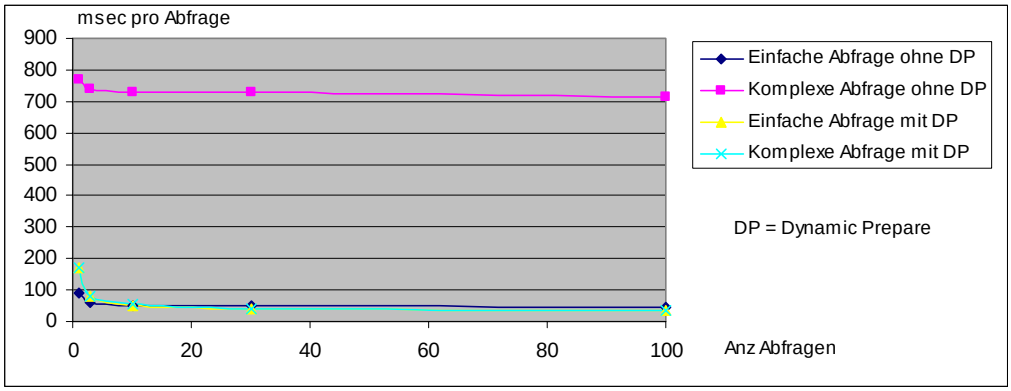
- Beispiel für das Einfügen von Daten

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

try {
   while ( ... ) {
     stat.setString( 1, ... );
     stat.setString( 2, ... );
     stat.executeUpdate();
   }
}
 catch( SQLException e ) { System.out.println ( e ); }
 ...
```



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
```

(Wie oft hat jeder Student eine bestimmte Frage beantwortet)

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 Eithernet, Firewall, CableCom Modem (512/256)

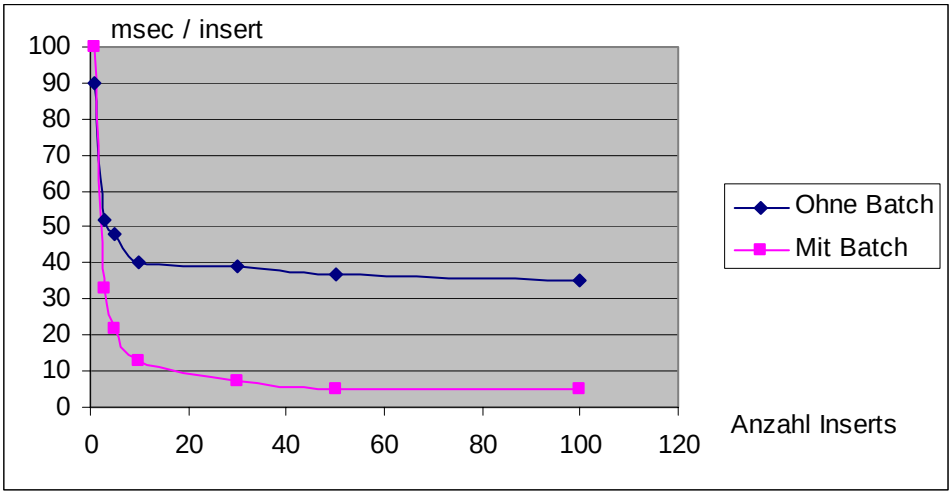
Client: IBM A22m (2001), Java 1.4, jConnect5

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 Eithernet, Firewall, CableCom Modem (512/256)

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

Metadaten, Beispiel

- Die Klasse ResultSetMetaData ist nützlich für die dynamische Bestimmung der Attribute eines Abfragesresultates. Beispiel:

```
ResultSet rs = stmt.executeQuery( "select * from Person" );
ResultSetMetaData rsm = rs.getMetaData();
while ( rs.next() ) {
   for ( int i = 1; i <= rsm.getColumnCount(); i++ ) {
     out.println( rsm.getColumnName(i) + ": "
                + rs.getString(i)  );
     out.println( columnSeparator );
   }
   out.println( rowSeparator );
}
```

Transaktionskontrolle

- JDBC enthält Methoden für
  - das Setzen des Commit-Modus
    `Connection.setAutoCommit( boolean )`
  - das Setzen des isolation level
    `Connection.setIsolationLevel( int )`
  - commit und rollback von Transaktionen
    `Connection .commit(), Connection.rollback()`

  sowie
  - das Setzen von Save Points
  - die Kontrolle von verteilten Transaktionen (XAResourcen)

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.

Fehlerbehandlung

- Da SQL-Befehle ad hoq 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.

Fehlerbehandlung, Beispiel

```
...
try {
  ... sql-statements ...
}
catch ( SQLException e ) {
  while ( e != null ) {
    out.println( "SQL State:  " + e.getSQLState()  );
    out.println( "Error Code: " + e.getErrorCode() );
    out.println( "Message:    " + e.getMessage()   );
    e = e.getNextException();
  }
  try {
    con.rollback();
  }
  catch( Exception x ) {}
}
...
```

Zu beachten ist, dass einige Fehlermeldungen spezielle Aktionen erfordern. Beispielsweise könnte bei einem Deadlock oder bei einem Verbindungsfehler eine Art Retry-Mechanismus nach folgendem Schema versucht werden:

```
boolean retry = true;
while ( retry ) {
  try {
    ... sql-statements ...
    retry = false;
  }
  catch ( SQLException e ) {
   if ( e.getSQLState().equals( "40001" ) ) {
     System.out.println("Deadlock!");
     retry = true;
   }
   else {
     try { con.rollback(); }
     catch( SQLException e2 ) {}
     retry = false;
   }
}
```

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

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

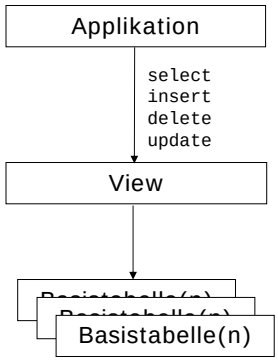
SQL II

- Views
- Constraints
- Triggers
- Funktionen und Prozeduren

Views

- Views sind virtuelle Tabellen, deren Inhalt dem Resultat eines select-Befehles entspricht.
- Views sind eine Datenschnittstelle.
- Views sind eine applikations- oder benutzerspezifische Sichtweise auf gemeinsame Basisdaten.
- Erzeugen einer View

```
create view view_names
[(column_names)]
as select_statement
[with check option]
```



Eine View kann wie eine Interface-Definition in einer Programmiersprache angesehen werden.

Eine View kann in vielerei Hinsicht wie eine gewöhnliche Tabelle behandelt werden. Man kann sie abfragen und mit gewissen Einschränkungen auch modifizieren oder mit Rechten versehen. Eine einmal erzeugte View bleibt solange bestehen, bis sie explizit wieder gelöscht wird (`drop view`). Wenn eine der View zugrundeliegende Tabelle gelöscht wird, merkt dies das Datenbanksystem eventuell erst beim nächsten Zugriff auf die View.

Der Inhalt einer View ist *virtuell*. Beim Abfragen einer View wird in jedem Fall auch der der View zugrundeliegende select-Befehl durchgeführt.

Es gibt auch Datenbanksysteme, welche eine View materialisieren können. Bei der Definition kann dann angegeben werden, in welchen Zeitintervallen die Materialisierung stattfinden soll. Materialisierte Views können zulasten des Speicherplatzes die Abfrage-Performance enorm steigern.

Eine View ist, genau wie eine Tabelle, mit Zugriffsrechten versehen. Auf die zugrundeliegenden Basistabellen benötigt nur der *Erzeuger* der View die notwendigen Rechte. Für die *Benutzung* der View sind ausschliesslich die Zugriffsrechte auf der View selbst ausschlaggebend.

Die Klausel `with check option` stellt sicher, dass Datensätze der View nur geändert oder modifiziert werden können, wenn sie der where-Bedingung des zugehörigen select-Befehles entsprechen.

Views, Beispiel 1

- Gewisse Attribute welche der Tabellenverwaltung durch Systemapplikationen dienen, sollen ausgeblendet werden.

```
create table Person
(
  idPerson        numeric(10,0) default autoincrement,
  kuerzel         varchar(5),
  name            varchar(50)),
  lastmodifdate   datetime null,
  lastmodifuser   varchar(10) null
)
create view PersonDaten (nr, kuerzel, name )
as
select idPerson, kuerzel, name
from Person
```

Obige View basiert nur auf einer einzigen Tabelle. Der zugehörige select-Befehl enthält keine Funktionen in der select-Liste und keine Gruppierung nach der where-Klausel. Über die View können, konzeptionell gesehen, wiederspruchsfrei und ohne Zweideutigkeiten Datenwerte eingefügt oder modifiziert werden. Die View ist deshalb auch *effektiv* modifizierbar und die Änderungen werden an die Basistabelle weiterpropagiert. Für die in der View nicht sichtbaren Felder `lastmodifdate` und `lastmodifuser` werden durch das System die Default-Werte eingefüllt, in diesem Fall also zweimal der Null-Wert.

Das Abfüllen *sinnvoller* Werte für die Attribute `lastmodifdate` und `lastmodifuser` könnte z.B. ein Trigger warnehmen (siehe Kapitel über Trigger).

Views, Beispiel 2

- Daten soll auf vorpräparierte Weise einer Applikation zur Verfügung gestellt werden. Die zugrundeliegenden Tabellen sind versteckt.

```
create table Person
(
  idPerson  numeric(10,0),
  name      varchar(50)
)
```

```
create table Adresse
(
   idAdresse  numeric(10,0),
   idPerson   numeric(10,0),
   strasse    varchar(64)
)
```

```
create view PersonenAdressen ( id, name, strasse )
as
select p.idPerson, p.name, a.strasse
from Person p natural left outer join Adresse a
```

Obiges Beispiel zeigt eine View, welche dem Benutzer das Resultat einer häufig benutzten Abfrage zur Verfügung stellt. Sie versteckt in diesem Beispiel ausserdem die produktabhängige Realisierung eines Outer-Joins.

Die dargestellte View kann nicht modifiziert werden, d.h. die Ausführung von insert-, delete- und update-Befehlen führt zu einem Fehler. Unter folgenden Umständen ist eine View nicht modifzierbar:

- Eines oder mehrere Attribute werden aus den Basistabellen *berechnet.*
  Beispiel: `create view v as select uppercase( name ) from Person`
- Der from-Teil enthält mehr als eine Tabelle (Join, Produkt, Union)
- Die Abfrage enthält eine Gruppierung (`group by` Klausel)
- Der select-Teil enthält das distinct-Schlüsselwort
- Die zugrundeliegende Tabelle enthält Attribute, welche nicht in der View-Definition vorkommen, aber auch keinen Default-Wert (zum Beispiel null) haben.

Constraints (Integritätsbedingungen)

- Ein Constraint bindet eine logische Bedingung an *eine* Tabelle. Es gibt 4 Constraint-Typen:

  `primary key`
  `unique`
  `foreign key`
  `check`

- Die Constraint-Prüfung wird ausgelöst durch einen `insert`-, `update`- oder `delete`-Befehl, und für *jeden von diesem Befehl betroffenen Datensatz* ausgewertet.

- Constraints müssen deterministisch sein, d.h. sie dürfen keine zeitabhängigen oder ausserhalb der Datenbank liegende Funktionen benützen.

`unique`-Constraint

Jede Kombination der genannten Attribute muss über die Tabelle hinweg eindeutig sein. Gemäss Standard dürfen beliebig viele NULL-Werte vorkommen. Zwei NULL-Werte gelten sinngemäss als verschieden.

`primary key`-Constraint

keines der genannten Attribute darf NULL sein. Jede Kombination der genannten Attribute muss über die Tabelle hinweg eindeutig sein.

`foreign key`-Constraint (Referentielle Integritätsbedingung)

Jeder Fremdschlüssel muss einen korrespondierenden Primärschlüssel besitzen oder der Fremdschlüssel muss NULL sein (sofern das die Definition des Fremdschlüsselattributes dies erlaubt). Es gibt zwei Bedrohungen gegen den foreign key Constraint:

1. Verletzung beim Einfügen eines Datensatzes in die Fremdschlüssel-Tabelle. Im diesem Fall weist das Datenbanksystem die Einfügung in zurück.
2. Verletzung beim Löschen von Datensätzen in der Primärschlüssel-Tabelle. In diesem Fall gibt es drei Möglichkeiten, welche letztlich wieder auf die Bedingung führen, dass ein Fremdschlüssel immer einen korrespondierenden Primärschlüssel besitzen oder NULL sein muss: Die Löschung wird verboten, die Löschung wird weitergegeben oder die Fremdschlüssel werden auf NULL gesetzt.

`check`-Constraint (allgemeine Integritätsbedingung)

Die gesetzte Bedingung muss für *jeden* Datensatz wahr oder NULL sein. Damit ein Datensatz korrekt, der check-Constraint also nicht verletzt ist, muss folgende Bedingung *wahr* sein:

```
NOT EXISTS ( SELECT * FROM table WHERE NOT ( check-condition ) )
```

Da ein gelöschter Datensatz nicht mehr existiert, löst ein delete-Befehl nicht die Prüfung der check-Bedingung aus.

Einige Datenbanksysteme beschränken die check-Klausel auf lokale Bedingungen, d.h. Bedingungen, die anhand der eigenen Tabelle oder sogar nur des betroffenen Datensatzes evaluiert werden können. Der Grund liegt im Aufwand herauszufinden, *unter welchen Umständen* die Prüfung von Check-Bedingungen ausgelöst werden muss. Beispiel:

```
create table Person (
 persnr  integer,
 ...
)
create table Adresse (
  persnr    integer,
  adrnr     integer,
  ...
)
alter table Person add check ( persnr in (select persnr from Adresse))
```

Folgende zwei insert Befehle funktionieren korrekt:

```
insert into Adresse values ( 1, 1, ... )
insert into Person values  ( 1, ... )
```

Folgende zwei insert Befehle funktionieren korrekterweise nicht, weil beim Einfügen der Person noch keine Adresse vorhanden ist:

```
insert into Person values  ( 2, ... )
insert into Adresse values ( 2, 1, ... )
```

Folgender delete-Befehl, welcher dazu führt, dass Personen ohne Adressen in der Datenbank auftreten können, wird fälschlicherweise vom DBMS zugelassen:

```
delete from Adresse
```

Es müsste nämlich die Prüfung der check-Bedingung auf der Personen-Tabelle aktiviert werden. Die Verletzung der check-Bedingung wird erst im Rahmen eines update-Befehles, z.B. `update Person set persnr = persnr`, bemerkt.

Constraint Syntax

```
CREATE TABLE tabelle
( {Spaltendef | cons-def [zeitpunkt] }, ... )
cons-def:
  [CONSTRAINT name ]
  {   UNIQUE ( Spaltenname, ... )
    | PRIMARY KEY ( Spaltenname, ... )
    | CHECK ( Bedingung )
    | FOREIGN KEY (Spaltenname, ... )
      REFERENCES tabelle [ (Spaltenname, ... ) ]
      [ ON DELETE { NO ACTION | RESTRICT | CASCADE
                    SET NULL | SET DEFAULT } ]
  }
zeitpunkt:
  [ INITIALLIY DEFERRED | INITIALLY IMMEDIATE ]
```

Die obige Syntax nach SQL-99 ist nicht vollständig, beleuchtet aber die wichtigsten Möglichkeiten. Weggelassen wurde u.a. dass ein Constraint für ein einzelnes Attribut auch nach der Datentyp-Definition angefügt werden kann.

Trigger

- Ein Trigger ist ein benutzerdefiniertes Programm, das automatisch durch einen insert-, delete- oder update-Befehl ausgelöst wird. Es besteht aus
  - Kontrollstrukturen
  - Referenz auf die geänderten Daten
  - Lokalen und globalen Variablen
  - SQL-Befehlen

- Anwendungsgebiete
  - Referentielle und allgemeine Integritätsprüfung
  - Historisierung
  - Automatische Folgeoperationen
  - Nachführen redundanter Daten

Trigger können Kontrollfunktionen wahrnehmen oder Dienstleistungen für den Benutzer zur Verfügung stellen, d.h. die Funktionalität von SQL-Befehlen applikationsorientiert verändern. Einige Beispiele für den Einsatz von Triggern:

- Eine Tabelle stelle einen Komponentenbaum dar. Beim Löschen einer Komponente werden durch den Trigger alle abhängigen Komponenten ebenfalls gelöscht.
- Beim Ändern einer Adresse wird automatisch die alte Adresse in einer History-Tabelle abgelegt.
- Beim Absinken des Lagerbestandes eines Artikels unter eine kritische Grenze wird automatisch eine Nachbestellung vorgenommen.
- Beim Eintragen einer Lektion in den Stundenplan wird automatisch die Lektionensumme des Faches, der Klasse und des Dozenten neu berechnet und in die entsprechenden Tabellen eingetragen.

Trigger können pro SQL-Befehl oder pro betroffenen Datensatz ausgelöst werden.

Der zu einem Trigger gehörende Programmcode kann insert-, delete- oder update-Befehle enthalten, welche wiederum zur Auslösung anderer oder desselben Triggers führen können. Dies kann für absichtliche Rekursionen ausgenützt werden, kann aber auch zu fälschlicherweise nicht-terminierenden Kettenreaktionen führen.

Trigger, Beispiel 1

- Automatisches Nachführen von Attributwerten in einer Tabelle

```
create table Person (
  idPerson        numeric(10,0) default autoincrement,
  kuerzel         varchar(5),
  name            varchar(50),
  lastmodifdate   timestamp null
);
```

```
create trigger setModifControl
after update of kuerzel, name on Person
referencing new as neu
for each row
begin
  update Person
  set lastmodifdate = current timestamp
  where idPerson = neu.idPerson
end;
```

Der Name des Triggers dient nur der Verwaltung, z.B. Löschen mit `drop trigger` *`triggername`*.

Ein Trigger bezieht sich immer auf *eine* Tabelle.

Trigger auf Views sind nicht möglich gemäss SQL-Standard.

Trigger, Beispiel 2

- Beim Ändern der Adresse einer Person sollen die alten Daten automatisch in einer History-Tabelle aufbewahrt werden:

```
create table Adresse (
  idAdresse  numeric(10,0)
  default autoincrement,
  strasse    varchar(50)
);
```

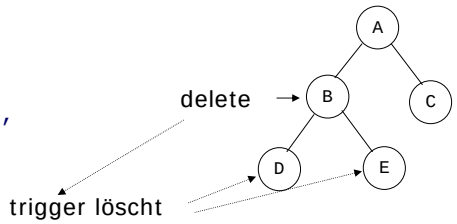
```
create table History (
  idAdresse   numeric(10,0),
  strasse    varchar(50),
  modifdate  timestamp
);
```

```
create trigger AdressChanged
after update of strasse on Adresse
referencing old as alt
for each row
begin
  insert into History (idAdresse, strasse, modifdate )
  values ( alt.idAdresse, alt.strasse, current timestamp )
end;
```

Trigger, Beispiel 3

- Beim Löschen einer Komponente in einem Komponentenbaum werde alle abhängigen Teile gelöscht.

```
create table Component
(
   compID       numeric(10,0),
   parentID     numeric(10,0) null,
   name varchar(100)
)
```




```
create trigger ComponentDeleted after delete on Component
referencing old as alt
for each row
begin
  delete from Component
  where parentID = alt.compID
end;
```

Für obiges Beispiel ist die Tabelle wie folgt belegt:

Component	compID	parentID	name
	1	null	A
	2	1	B
	3	1	C
	4	2	D
	5	2	E

Die Tabelle Component steht für alle Arten von Tabellen, welche eine hierarchische Struktur definieren. Konkrete Beispiele sind:

- Mitarbeiter in einer hierarchischen Organisationsstruktur
- Geräte, welche aus verschiedenen Komponenten bestehen

Allgemeine Triggersyntax

```
create trigger triggername
before | after | instead of
insert | delete | update of columns
on table
order number
referencing reference
for each row | statement
when condition
SQL statements
```

Obige Darstellung ist eine vereinfachte Syntax gemäss SQL-3 Standard. Sie soll hier vorallem für die Erklärung der verschiedenen Elemente eines Triggers dienen.

`triggername`

Name für die Verwaltung des Triggers. Es sind mehrere Triggers pro Tabelle möglich.

`before | after| instead of`

Der Trigger wird vor oder nach der Durchführung des auslösenden SQL-Befehles ausgeführt. Viele Datenbanksysteme kennen nur `after`. Einige DBMS kennen auch die Klausel `instead of`, diese gehört jedoch nicht zum SQL-3 Standard.

Die Aufrufreihenfolge im Rahmen einer SQL-Änderungsoperation ist wie folgt:

1. Before Trigger
2. Referentielle Aktionen
3. Eigentliche Modifikationsoperation
4. Constraint-Prüfung
5. After Trigger
6. Deferred Contraints

`insert | delete | update of` *`columns`*

Die auslösenden Operationen des Triggers. Für Update Operationen können die Attribute angegeben werden. Eine oder mehrere Angaben sind möglich.

`on`

Tabellenname. Trigger auf Views und temporären Tabellen sind nicht erlaubt.

`order`

gehört nicht zu SQL-3, ist aber bei einigen DBMS vorhanden. Bestimmt die Ausführungsreihenfolge bei mehreren Triggern auf derselben Tabelle.

`referencing`

Zu jedem Trigger gehört eine Referenz auf den alten und neuen Datensatz, rsp. bei Triggern, die pro Statement nur einmal ausgelösten werden, je eine interne Tabelle mit den alten und neuen Datensätzen. Die vorgeschriebene Syntax für die `referencing` Klausel ist:

```
old as oldvalue new as newvalue
```

rsp.

```
old_table as oldname new_table as newname
```

In vielen Produkten kommt dieser Teil in der Triggerdefinition nicht vor, da die Namen fest vorgegeben sind.

`when`

Eine Bedingung, die festlegt, ob überhaupt in den Triggercode eingetreten werden soll. Die Bedingung muss determinstisch sein, darf also beispielswese keine Zeitvergleiche mit der aktuellen Zeit enthalten.

`for each row | statement`

Definiert ob der Trigger pro Datensatz oder pro Statement ausgeführt werden soll. Programmtechnisch ist es einfacher, mit einem Statement-Trigger einen Datensatz-Trigger nachzubilden als umgekehrt.

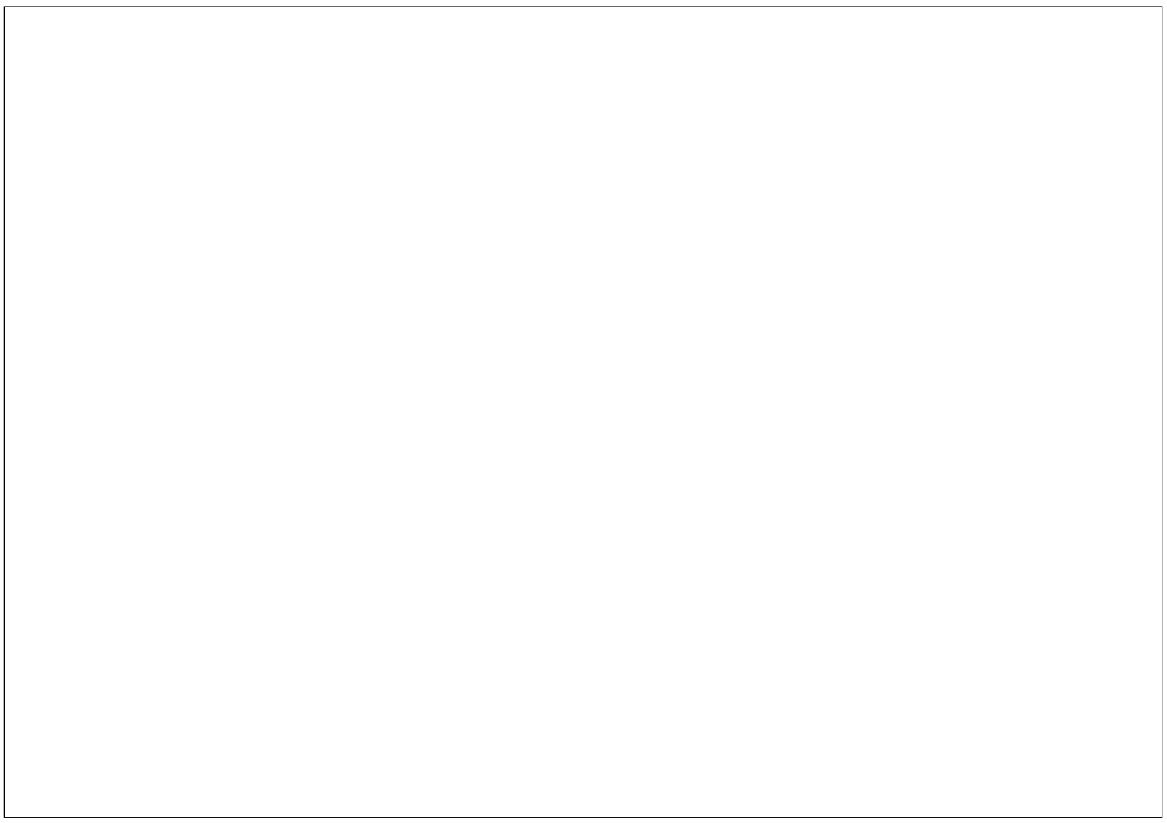
Vergleicht man dieselbe Aufgabe mit Statement-Trigger oder Row-Trigger implementiert, so sind Statement-Trigger klar effizienter in der Ausführung. Auf der anderen Seite erlauben Row-Trigger eventuell einen frühzeitigen Abbruch einer umfangreichen SQL-Operation. Auch in diesem Bereich sind teilweise Einschränkungen durch die Hersteller zu erwarten. So erlaubt beispielsweise Sybase ASA 9.0 keine Statement Triggers mit `before`-Angabe.

Anwendungen Before-Trigger

- Frühzeitiges Testen von Integritätsbedingungen und Abbrechen von unerlaubten Operationen. Beispiel: Eine Reservation für ein Hotelzimmer darf nur gelöscht werden, wenn sie zeitlich bereits vorbei ist. Der Trigger enthält in diesem Fall eine `Rollback [Trigger]` Anweisung.
- Ausführen von Ergänzungen oder Berechnungen für einen Datensatz vor dem Einfügen in die Tabelle. Beispiel:

```
create trigger t1 before insert on Person
referencing new as newP for each row
begin
set newP.kuerzel = substring( newP.name,1,3)
end
```

Die rote Zeile ist ein eigentliches Kernkonstrukt für einen Before-Trigger. Es würde beispielsweise erlauben, intelligente Default-Werte beim Einfügen von neuen Datenwerten zu generieren. Ein Beispiel dazu wäre etwa: Beim Einfügen eines neuen Datensatzes in die Adresstabelle die private Telefonnummer gleich der Geschäftstelefonnummer zu setzen, wenn die Geschäftstelefonnummer ausgelassen wird. Gleichzeitig kann mit einem check-Constraint erzwungen werden, dass eine Geschäftstelefonnummer existiert. Der check-Constraint wird ja erst geprüft, nachdem der before-Trigger bereits abgelaufen und die eigentliche Einfüge- oder Änderungsoperation bereits ausgeführt ist.



Anwendungen Instead Of-Trigger

- Einen delete-Befehl so verfremden, dass ein Datensatz nicht effektiv gelöscht, sondern nur ein deleted-Flag gesetzt wird.
- Einen update-Befehl so verfremden, dass ein Datensatz nicht geändert sondern nur eine neue Version eingefügt wird.
- Einen insert-Befehl so verfremden, dass mehrere Tabellen auf einmal abgefüllt werden können (Trigger auf eine Platzhalter-Tabelle oder eine View setzen, wenn das DMBS dies zulässt).

SQL-Funktionen und -Prozeduren

- *In der Datenbank gespeichert,* von *jeder Art Client* aus aufrufbar.
- Grundkonzept wie in jeder Programmiersprache: Konstrukte für Parameter, Variablen, Schlaufen, Verzweigungen, Operationen, Fehlerbehandlung etc.
- Sehr gute Performance gegenüber Durchführung derselben Aufgabe im Client.
- Funktionen und Prozeduren sind Datenbankobjekte, sie unterliegen also Zugriffsrechten.
- Viele DB-Hersteller erlauben auch in Java geschriebene Funktionen und Prozeduren (SQL-J Standard).

SQL-Funktionen, Anwendung

- Funktionen werden im Rahmen eines `select`, `insert`, `update` oder `delete`-Befehles aufgerufen.
- Anwendungsbeispiele:

```
select name from Person
where alter( gebDatum ) > 20

insert into Person( idPerson, kuerzel, ... )
values ( nextKey(), kuerzel( name, vorname ), ... )
```

Die Funktion alter() muss nur eine einfache Datumsdifferenz berechnen. Die Funktion kuerzel() entnimmt dem Namen und Vornamen ein paar wenige Zeichen und gibt diese als Namenskürzel zurück (Beispiel: Erstes und drittletztes Zeichen des Nachnamens, erstes Zeichen des Vornamens).

Funktionen sind deshalb sehr interessant, weil sie direkt in die SQL-Befehle eingebettet werden können, und weil sie zentral in der Datenbank zur Verfügung stehen.

Für berechnungsintensive Aufgaben sind Funktionen nur geeignet, wenn Sie in einer klassischen Programmiersprache implementierbar sind. Beispielsweise wären Funktionen wie encrypt(), decrypt(), compress(), uncompress() eventuell wünschenswert als SQL-Funktionen. SQL stellt aber nur ungenügende oder wenig performanente Hilfsmittel für die Verarbeitung von Byte-Daten oder Arrays, für komplexe Arithmetik usw. zur Verfügung. Solche Funktionen werden deshalb eher extern in den Applikationen realisiert.

SQL-Funktionen, Definition

Funktionen werden mit `create function` erstellt. Beispiel:

```
create function nextKey()
returns numeric(10,0) not deterministic
begin
  declare v numeric(10,0);
  update KeyValue
  set currentValue = currentValue + 1;
  select currentValue into v
  from KeyValue;
  return v;
end;

-- Hilfstabelle
create table KeyValue ( currentValue numeric(10,0) )
```

Dieses klassische Beispiel ermöglicht es, die Erzeugung von Primärschlüsseln gegenüber dem insert-Befehl zu abstrahieren.

Die Funktion kann beispielsweise wie folgt erweitert werden:

- Mit einem Aufruf können ganze Wertebereiche (z.B. die nächsten 10) für Schlüssel reserviert werden. → Parameter für Anzahl abzuholender Werte.
- Es können mehrere Schlüsselzähler verwaltet werden (z.B. einen pro Tabellenname). → Parameter für Name des Schlüssels und Ergänzung der Tabelle KeyValue um Attribut mit dem Schlüsselnamen.
- Der Primärschlüssel kann in der Funktion um Hostname, Zeitstempel etc. ergänzt werden, wenn beispielsweise ein global eindeutiger Schlüssel generiert werden muss. → Der Returnwert muss vom Typ varchar() o.ä. sein.

Die Angabe `not deterministic` legt fest, dass der Aufruf von nextKey() bei jedem Aufruf einen anderen Rückgabewert generiert, auch wenn die Funktionsparameter nicht ändern, respektive keiner vorhanden ist. Returnwerte von `deterministic` Funktionen können im Cache-abgelegt werden, solche von `not deterministic` Funktionen nicht.

SQL-Prozeduren

- Unterschied zu Funktionen:
  - Eigenständiger Aufruf mit `call` *`procname`*`(` *`parameterliste`* `)`, unabhängig von einem äusseren select-, update-, delete- oder insert-Befehles.
  - Prozeduren können `In` und/oder `out` Parameter haben.
  - Prozeduren haben einen Returnwert (nur für technische Zwecke).
  - Prozeduren können select-Befehle enthalten, derene Resultat. direkt an den Client zurückliefert wird (als ResultSet in Java).

- Anwendungsgebiete:
  - Komplexe Abfragen (Erweiterte Möglichkeiten zu Views)
  - Zur Vermeidung von mehrfachem Code in Triggern: *Eine* Prozedur für mehrere Trigger.
  - Als Interface zu den Tabellen, anstelle direkter SQL-Befehle.

Es gibt Firmen, in denen der Zugriff von Applikationen auf die Datenbank grundsätzlich nur über Prozeduren erlaubt ist. Das Ausführungsrecht auf select, insert, delete, update wird den Datenbankbenützern entzogen, dafür das Ausführungsrecht auf die jeweiligen Prozeduren erteilt.

SQL-Prozeduren, Beispiel

```
create procedure getFaelle( p_status varchar(64) )
begin
if p_status = 'eingegangen' then
  select * from Fall f where f.status = p_status ;
elseif p_status = 'übernommen' then
  select * from Fall f join Mitarbeiter m
   where f.status = p_status ;
elseif  p_status = 'abgeschlossen' then
  select * from Fall f join Kunde k
  where f.status = p_status ;
else
  select 'Fehlerhafter Parameterwert';
end if;
end
```

Mit dieser Prozedur werden Supportfälle, abhängig vom Status abgefragt. Die Prozedur liefert ein ResultSet zurück. Ein Returnwert ist nicht deklariert, könnte jedoch für die Fehlerbehandlung eingesetzt werden.

Transaktionsmodell

- Definition
- ACID Regel

Ein Transaktionsmodell ist notwendig, weil letzlich die phyischen Resourcen bestimmten Einschränkungen unterliegen:

- Der Zeitbedarf für eine Abfrage oder Änderung ist nicht beliebig klein.
- Speichermedien sind nicht beliebig gross und der Zugriff darauf nicht beliebig schnell.
- Programme, Hardware und Kommunikationskanäle können fehlerhaft sein.

Aus diesen Gründen ist eine Zusammenarbeitsvereinbarung zwischen den Clients und der Datenbank notwendig. Das Transaktionsmodell definiert die Grundsätze dieser Zusammenarbeit.

Was ist eine Transaktion

- Aus logischer Sicht ist eine Transaktion ein Arbeitspaket, das einen geschäftlichen Nutzen erzeugt.
  - So klein wie möglich.
  - so gross wie nötig, um alle Integritätsbedingungen einhalten zu können.
- Aus technischer Sicht ist eine Transaktion eine Folge von Lese- und Änderungsoperationen in der Datenbank, mit einem definierten Beginn und einem definierten Abschluss.
- Die Transaktionsverwaltung ist eine der Kernaufgaben eines Datenbanksystems.

ACID-Regel

- Das Datenbanksystem garantiert für eine Transaktion folgende Eigenschaften:

  A Atomarität

  C Konsistenz

  I Isolation

  D Dauerhaftigkeit

  Diese Eigenschaften werden als ACID Regel bezeichnet.

Die Einhaltung der ACID-Regel ist ein zentraler Grundsatz aller Datenbanksysteme.

Atomarität
: Eine Transaktion kann alle gewünschten Operationen durchführen, oder sie hat gar keine Auswirkungen auf den Zustand der Datenbank ("Alles oder Nichts"-Prinzip). In Fehlersituationen kann eine Transaktion durch das Applikations-programm oder durch das DBMS abgebrochen werden. Das DBMS ist dafür verantwortlich, dass alle Änderungen am Datenbestand seit Beginn der Transaktion rückgängig gemacht werden (Undo-Recovery).

Konsistenz
: Bei Abschluss der Transaktion muss ein konsistenter Datenbankzustand vorliegen. Jede im DBMS enthaltene Integritätsregel muss spätestens beim Abschluss der Transaktion erfüllt sein. An einer umfassenden Konsistenzerhaltung ist natürlich auch die Applikation beteiligt, weil es praktisch unmöglich ist, alle Forderungen in Form von Constraints (siehe Folien über Constraints) *in* der Datenbank zu realisieren.

Isolation
: Die Datenbank muss so erscheinen, wie wenn sie jedem Benutzer einzeln gehörte: Laufen mehrere Transaktionen quasi-parallel ab, so müssen die einzelnen Transaktionen so gesteuert werden, dass sie gegenseitig voneinander nichts merken (Wenigstens in Bezug auf den Datenzustand und die Abfrageresultate, sicher nicht vollständig in Bezug auf die Performance).

Dauerhaftigkeit
: Nach Abschluss einer Transaktion sind die von ihr ausgeführten Änderungen gegen alle Arten von Ausfällen gesichert. Auch Prozessabstürze und Plattenfehler dürfen nicht zu Datenverlust führen.

Die Garantie der ACID-Regel bedeutet für den Applikationsprogrammierer eine enorme Erleichterung. Er kann unter allen Umständen von einem korrekten, den Spezifikationen entsprechenden Datenbankzustand ausgehen.

Sowohl relationale wie auch objektorientierte Datenbanken halten sich an die ACID Regel.

Innerhalb einer Transaktion stellt das Datenbanksystem der Applikation Hilfsmittel zur Verfügung, um den Ablauf der Transaktion teilweise rückgängig zu machen, ohne dass gerade die gesamte Transaktion abgebrochen oder wiedeholt werden muss. Im Falle eines Deadlocks kann beispielsweise je nach Systemeinstellung nur der verursachende SQL-Befehles rückgängig gemacht werden.

Auch die Isolationsbedingung kann gelockert werden, wenn ein Prozess beispielsweise mehr Interesse an der Verfügbarkeit als der Korrektheit von Daten hat.

Arbeiten mit Transaktionen

- Jeder lesende oder schreibende Zugriff auf die Datenbank kann nur innerhalb einer Transaktion stattfinden.
- Eine Transaktion beginnt explizit mit einem "begin transaction" Befehl oder implizit mit dem ersten SQL-Befehl.
- Eine Transaktion wird nur mit dem "commit"-Befehl korrekt abgeschlossen. Andernfalls gilt sie noch nicht als korrekt beendet.
- Eine Transaktion kann explizit mit "rollback" oder implizit durch ein äusseres Ereignis abgebrochen werden.

Beispiel einer korrekten Transaktion:

```
/* beginn der Transaktion durch ersten select-Befehl */
select * from Person
delete from Person
where name = 'Müller'
commit
/* Alle Änderungen sind nun definitiv */
```

Beispiel einer durch den Benutzer abgebrochenen Transaktion:

```
delete from Person
where name = 'Müller'
rollback /* Änderungen werden rückgängig gemacht */
```

Eine Transaktion kann aus verschiedenen Gründen durch das Datenbanksystem zwangsweise abgebrochen werden:

- Deadlock von zwei Benutzerprozessen
- Verletzung von Zugriffsrechten oder Integritätsbedingungen
- Crash rsp. Verbindungsabbruch des Benutzerprozesses

Das Datenbanksystem liefert dem Benutzer einen Fehlerstatus zurück. Diesen auszuwerten und ev. einen fehlgeschlagenen SQL-Befehl oder eine Transaktion zu wiederholen, ist Sache der Applikation.

Auch das reine Lesen von Daten kann ausdrücklich nur innerhalb einer Transaktion geschehen. Es werden zwar keine Daten verändert, der Anfangs- und der Endzustand sind daher derselbe, aber es wird ausdrücklich verlangt, dass ein *konsistenter* (korrekter) Zustand gelesen wird. Damit dies vom Datenbankmanagementsystem auch bei mehreren konkurrenzierenden Benutzern der Datenbank richtig gehandhabt werden kann, müssen z.B. Lesesperren auf die gelesenen Daten gesetzt werden. Dies beeinflusst wiederum schreibende Transaktionen, die warten müssen, bis die Lesetransaktionen beenden und damit (implizit) ihre Sperren freigeben.

Letzlich kann nur der Datenbankbenutzer entscheiden, wann alle Änderungen oder Abfragen, die zu einem korrekten Zustandsübergang gehören, ausgeführt sind. Das Absetzen des commit-Befehles ist daher Sache des Datenbank-Benutzers (Clients) und nicht des Datenbanksystems. Das Datenbanksystem kann allenfalls prüfen, ob gewisse Integritätsregeln verletzt sind und die Transaktion zwangsweise abbrechen und zurücksetzen. Beispiel: Gemäss ERD wird verlangt, dass zu einer Person immer mindestens eine Adresse gehört. Das Eingeben einer neuen Person mit einer oder mehrerer Adressen ist nun Sache des Benutzers. Das DBMS kann nicht selber Adressen zu einer Person erzeugen. Es kann lediglich bei Abschluss der Transaktion prüfen ob mindestens eine Adresse da ist. Im allgemeinen gilt also auch:

Transaktion ≠einzelner SQL-Befehl

Allerdings arbeiten viele Datenbanksysteme per Default in einem "Autocommit"-Modus, d.h. nach jedem SQL-Befehl wird durch das DBMS ein commit-Befehl ausgelöst. In sehr vielen Fällen ist das jedoch im Sinne des Datenmodells falsch und kann zu einem scheinkorrekten Zustand der DB führen.

Eine Datenbank muss sich jederzeit in einem konsistenten (korrekten) Zustand befinden. Dies ist der Fall, wenn

- sich ihre Datenwerte mit allen Integritätsbedingungen vertragen,
- ihre Datenwerte mit der gegenwärtigen Realität übereinstimmen,
- alle relevanten Daten vollständig in der Datenbank vorhanden sind.

Verletzungen der ersten Bedingung haben meist technische Ursachen, z.B. Speicherüberlauf, Systemabsturz, Disk-Crash, Hardware-Fehler, logische Fehler in der Applikation.
Die beiden letzten Bedingungen können nur über organisatorische Massnahmen garantiert werden, es sei denn, die Datenbank arbeitet beispielsweise mit einem automatisierten Produktionssystem zusammen.

Eine Transaktion wird immer über eine Verbindung (Session) mit der Datenbank abgewickelt. Eine Verbindung kann gleichzeitig nur eine Transaktion bedienen, und ein Verbindungsabbruch hat immer einen Transaktionsabbruch zur Folge. Über eine Verbindung werden nacheinander eine oder mehrere Transaktionen abgewickelt. Eine Transaktion sollte möglichst rasch abgewickelt werden, da sie immer Resourcen reserviert, welche anderen Transaktionen nicht zur Verfügung stehen.

Jede Transaktion ist im DBMS meist ein eigener Thread. Die Transaktions-Threads konkurrieren um die vorhanden Resourcen.

Concurreny Control

- Zweck
- Serialisierbarkeit
- Locking
- Deadlock

Zweck

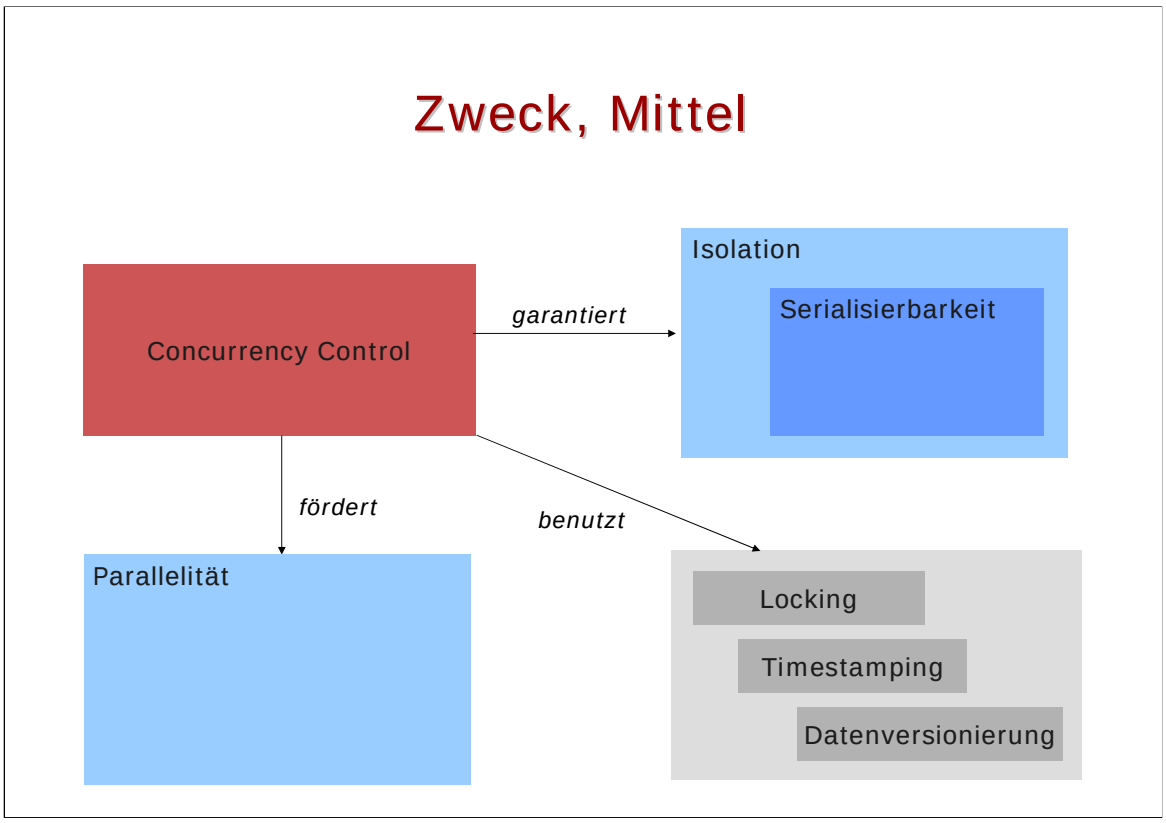
- Einerseits: *Isolation*
  - Änderungen am Datenbestand dürfen erst bei Transaktionsabschluss für andere sichtbar sein.
- Andererseits: *Parallelität*
  - Eine Datenbank muss mehrere Benutzer(-prozesse) gleichzeitig bedienen können und es sollen möglichst wenig Wartesituationen entstehen.

→ Realisierungsstrategie

Die parallele Ausführung von Transaktionen muss bezüglich Datenzustand und bezüglich Resultat-ausgabe zum Client identisch mit der seriellen Ausführung von Transaktionen sein.

Die beiden Anforderungen *Isolation* und *Parallelität* arbeiten im Prinzip gegeneinander, wobei der Isolation die höhere Priorität zukommen muss. Ein sehr einfache Realisierung der Isolationsbedingung wäre es, die Datenbank exklusiv für die gerade laufende Transaktion arbeiten zu lassen. Alle anderen Transaktionen müssten warten, bis die gerade laufende fertig ist. Alle Transaktionen werden also faktisch hintereinandergeschaltet oder *serialisiert*.

Eine schlauere Datenbank wird versuchen, von allen Transaktionen den jeweils nächsten anstehenden SQL-Befehl entgegenzunehmen, und davon immer denjenigen auszuführen, dass man nach am Schluss aller Transaktionen sagen kann: Die abwechslungsweise Ausführung von SQL-Befehlen ist auf dasselbe herausgekommen, wie wenn alle Transaktionen streng hintereinander ausgeführt worden wären. Damit ist sowohl die Isolationsbedingung, wie die Forderung möglichst hoher Parallelität erfüllt. Die Datenbank muss also für *Serialierbarkeit* sorgen. *Serialisiert* und *serialisierbar* unterscheiden sich lediglich in der Art der Ausführung, nicht in Bezug auf die Auswirkung auf die Daten.



Isolation ist ein allgemeine Anforderung, die an die Datenbank gestellt wird. Sie ist quantitativ schwer fassbar. Serialisierbarkeit ist *eine* konkrete Art der Isolation, welche verhältnismässig einfach zu implementieren ist, und deshalb in den marktüblichen Datenbanksystemen die einzig vorkommende.

Das Concurrency-Control ist der *Mechanismus*, welcher die Isolation und die Parallelität sicherstellt. Locking, Timestamping oder Datenversionierung sind mögliche *Werkzeuge*, welche das Concurrency Control für seine beiden Hauptaufgaben benutzt.

Serialisierbarkeit

Jeder parallele Ablauf mehrerer Transaktionen muss inhaltlich einem seriellen Ablauf entsprechen. Beispiel:

Transaktion 1

1.1 select *:persnr* from Person
where name = "Schmid"

1.2 delete from Adressen
where persnr = *:persnr*

1.3 delete from Person
where persnr = *:persnr*
commit

Transaktion 2

2.1 select *:persnr* from Person
where name = "Schmid"

2.2 select * from Adressen
where persnr = *:persnr*
commit

Ein korrekter Ablauf im Sinne obiger Definition ist:

1.1 → 2.1 → 2.2 → 1.2 → 1.3

Serialisierbarkeit ff

Unter der Annahme, dass die Datenbank keine Synchronisationsmittel einsetzt und jedes SQL-Statement ein atomarer Schritt ist, sind verschiedene zeitliche Abläufe der beiden Transaktionen denkbar:

1.	1.1 → 2.1 → 1.2 → 2.2 → 1.3	(f)
2.	1.1 → 2.1 → 1.2 → 1.3 → 2.2	(f)
3.	1.1 → 2.1 → 2.2 → 1.2 → 1.3	(k)
4.	1.1 → 1.2 → 2.1 → 1.3 → 2.2	(f)
5.	1.1 → 1.2 → 2.1 → 2.2 → 1.3	(f)
6.	1.1 → 1.2 → 1.3 → 2.1 → 2.2	(s)
7.	2.1 → 1.1 → 1.2 → 2.2 → 1.3	(f)
8.	2.1 → 1.1 → 2.2 → 1.2 → 1.3	(k)
9.	2.1 → 1.1 → 1.2 → 1.3 → 2.2	(f)
10.	2.1 → 2.2 → 1.1 → 1.2 → 1.3	(s)

Die beiden mit (s) gekennzeichneten Abläufen entsprechen dem vollständigen Nacheinander beider Transaktionen (serialisierte Abläufe). Alle anderen Abläufe müssen bezüglich Ausgabe zum Prozess, rsp. bezüglich Datenbankzustand einem der beiden serialisierten und damit korrekten Abläufe entsprechen (k), sonst sind sie falsch (f).

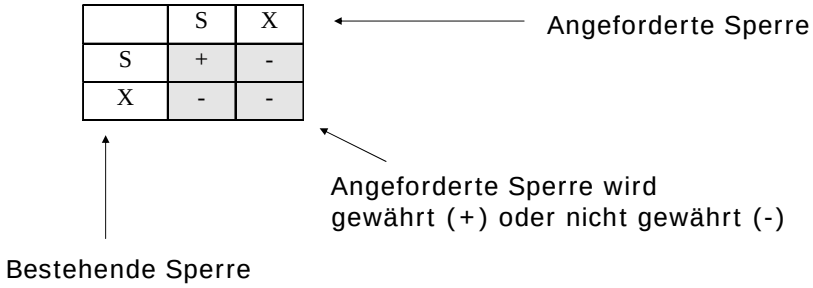
Das Datenbanksystem muss Synchronisations-Mittel besitzen und einsetzen, damit

- eine hohe Parallelität gewährleistet ist.
- keine falschen Abläufe möglich sind (garantierte Serialisierbarkeit).

Serialisierbarkeit löst die Frage nicht, in welcher Reihenfolge die Transaktionen am besten ausgeführt werden, sondern gibt nur an, dass eine parallele Ausführung keine anderen Resultate als eine serielle Ausführung hat. Die durch die sequenzielle Ausführung einer Reihe von Transaktionen erzielten Ergebnisse gelten alle als richtig.

Locking

- Locking ist die häufigste Möglichkeit, die Serialisierbarkeit zu gewährleisten.
  - Für das Lesen eines Datensatzes wird ein S-Lock gesetzt
  - Für das Ändern, Löschen oder Einfügen eines Datensatzes wird ein X-Lock gesetzt.
- Die gesetzten Locks sind gemäss einer Verträglichkeitstabelle untereinander kompatibel oder nicht:



Das Datenbanksystem setzt für jeden SQL-Befehl automatisch entsprechende Sperren auf den ausgewählten Datenelementen.

Schreibsperren werden bis zum Transaktionsende gehalten. Eine vorherige Freigabe würde die Isolationsbedingung verletzen: Vor dem Transaktionsabschluss ist nicht garantiert, dass Änderungen nicht noch rückgängig gemacht werden, sei es durch den Client (Rollback-Befehl) oder den Datenbankserver (Crash, Deadlock). Schreibsperren können nicht umgangen oder ausser Kraft gesetzt werden.

Lesesperren werden ebenfalls bis zum Transaktionsende gehalten. Ein Benutzerprozess kann daher sicher sein, dass einmal eingelesene Daten in der Datenbank zwischenzeitlich nicht geändert werden. Im Gegensatz zum Sperren von Daten beim Schreiben sind aber auf Wunsch des Programmierers oder des Datenbank-Administrators verschärfte oder erleichterte Sperren beim Lesen möglich: Es dürfen X-Locks zum Lesen gesetzt werden oder man kann ohne Sperren lesen (Dirty Read). Bei gewissen Datenbanksystemen können Lesesperren auch unmittelbar nach dem Lesen, anstatt erst bei Transaktionsende zurückgegeben werden (Short Locks).

Jedes Datenbanksystem hat eigene Feinheiten in der Locking Strategie, die es bei einer stark gebrauchten Datenbank zu beachten gilt, und die Ausgangspunkt für Tuning-Massnahmen sind. Einige Spezialitäten sind im folgenden beschrieben.

Spezialitäten

Mit D-Locks (Sybase, SQL-Server) kann verhindert werden, dass zeitlich überlappende Lesesperren einen schreibwilligen Prozess dauernd vom Zugriff ausschliessen. Ein D-Lock wird von einem Schreiber angefordert, und sobald alle Lesesperren verschwunden sind, in einen X-Lock umgewandelt.

	S	X	U	D	null
S	+	-	+	+	+
X	-	-	-	-	+
U	+	-	-	-	+
D	-	-	-	-	+
null	+	+	+	+	+

Update-Locks (U) werden für die Deadlock-Verhinderung eingesetzt. U ist mit S verträglich, aber nicht mit anderen U und X. Sobald tatsächlich die gelesenen Daten geändert werden, wird U in X konvertiert. Dies ist nur möglich, wenn keine anderen Lesesperren vorhanden sind. Update-Locks sind insbesonde für das Arbeiten mit Cursorn geeignet: Die Datenbank liest die potentiell zu modifizierenden Daten relativ rasch ein, der genaue Zeitpunkt des Updates ist aber durch den Datenbank-Client bestimmt:

```
declare c cursor for
select persnr
from person
for update of adresse;

/* Records werden mit U statt mit S gesperrt.   */

update person
set adresse = "neu"
where current of c;  /* Hier wird U in X umgewandelt. */
```

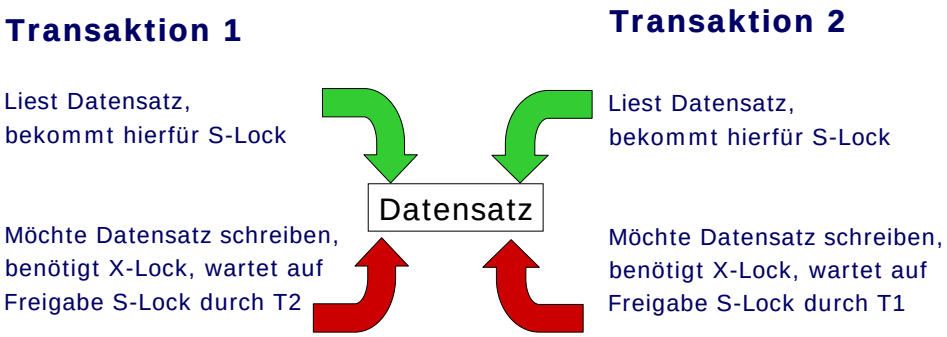
Null-Locks werden für unsicheres Lesen (Dirty Read) eingesetzt. Ein Null-Lock ist gar keine Sperre und ist daher mit allen anderen Locks verträglich.

Der Zugriff auf die Indextabellen ist ebenfalls zu berücksichtigen, es gelten meist diesselben Regeln wie auf den eigentlichen Datentabellen.

Neben den "High-Level" Locks für die eigentlichen Daten, die durch das DBMS wie oben beschrieben gehandhabt werden, kommen für den Zugriff auf Hilfsresourcen (z.B. die Locktabelle selbst) auch die Synchronisations-Mechanismen des Betriebssystems zum Einsatz, beispielsweise Semaphore, Mutex-Variablen oder Spin Locks bei Multiprozessor-Systemen.

Deadlock

- Beim Arbeiten mit Locks können so genannte Deadlocks auftreten. Deadlocks sind in der Informatik ein allgemein bekanntes Problem:



Gegenseitiges Warten = Deadlock

Deadlocks bei der Serialisierung

- Einige der Abläufe unter 'Serialisierbarkeit' erzeugen einen Deadlock. Der Deadlock ist konzeptionell gesehen nicht ein Fehler, sondern bedeutet:
  - Es gibt keinen Weg mehr, die anstehenden Transaktionen so zu steuern, dass ein serialisierbarer Ablauf entsteht.
  - Eine der beteiligten Transaktionen wird zurückgesetzt, so dass für die übrigen wieder die Chance besteht, gemäss Serialisierbarkeitsprinzip abzulaufen.

Mögliche Deadlocks sind der Preis für die Anforderung hoher Parallelität. Man kann die Serialisierbarkeit auch erreichen ohne Deadlock-Bedrohung, indem die involvierten Tabellen oder Datensätze zum vornherein exklusiv gesperrt werden. Die meisten Datenbankssysteme kennen entsprechende Befehle, beispielsweise: `LOCK TABLE tablename in EXCLUSIVE MODE`. Damit ist aber jede Parallelität von Transaktionen verhindert.

Zwei-Phasen Sperrprotokoll

1. Bevor eine Transaktion auf einem Datensatz aktiv wird, muss sie eine Sperre (S oder X) erwerben.

2. Sperren werden beim Commit zurückgegeben.

-> Dieses Protokoll garantiert einen serialisierbaren Ablauf von parallelen Transaktionen, ohne Berücksichtigung des Phantom-Problems.

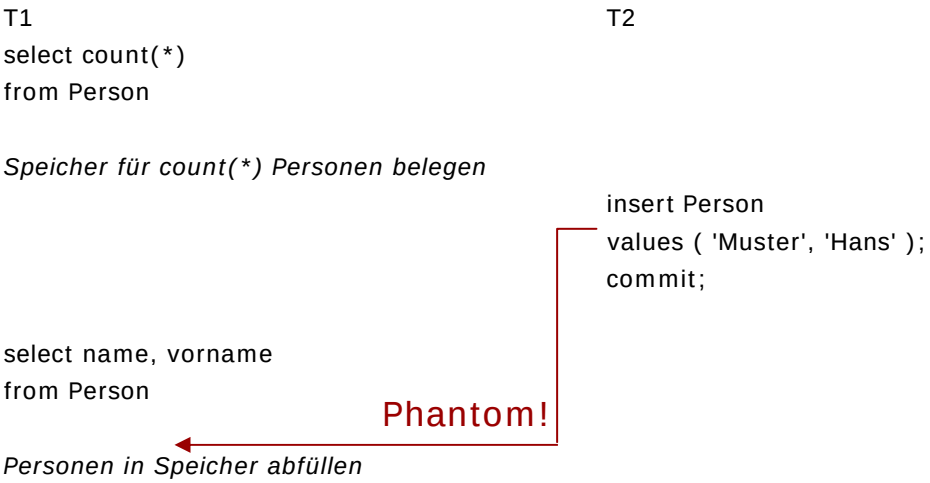
-> Das Protokoll ist nicht Deadlock-frei.

Der Kurzname 2PL ist aus 'Two-Phase Locking' abgeleitet.

Die Phase 1 entspricht praktisch der Zeit während der Lese- und Änderungsoperationen durch die Transaktion durchgeführt werden.

2PL garantiert einen serialisierbaren Ablauf, wenn nur auf *vorhandenen* Datensätzen gelesen oder geändert wird. Betrachtet man die zusätzliche Situation, dass von anderen Transaktionen Datensätze *eingefügt* werden, die unter Umständen einer Auswahlbedingung der ersten Transaktion genügen, können Konsistenzprobleme auftreten (siehe Folie über Phantome).

Phantomproblem



Sperren auf Tabellenebene

- Um Serialisierbarkeit auch für Transaktionen zu erreichen, welche Datensätze *einfügen*, werden Sperren auf *Tabellenebene* verwendet (→ Phantomproblem).
- Minimalkonzept: Eine Lesesperre S auf der Tabelle verhindert das Einfügen von neuen Datensätzen.
- In Datenbanken werden heute meist Range-Locks zur Vermeidung von Einfügekonflikten verwendet: Ein Range Lock sperrt nur einen kritischen Bereich der Tabelle.

Sperren auf Tabellen-Ebene kommen in folgenden Fällen zur Anwendung:

- Mit einer Sperre auf Tabellen-Ebene (S oder X) kann das Einfügen neuer Datensätze durch andere Transaktionen verhindert werden (Mit Sperren auf Datensatz-Ebene ist dies nicht möglich). Das schliesst mögliche Konsistenzprobleme aus (siehe Folie über Phantome). Besitzt eine Transaktion eine S-Sperre auf einer Tabelle können keine X-Sperren auf Datensätzen von anderen Transaktionen erworben werden. Besitzt eine Transaktionen X-Sperren auf einzelnen Datensätzen, kann eine andere Transaktion keine S-Sperre auf der ganzen Tabelle erwerben.
- Bei SQL-Befehlen, die sonst falsche Ergebnisse oder inkonsistente Zustände zur Folge hätten, wie create index, create constraint, grant, revoke, drop table.
- Wenn mehr als eine vorgesehene Anzahl Datensätze gesperrt werden müssen, und damit die interne Verwaltung zu gross würde, eskaliert das DBMS die Einzelsperren auf die ganze Tabelle. Die Sperren auf den Datensätzen können dann freigegeben werden.
- Wenn der Benutzerprozess oder der Administrator dies aus applikatorischen Gründen wünscht. Ein X-Sperre auf Tabellen-Ebene garantiert dem Besitzer lesenden und schreibenden Zugriff auf sämtliche Datensätze der Tabelle ohne jegliche Deadlock-Gefahr.

Anstelle eines S-Lock auf der ganzen Tabelle hat sich in letzter Zeit der Range-Lock durchgesetzt: Es wird via Index nur derjenige Bereich von Datensätzen gesperrt, der durch die where-Bedinung in einem SQL-Befehl definiert ist. Datensätze ausserhalb dieses Bereichs kommen für Konflikte gar nicht in Frage und müssen daher auch nicht gesperrt werden. Range-Locks sind umso günstiger, je mehr Datensätze mit unterschiedlichen Werten vorhanden sind. Die einzige Bedingung ist das Vorhandensein eines Index und einer where-Bedingung, welche die gesuchten Daten wesentlich einschränkt.

Isolationsgrade

- Je nach Applikation kommen gewisse SQL-Befehlsfolgen, welche die Serialisierbarkeit gefährden, nicht vor. Unter Umständen will man auch die vollständige Serialisierbarkeit aufgeben, zugunsten einer erhöhten Parallelität.
- SQL-3 definiert deshalb verschiedene Isolations*grade beim Lesen* von Daten:

SERIALIZABLE	3
REPEATABLE READ	2
READ COMMITTED	1
READ UNCOMMITTED	0

Der Befehl zum Setzen des Isolationsgrades ist:

`set transaction isolation level` *konstante* | *zahl*

SERIALIZABLE

Der Modus SERIALIZABLE garantiert die Serialisierbarkeit einer Transaktion. Die Implementation basiert im einfachsten Fall auf einer Lese-Sperre der ganzen Tabelle. Einfügungen neuer Datensätze (Phantom-Problem) werden damit verhindert. Diese Implementation lässt immer noch andere Leseprozesse zu, eine gewisse Parallelität ist also gewährleistet. Deadlocks können auftreten, wenn zwei Transaktionen je eine Lesesperre auf der Tabelle besitzen und anschliessend innerhalb der Tabelle Änderungen vornehmen wollen. Der zu erwerbende X-Lock für einen Datensatz verträgt sich per Definition nicht mit dem S-Lock einer anderen Transaktion auf der Tabelle. Die meisten Datenbankssysteme bieten zusätzlich Lock-Befehle an, mit denen man die ganze Tabelle exklusiv sperren kann. Damit ist die relativ hohe Deadlock-Gefahr gebannt und die Serialiserbarkeit garantiert, aber auch jede Parallelität verunmöglicht. Der entsprechende Befehl hat häufig folgende Form: `LOCK TABLE tablename in EXCLUSIVE MODE`.

SERIALIZABLE			
Befehl	Sperre auf Datensatz	Sperre auf Tabelle	Sperrdauer
select	-	S	bis commit
update	X	-	bis commit
delete	X	-	bis commit
insert	X	-	bis commit
create index	-	X	bis commit
alter table	-	X	bis commit

Zu beachten: Ein X-Lock auf einen Datensatz kann nur gesetzt werden, wenn kein S-Lock auf die Tabelle besteht und umgekehrt.

Ein Modus SERIALIZABLE arbeitet mit Range Locks (siehe Anhang). Ein Range-Lock sperrt logische Bereiche (Die Bedingung in der where-Klausel von SQL-Befehlen), statt einzelne Datensätze.

REPEATABLE READ

Das wiederholte Absetzen desselben Abfrage-Befehles liefert dasselbe Resultat. Einmal in die Applikation gelesene Daten können also durch andere nicht verändert werden. Realisierungsmöglichkeit 1: Lesesperren, die bis zum Transaktionsende gehalten werden.

REPEATABLE READ			
Befehl	Sperre auf Datensatz	Sperre auf Tabelle	Sperrdauer
select	S	-	bis commit
update	X	-	bis commit
delete	X	-	bis commit
insert	X	-	bis commit
create index	-	X	bis commit
alter table	-	X	bis commit

READ COMMITTED

Es besteht die Anforderung, dass nur bestätigte Daten gelesen werden, d.h.solche die sich nicht in Bearbeitung befinden. Anwendungsbeispiel: Management- Informationsysssteme- und statistitische Auswertungen. Realisierungsmöglichkeit 1: Eine Abfrage gibt ihre Lesesperren unmittelbar nach dem Lesevorgang zurück, statt sie bis zum Transaktionsende zu behalten. Eine andere Transaktion kann damit die Daten verändern. Nachteil: Schreibsperren können die Abfrage immer noch blockieren. Realisierungsmöglichkeit 2: Der letzte gültige Zustand eines Datenelementes wird aufbewahrt für die Lesetransaktionen (Siehe Concurrency Control mit Versionen). Realisierungsmöglichkeit 3: Zeitstempelverfahren (siehe Folie über 'Concurrency Control mit Zeitstempeln').

READ COMMITTED			
Befehl	Sperre auf Datensatz	Sperre auf Tabelle	Sperrdauer
select	S	-	nur select
update	X	-	bis commit
delete	X	-	bis commit
insert	X	-	bis commit
create index	-	X	bis commit
alter table	-	X	bis commit

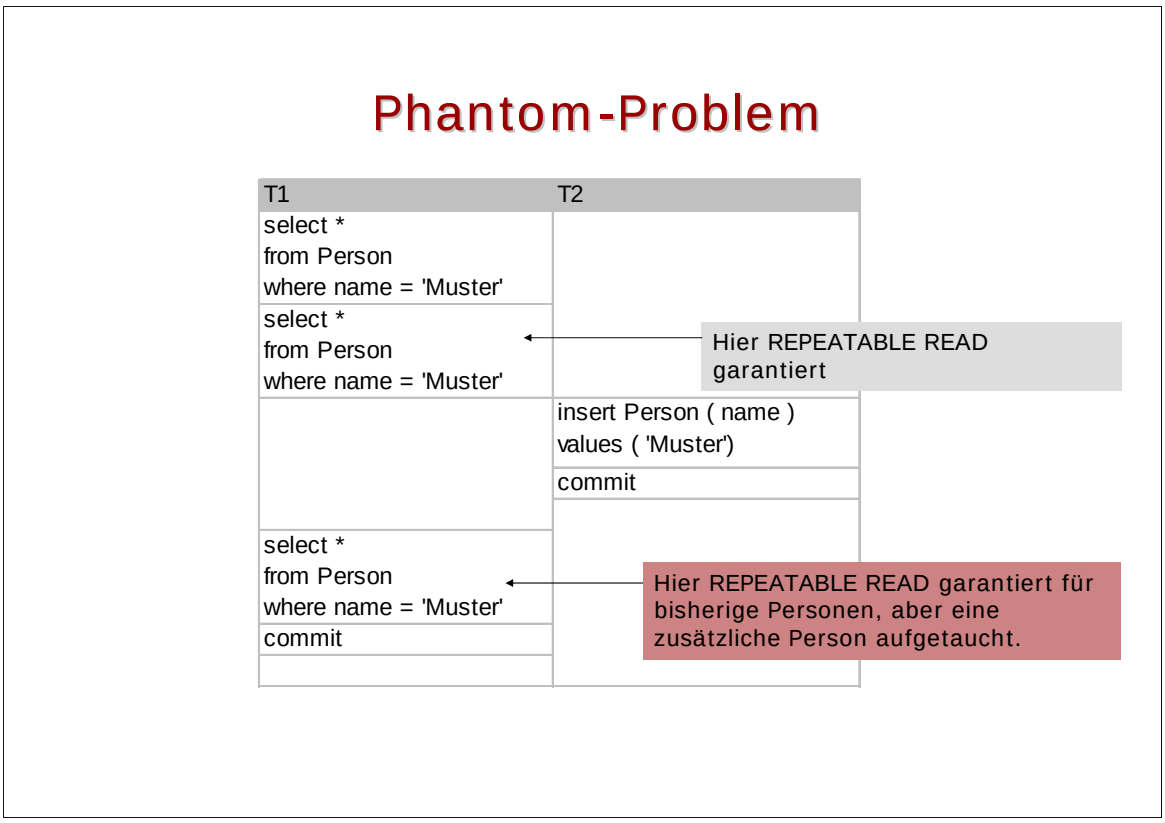
Es dürfen unbestätigte Daten gelesen werden. Realisierung 1: Es wird mit Null-Locks gearbeitet. Vorteil: es gibt mit Sicherkeit keine Wartesituationen für die eigene oder andere Transaktionen. Nachteil: Es können nicht bestätigte Daten gelesen werden, d.h. solche, die von einer anderen Transaktion ev. wieder zurückgesetzt werden.

READ UNCOMMITTED			
Befehl	Sperre auf Datensatz	Sperre auf Tabelle	Sperrdauer
select	-	-	-
update	X	-	bis commit
delete	X	-	bis commit
insert	X	-	bis commit
create index	-	X	bis commit
alter table	-	X	bis commit

Mögliche Inkonsistenzen

- Die Abschwächung des Isolationsgrades hat den Vorteil, dass die Parallelität erhöht wird.
- Die Abschwächung des Isolationsgrades hat den Nachteil, dass gewisse Inkonsistenzen auftreten können, resp. in Kauf genommen werden:

`SERIALIZABLE`	keine Inkonsistenzen
`REPEATABLE READ`	Phantome möglich
`READ COMMITTED`	Lost Updates möglich
`READ UNCOMMITTED`	Lesen unbestätigter Daten möglich

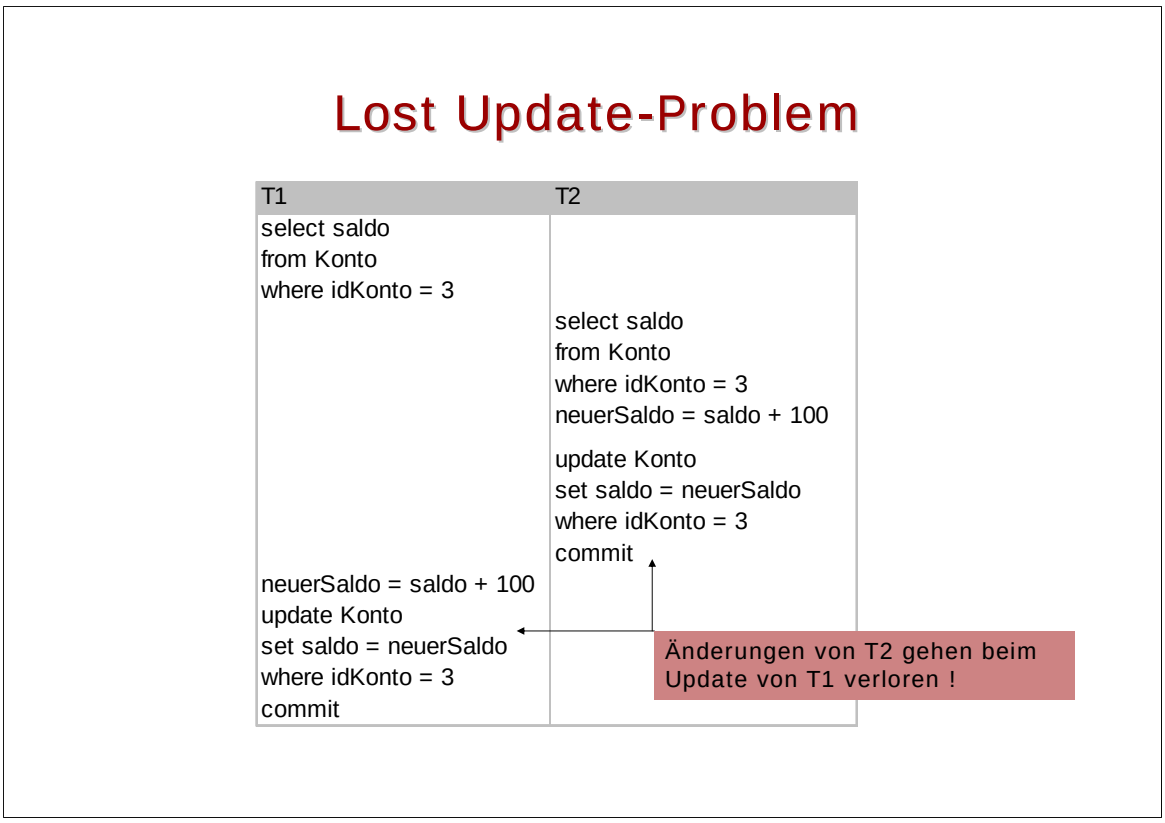


Phantom-Problem

Eine erste Transaktion selektiert Datensätze nach bestimmten Kriterien. Eine zweite Transaktion fügt einen neuen Datensatz ein und committet diesen. Wenn die erste Transaktion ihre Abfrage nochmals durchführt, findet sie den neuen Datensatz. Der neue Datensatz wird bezüglich der ersten Transaktion als Phantom bezeichnet.

Phantome können zu Schwierigkeiten bei statistischen Auswertungen führen. Phantome können auch zu Programmierproblemen führen, wenn beispielsweise eine Abfrage abgesetzt wird, um die Anzahl Datensätze zu zählen, dann aufgrund der Zählung Speicherplatz für die zu erwartenden Datensätze bereitsgestellt wird. Wenn in der zweiten Abfrage für die eigentlichen Datensätze dann plötzlich zusätzliche Datensätze auftreten, kann dies zu Speicherüberläufen führen.

Ein Phantom-Problem kann auch beim Löschen von Datensätzen auftreten. Eine erste Transaktion löscht einen bestimmten Datensatz, führt aber noch kein commit durch. Eine andere Transaktion liest alle Datensätze der Tabelle. Die erste Transaktion führt ein Rollback durch. Die zweite Transaktion liest die Tabelle noch einmal und sieht nun den vorher gelöschten Datensatz. Aufgrund der Implementation der Löschoperation bei vielen Datenbanksystemen ist jedoch dieser Fall meist nicht möglich, da gelöschte Datensätze in der Tabelle verbleiben, allerdings mit einer Schreibsperre versehen, und der Leseprozesse beim Zugriff auf diesen Datensatz warten muss, bis die löschende Transaktion entweder ein Commit oder ein Rollback durchführt.



Lost Update-Problem

Wenn die gelesenen Daten verändert und anschliessend wieder in die Datenbank zurückgeschrieben werden kann das Problem auftreten, dass eine andere Transaktion in der Zwischenzeit die Daten ebenfalls gelesen, verändert und bereits committet hat. Die erste Transaktion wird dann beim Zurückschreiben ihrer Daten die Änderungen der anderen Transaktion zunichten machen. Der Update der anderen Transaktion geht damit verloren.

Lange Transaktionen

- Kurze vs lange Transaktion
- Checkout/Checkin-Verfahren
- Zeitstempel/Prüfregel-Verfahren

Kurze und lange Transaktionen 1

Bisher: Kurze, technische Transaktionen

- Oberstes Ziel ist ein serialisierbarer Ablauf
- Keine spezifische Semantik bei Konflikten, sondern Abbruch der Transaktion.
- Keine Benutzerinteraktion *während* der Transaktion (Alle Informationen und Regeln zur Durchführung der Transaktion sind vorgängig bekannt)
- Minimalen Zeitbedarf anstreben.

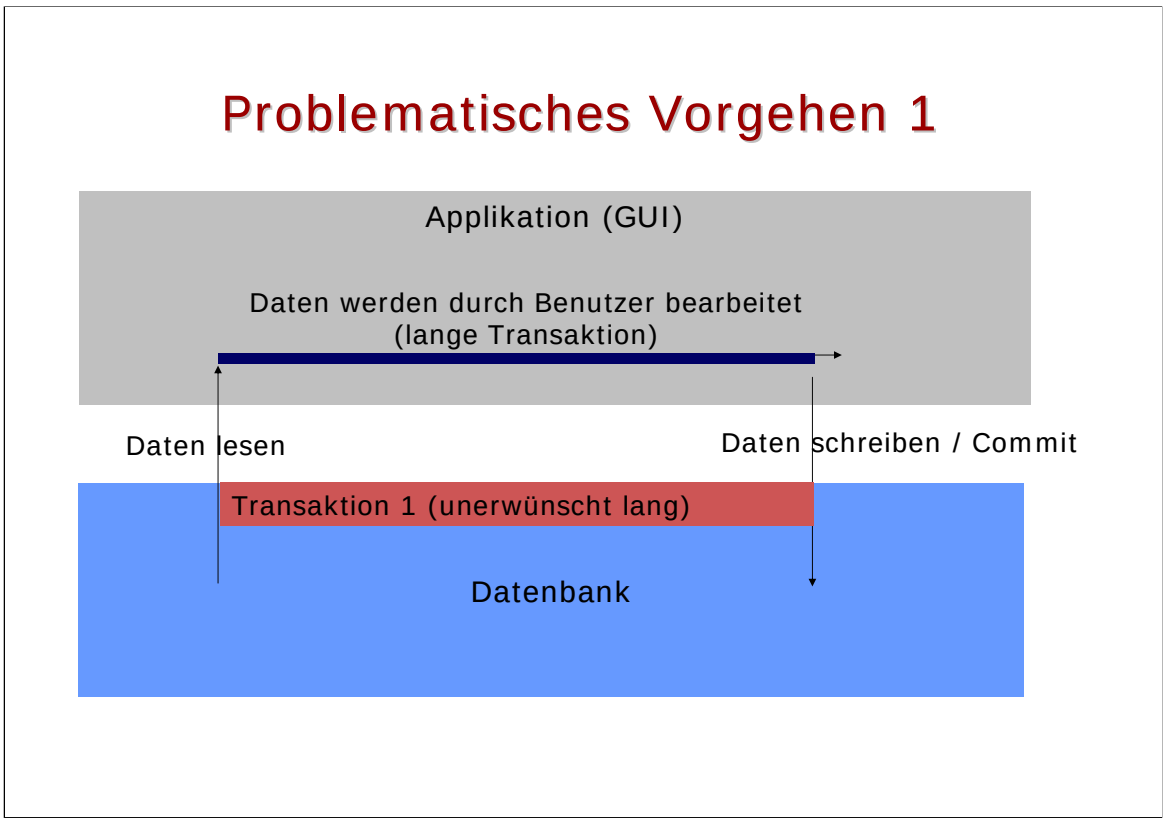
Die technischen Transaktionen haben eigentlich keine Semantik, sondern sind lediglich ein Konstrukt, um zu verstecken, dass der Zugriff auf Daten eine endliche Zeit beansprucht und mit beschränkter Sicherheit stattfindet. Wären Applikation, Netzwerk und Datenbank unendlich schnelle und unendlich sichere Resourcen, bestünde kein Bedarf an einem technischen Transaktionskonzept. Jede Transaktion wäre momentan begonnen und momentan beendet und damit in jedem Fall komplett *vor* oder komplett *nach* einer anderen Transaktion durchgeführt. Die Serialisierbarkeit wäre also gewährleistet.

Kurze und lange Transaktionen 2

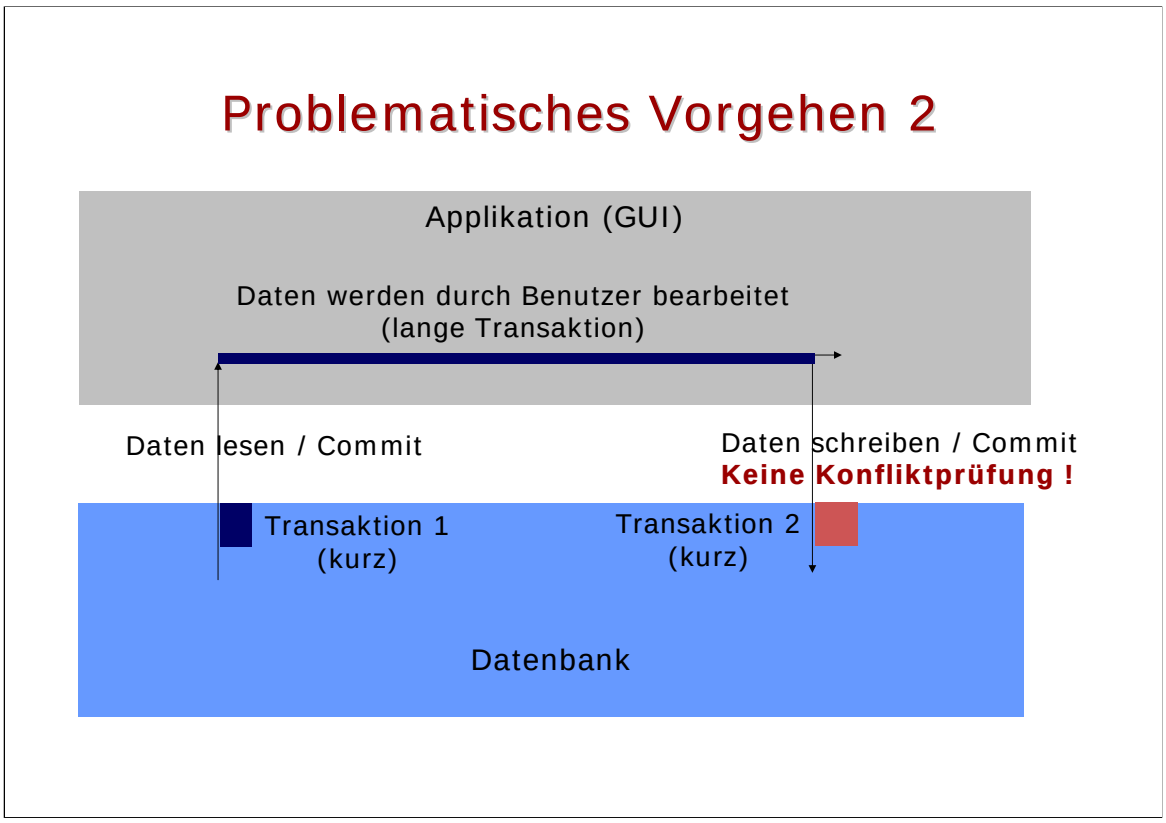
Neu: Lange, logische Transaktionen

- Daten aus DB entnehmen für Applikation
- Bearbeitung in der Applikation beliebig lange
- Benutzerinteraktion während der Transaktion muss möglich sein, da nicht alle Informationen oder Regeln explizit bekannt sind.
- Zurückschreiben in die Datenbank
- Basis für Entnahme und Zurückschreiben: Kurze Transaktionen.

Eine lange Transaktion entspricht eigentlich der Realisierung eines Use Cases gemäss UML.



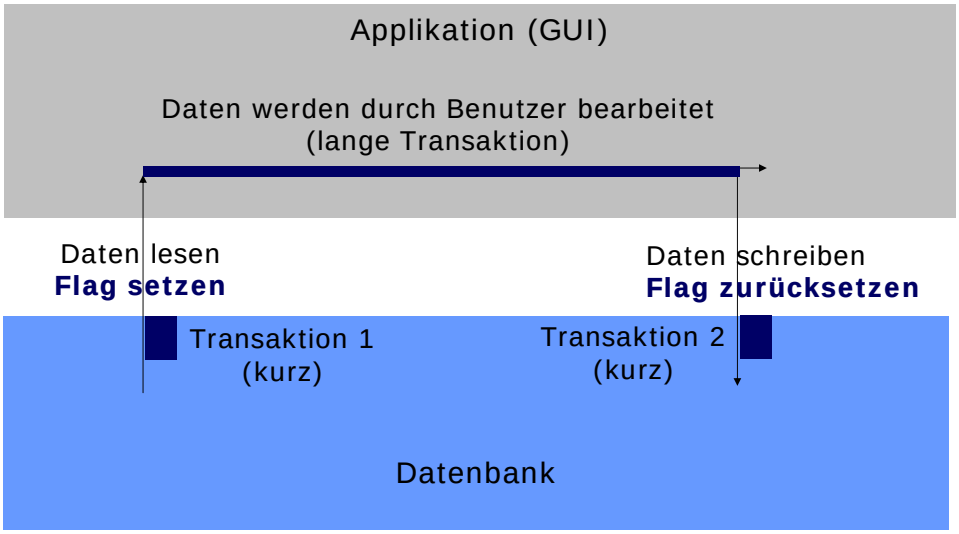
Nur für den Fall wo der "Benutzer" ein anderer Computer ist, der seine Aufgabe in kurzer Zeit löst, kann mit einer einzigen Transaktion gearbeitet werden. Für den Fall, wo die gelesenen Daten durch einen menschlichen Benutzer bearbeitet werden, und die benötigte Zeit undefinierbar lang ist, müssen zwei unterschiedliche Transaktionen verwendet werden für das Lesen der Daten und das letztendliche Zurückschreiben. Eine kurze Zeit sei hier definiert als eine Zeitspanne innerhalb der es akzeptiert ist, dass es zu gewissen Resourcen-Blockierungen (Locks) kommt, welche den Zugriff anderer Transaktionen auf die gemeinsamen Daten verzögern.



Für den Fall, wo die gelesenen Daten durch einen menschlichen Benutzer bearbeitet werden, und die benötigte Zeit lang ist, müssen zwei unterschiedliche Transaktionen verwendet werden für das Lesen der Daten und das letztendliche Zurückschreiben. Eine lange Zeit sei hier definiert als eine Zeitspanne innerhalb der es nicht akzeptiert ist, dass es zu Resourcen-Blockierungen (Locks) kommen kann, welche den Zugriff anderer Transaktionen auf die gemeinsamen Daten verzögern.

Das wesentliche Problem, welches man sich hier einhandelt, ist die Unsicherheit, ob das Zurückschreiben noch auf den ursprünglich gelesenen Daten erfolgte, oder ob zwischenzeitlich eine andere Transaktion die Daten verändert hat. Änderungen einer anderen Transaktion, welche in der Zwischenzeit stattgefunden haben, werden einfach überschrieben und damit zunichte gemacht. Das kann erlaubt sein, nach der Idee: Die neuesten Änderungen sind sowieso die wahrscheinlich korrektesten. Es kann aber auch unerwünscht sein, wenn es sich beispielsweise um das Ändern eines Lagerbestandes in einer Materialbewirtschaftung handelt: Zwei Transaktionen lesen einen Lagerbestand von 10 Stück, ziehen jede ein Stück ab, und schreiben anschliessend je einen aktuellen Bestand von 9 zurück. Eine der Änderungen geht verloren. Als weitere Möglichkeit kann einfach gefordert werden, dass eine Applikation einfach benachrichtigt wird, wenn beim Zurückschreiben entdeckt wird, dass ein Konflikt entstehen könnte.

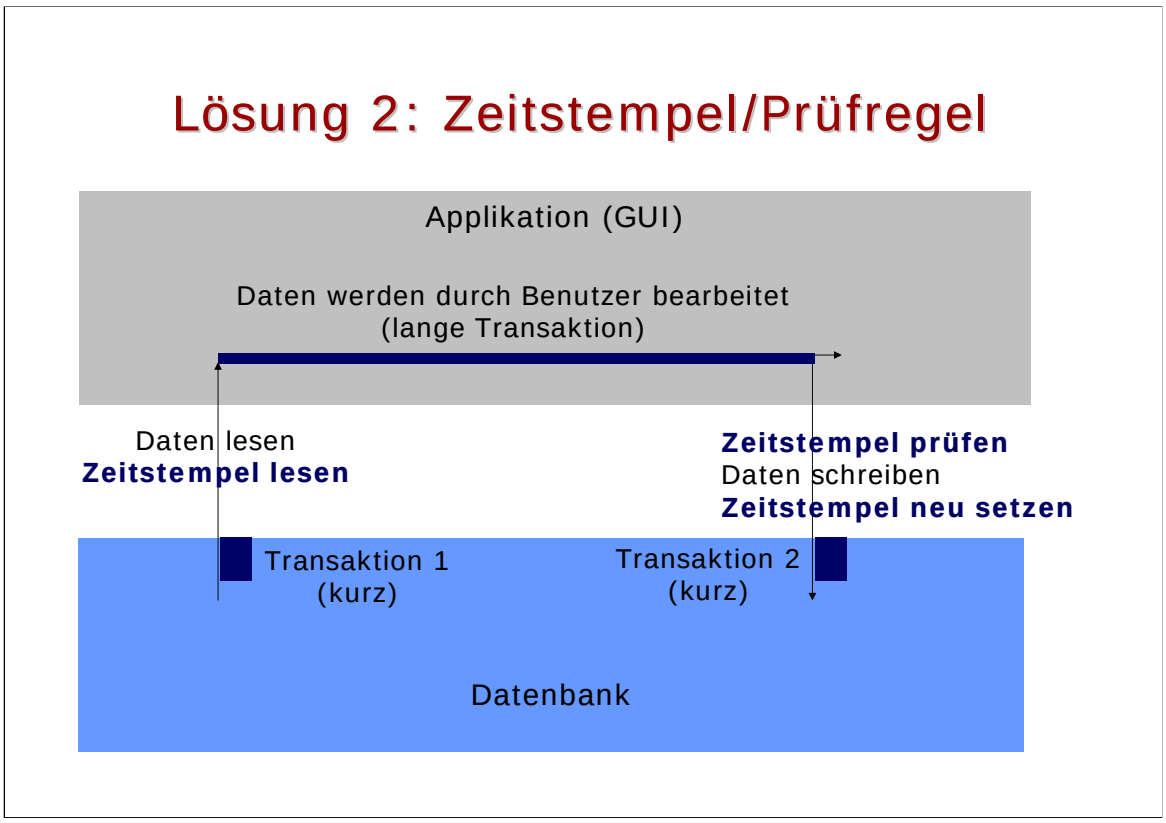
Lösung 1: checkout / checkin



Der checkout/checkin Mechanismus basiert auf freiwilliger Koordination mehrerer Applikationen: Wenn das Flag durch eine Applikation gesetzt ist, verzichten andere Applikationen auf den Gebrauch der markierten Daten. Das Flag kann unterschiedlich ausgestaltet sein:

- Binäres Flag, im Sinne eines Locks: Das Flag kann die Bedeutung einer exklusiven Sperre haben. Das Flag kann aber auch die Bedeutung haben, dass die flag-setzende Applikation die Daten ändern und zurückschreiben darf, während die anderen Applikationen nur lesend auf die Daten zugreifen dürfen.
- Flag + Zeitstempel + Benutzerinformation: Das Flag kann die Bedeutung einer exklusiven Sperre haben. Das Flag kann aber auch die Bedeutung haben, dass die flag-setzende Applikation die Daten ändern und zurückschreiben darf, während die anderen Applikationen nur lesend auf die Daten zugreifen dürfen. Die zusätzliche Information kann für folgenden Fall benutzt werden: Die Applikation, welche das Flag besitzt, vergisst vielleicht, das Flag wieder zurückzusetzen. Eine andere Applikation, respektive ein anderer Benutzer kann anhand von Zeitstempel und Benutzerinformation mit einer gewissen Sicherheit feststellen, ob die Daten tatsächlich noch in Benutzung sind, oder lediglich das Flag nie zurückgesetzt wurde.

  Zu beachten: Transaktion 2 muss im Modus SERIALIZABLE ablaufen!



Beim Lesen eines Datensatzes wird immer der Zeitstempel mitgenommen, welcher den Zeitpunkt der letzten Änderung angibt. Beim Zurückschreiben wird geprüft, ob der Zeitstempel noch stimmt. Wenn ja, werden die eigenen Änderungen in die Datenbank zurückgeschrieben. Wenn nein, werden die eigenen Änderungen verworfen.

Gegenüber dem checkout/checkin-Verfahren wird beim Zeitstempelverfahren zum Zeitpunkt des Lesens von Daten noch kein Flag oder eine andere Markierung gesetzt. Der Vorteil besteht darin, dass keine Flag-"Leiche" in der Datenbank entstehen kann, weil eine Applikation ein gesetzt Flag nicht korrekt zurücksetzt. Der Nachteil besteht darin, dass eine Applikation unter Umständen im späten Zeitpunkt des Zurückschreibens gezwungen ist, auf das Zurückschreiben zu verzichten, weil ein anderer Benutzer die Daten zwischenzeitlich verändert hat.

Das Zeitstempelverfahren kann abgewandelt werden:

- Im Falle eines Konfliktes wird die Applikation resp. der Benutzer informiert. Dieser kann sich dann entscheiden, ob er die Änderungen trotzdem vornehmen will oder nicht. Zu beachten ist für dieses Vorgehen aber, dass die Daten sich *nochmals* ändern können, während der Benutzer über den Abbruch entscheidet.
- Anstelle eines Zeitstempels werden die ursprünglich von der Applikation gelesenen Daten aufbewahrt (Nebst den vorgenommenen Modifikationen). Zum Zeitpunkt des Zurückschreibens wird geprüft ob die ursprünglich gelesenen Daten noch mit den aktuellen in der Datenbank übereinstimmen. Der Vorteil liegt darin, dass die Datenbank in keiner Weise mit Hilfsinformationen wie Zeitstempeln "verschmutzt" wird. Der Nachteil liegt im erhöhten Speicher- und Zeitbedarf für das Aufbewahren und Vergleichen der ursprünglichen Daten.
- Anstelle eines Zeitstempels werden ganz spezifische Konfliktprüfungsregeln angewendet. Beispielsweise könnte es erlaubt sein, dass zwei Benutzer gleichzeitig eine Reservation einfügen (Änderung einer Reservationstabelle durch Einfügen eines neuen Datensatzes), sofern sich die beiden eingefügten Datensätze bezüglich von- und bis-Datum der Reservation nicht überschneiden.

Transaktion vs. Verbindung

- Eine kurze Transaktion wird immer über *eine* Verbindung abgewickelt.
- Über eine Verbindung können *mehrere* Transaktionen abgewickelt werden, aber immer nur *nacheinander*.
- Kurze Transaktionen können die zugrundeliegende Verbindung nicht überdauern.
- Ein *Verbindungsabbruch* hat immer einen *Transaktionsabbruch* zur Folge.
- Lange Transaktionen beinhalten unter Umständen einen gewollten oder ungewollten Unterbruch der Verbindung: Sie müssen in der Regel selber implementiert werden.

Der wesentliche Vorteil einer Transaktion, die im Rahmen *einer* Verbindung abläuft ist, dass das Datenbanksystem selber entscheiden kann, wann die Transaktion zwangsläufig abgebrochen werden muss: Beim Verbindungsabbruch und damit auch bei einem allfälligen Servercrash. Kann oder muss eine Transaktion einen Verbindungsunterbruch überdauern, muss das Datenbanksystem in der Lage sein, alle Hilfsdaten der Transaktion (zum Beispiel Transaktionsname und gesetzte Locks) dauerhaft aufzubewahren. Dies ist bei allen gängigen Datenbank-Produkten aber nicht der Fall.

Unterstützung für lange Transaktionen

- Das Zeitstempelverfahren erlaubt das Abwickeln langer Transaktionen (READ COMMITTED-Garantie und Vermeidung von Lost Updates) innerhalb *einer* technischen Transaktion/Verbindung.
- Das Versionenverfahren von Oracle erlaubt das Abwickeln langer Lesetransaktionen (Daten mit REPEATABLE READ Garantie) innerhalb *einer* technischen Transaktion/Verbindung.

Zusammenfassung

- *Generische* Basismechanismen für das Concurreny Control werden von der Datenbank zur Verfügung gestellt.
- Für kurze Transaktionen und niedrig belastete Datenbanken kann generell mit dem höchstem Isolationsgrad gearbeitet werden.
- Bei hoher Datenbankbelastung muss eine schwächere Einstellung des Isolationsgrades geprüft werden.
- Lange Transaktionen erfordern immer eine *problembezogene* Vorgehensweise um Performance- und Korrektheit der Daten unter einen Hut zu bringen.

Recovery

- Aufgaben
- Fehlerarten
- Logging
- Fehlerbehebung

Das Recovery-System

- Das Recovery-System eines DBMS enthält alle Hilfsmittel zum Wiederherstellen eines korrekten Datenbank-zustandes nach
  - Transaktionsfehlern
  - Systemfehlern
  - Plattenfehlern
- Das Recovery-System garantiert die Atomarität und Dauerhaftigkeit einer Transaktion (ACID Regel).
- Das Recovery-Systems basiert auf dem Führen eines Logfiles, in welchem Änderungen protokolliert werden.
- Abschätzen und Überwachen der Grösse und Festlegen des Speicherortes für das Logfile sind zwei wichtige Aufgaben der Datenbank-Administration

Das Recovery-System bestimmt wesentlich die Performance eines Datenbanksystems. Eine Faustregel geht von 10-100 Transaktionen pro Sekunde aus. Die Zahl ergibt sich dadurch, dass spätestens bei Transaktionsende für jede Änderungstransaktion einige I/O-Pages ungepuffert auf das Logfile geschrieben werden müssen.

Bei hoher Transaktionslast kann die Anzahl Transaktionen pro Sekunde z.B. wesentlich erhöht werden, wenn die Änderungen mehrerer Transaktionen *zusammen* auf das Logfile geschrieben werden können ("grouped commit").

Fehlerarten

- Transaktionsfehler
  - Rollback-Befehl durch Applikation
  - Verletzung von Integritätsbedingungen
  - Verletzung von Zugriffsrechten
  - Deadlock
  - Verbindungsunterbruch oder Client-Crash
- Systemfehler
  - Stromausfall, Hardware- oder Memory-Fehler
- Plattenfehler
  - Speichermedium wird physisch defekt, Fehlfunktionen des Controllers

Transaktionsfehler sind relativ häufig und müssen effizient gehandhabt werden. Die Aufgabe der Datenbank ist es, den Zustand der Daten wie vor der Transaktion wieder herzustellen.

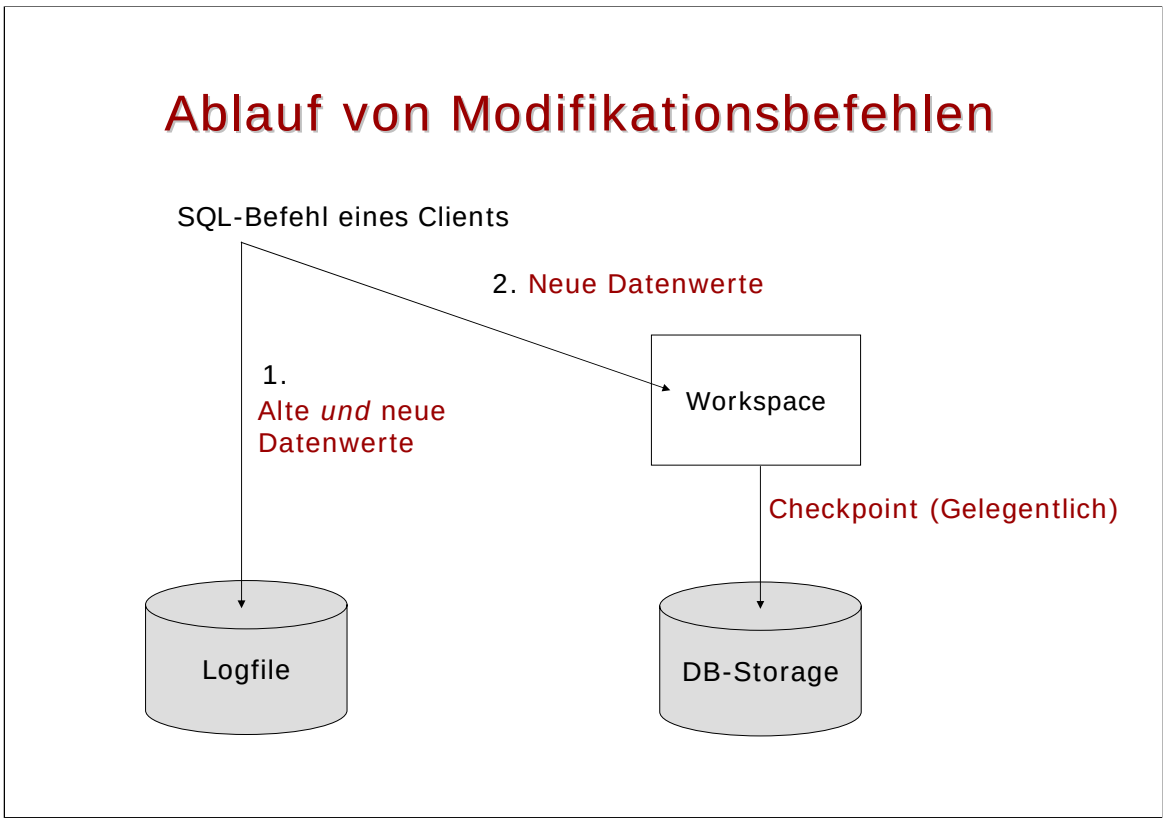
Bei Systemfehlern (auf Serverseite) gehen laufende Transaktionen und alle Memory-Inhalte des Servers verloren. Die Aufgabe des Datenbanksystems beim Restart ist es, den jüngsten korrekten Zustand der Datenbank wieder herzustellen. Dazu müssen die Änderungen aller zum Fehlerzeitpunkt laufenden Transaktionen rückgängig gemacht werden. Die Änderungen aller zum Fehlerzeitpunkt korrekt abgeschlossenen Transaktionen, welche aber nur im Memory abgelegt waren, müssen anhand des Logfiles rekonstruiert werden.

Bei Plattenfehlern hilft nur eine Wiederherstellung der Datenbank aus speziellen Backup-Kopien (On-Line-Backups). Die Aufgabe der Datenbank besteht darin, regelmässig solche Backups herzustellen.

Beachte, dass eine wegen einem Deadlock abgebrochene Transaktion nicht durch das DBMS wiederholt werden kann! Beispiel:

```
begin transaction
select lohn from Mitarbeiter where mitarbId = 9
// Applikation berechnet: neuer_lohn = f(lohn)
update Mitarbeiter set lohn = neuer_lohn where mitarbId = 9
// hier beispielsweise Deadlock ...
```

Durch den Abbruch der Transaktion werden die Sperren auf dem Mitarbeiter-Datensatz freigegeben. Das DBMS kann also nicht garantieren, dass das Attribut `lohn` noch denselben Wert hat wie beim ersten `select`. Da es die Funktion f nicht kennt, kann es nicht entscheiden, ob der update nochmals durchgeführt werden darf oder nicht.



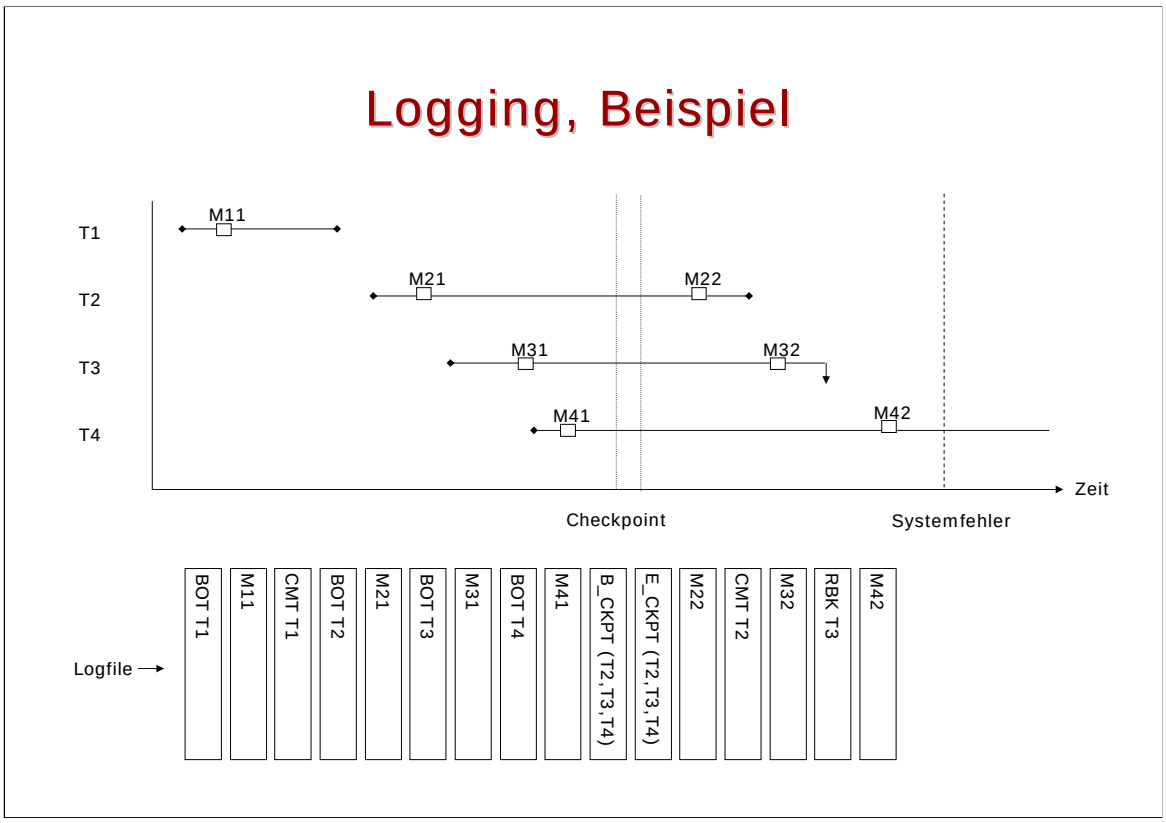
Im Logfile werden die Datensätze mit ihrem neuen und alten Zustand abgelegt. Damit ist grundsätzlich eine Wiederherstellbarkeit der Datenbank nach vorwärts wie nach rückwärts gewährleistet.

Der Workspace ist ein von der Datenbank vollständig kontrollierter Cache der permanenten Datenbank (DB-Storage). Hier befinden sich alle in Gebrauch stehenden Datensätze. Der Workspace wird zu bestimmten Zeitpunkten auf die permanente Datenbank zurückgeschrieben.

Die Bewirtschaftung des Logfiles kann nach zwei Strategien erfolgen:

1. Jeder SQL-Befehl wird sofort und ungepuffert protokolliert. Damit besteht keine Abhängigkeit vom Status des Workspace und des Ausführens eines Checkpoints. Gleichzeitig ist diese Variante aus Performance-Sicht etwas langsam.
2. Das Logfile hat einen eigenen Cache. Dieser muss jedoch spätestens beim Commit-Befehl oder vor einem Checkpoint in das permanente Logfile übertragen werden. Vorteil: mehrere commit-willige Transaktionen können zusammengefasst werden und deren Änderungen in einem Durchlauf vom Cache des Logfiles in das permanente Logfile geschrieben werden. Nachteil: Schwieriger zu implementieren für den DB-Hersteller. Eine Applikation muss eventuell auf den Commit-Befehl etwas länger warten.

Die Variante 2 ist aus globaler Sicht effizienter, aus lokaler Applikationssicht kann sie schlechter sein.



Das hier vorgestellte Logging dient der Behebung von Transaktions- und Systemfehlern. Es gibt verschiedenste Varianten in der Logging-Technik. Die hier präsentierte entspricht einem "guten Durchschnitt" der bekannten DB-Systeme, ohne allzu stark auf Details einzugehen. Sehr genaue Informationen sind in [3] zu finden.

Das Logfile

- Jeder Eintrag im Logfile ist mit einer LSN (Log Sequence Number) eindeutig identifiziert. Die LSN entspricht der Adresse des Eintrages. Alle Einträge einer bestimmten Transaktion sind untereinander rückwärtsverkettet, damit eine effiziente Behandlung von Transaktionsfehlern möglich ist.
  Alle Einträge im Logfile müssen permanent sein. Das Schreiben in das Logfile darf nicht gepuffert sein.
- Jede Änderung am Transaktionsstatus eines Client-Prozesses wird im Logfile festgehalten. Es wird also der Beginn einer Transaktion (BOT), der Commit (CMT) oder Rollback (RBK) festgehalten. Ein Rollback-Eintrag wird durch das Datenbanksystem auch erzeugt, wenn der Transaktionsabbruch erzwungen ist, z.B. durch einen Deadlock.
- Bei jedem SQL-Befehl, der Änderungen am Datenbestand zur Folge hat, werden die alten und neuen Datenwerte, inkl. Transaktionsnummer und Verweis auf die zugehörige Datenadresse in der Hauptdatenbank, protokolliert. In der Regel werden ganze I/O-Page in das Logfile geschrieben (physisches Logging). Die Platzverschwendung steht dabei einer effizienten Wiederherstellung der Datenbank gegenüber.
  Eine I/O-Page mit den alten Werten ("Before-Image") wird für die Wiederherstellung des Datenbankzustandes bei nicht korrekt abgeschlossenen Transaktionen verwendet. Eine I/O-Page mit neuen Werten ("After-Image") wird zur Wiederherstellung nach Systemfehlern benötigt, wenn eine Transaktion zwar korrekt beendet wurde, deren neue Datenwerte aber noch nicht vom Datenbank-Buffer in die physische Datenbank (Festplatte) geschrieben werden konnte.

- Jeder Checkpoint (siehe unten) mit Verweisen auf die laufenden Transaktionen wird ebenfalls im Logfile festgehalten.

Checkpoint

Als Checkpoint bezeichnet man den Zeitpunkt, an dem modifizierte I/O-Pages im Workspace der Datenbank auf das Speichermedium (Festplatte) geschrieben werden. Aus der Sicht von Benutzerprozessen möchte man möglichst wenige Checkpoints durchführen, da diese die Performance negativ beeinflussen. Aus Sicht einer raschen Wiederherstellung der Datenbank nach einem Crash möchte man möglichst viele Checkpoints durchführen. Zwingend wird ein Checkpoint, wenn der Workspace für alle modifizierten Seite zu klein wird. Nach dem Checkpoint können alte, nicht mehr benötigte I/O-Pages, ob geändert oder nicht, entfernt werden.

Grösse des Logfiles

jede Änderung eines Datensatzes erfordert die Speicherung des *alten* und des *neuen* ganzen Datensatzes im Logfile. Eingefügte resp. gelöschte Datensätze erfordern das Abspeichern des Datensatzes im Log. Die Grösse des Logfiles wächst also proportional zur Anzahl Änderungen. Da das Logfile häufig für inkrementelle Backups verwendet wird, muss genügend Platz für alle Änderungen zwischen zwei Backups vorhanden sein:

Grösse = a * (((m * g * 2) div p) + p)

a = Anzahl Transaktionen, die im Logfile aufbewahrt werden müssen

m = Anzahl geänderte Datensätze pro Transaktion

g = Grösse eines geänderten Datensatzes

div = Ganzzahl-Division, (3 div 2 ist also beispielsweise 0)

p = Seitengrösse der Datenbank (typischerweise ca. 2048 Bytes)

Beispiel: 1000 * ((10 * 2 * 1000) div 2048 + 2048) = 200 MB

Eine exakte Berechnung ist schwierig, da weitere Hilfsinformationen im Logfile abgelegt werden, was die Grösse erhöht. Eventuell können Transaktionen zusammengefasst werden, wenn sie zeitlich sehr nahe beeinanderliegen, was die Grösse verkleinert. Gewisse Datenbanken bewahren nur die neuen Datenwerte auf, weil die alten ausschliesslich im Cache gehalten werden. Dann entfällt der Faktor 2.

Behebung von Transaktionsfehlern

- Bei einem Transaktionsfehler werden aus den rückwärts verketteten Transaktionseinträgen im Logfile die alten Daten (Before Images) in den Cache übertragen.
- Das Datenbanksystem führt hierzu für jede laufende Transaktion einen Verweis auf den letzten Log- Eintrag mit. Der Transaktionsabbruch wird im Logfile ebenfalls protokolliert.
- Beispiel: Für Transaktion T3 müssen die Before-Images von M31 und M32 zurückgeladen werden.

Behebung von Systemfehlern

- Gewinner- und Verlierer-Transaktionen ermitteln
- Verlierer-Transaktionen mit Hilfe der Before-Images zurücksetzen
- Gewinner-Transaktionen mit Hilfe der After-Images noch einmal durchführen
- Checkpoint durchführen
- Beispiel
  - Gewinner: T2 -> M22 nachspielen.
  - Verlierer: T3 und T4 -> M31, M41 zurücksetzen.

Beim Restart des Datenbanksystems wird folgendes Recovery-Prozedere angewendet:

- Ausgehend vom letzten Checkpoint werden Gewinner- und Verlierer-Transaktionen ermittelt. Gewinner sind alle, für die ein Commit-Eintrag existiert. Verlierer sind alle, für die ein Rollback oder gar kein Eintrag vorliegt.
- In einem *Redo*-Lauf werden alle I/O-Pages in den Workspace zurückgeladen. Es müssen nur I/O-Pages berücksichtigt werden, die jünger als der letzte Checkpoint sind.
- In einem *Undo*-Lauf werden von den Verlierer-Transaktionen alle I/O-Pages mit den alten Datenwerten ("Before-Images") in den Workspace zurückgeladen. Das Logfile wird hierzu rückwärts abgearbeitet. Die Logfile-Einträge müssen zurückreichen bis zum Beginn der ältesten beim letzten Checkpoint noch laufenden Transaktion.

  Beachte, dass bei Verlierer-Transaktionen auch alle 'Before-Images' vor dem Checkpoint benötigt werden. Wurde nämlich eine Verlierer-Transaktion bei einem Rollback (nach dem Checkpoint) bereits einmal zurückgesetzt, ist diese Rücksetzung durch den Systemfehler ev. zunichte, wenn noch kein weiterer Checkpoint stattgefunden hat.
- Anschliessend an das Recovery wird ein Checkpoint ausgelöst und die Datenbank für den Multiuser-Betrieb freigegeben.

Neuere Datenbanktechnologien (Objektdatenbanken) verfolgen einen *Pure-Redo* Strategie. Diese geht davon aus, dass sämtliche Änderungen einer Transaktion bis zum commit-Befehl, rollback-Befehl oder Transaktionsabbruch im Speicher des Benutzerprozesses, rsp. in einem für ihn reservierten, privaten Workspace der Datebank gehalten werden können. Bei einem Transaktionsabbruch wird einfach dieser Workspace freigegeben. Das DBMS muss lediglich noch den Abbruch der Transaktion notieren und allfällige Sperren freigeben. Bei korrektem Abschluss der Transaktion mit einem commit-Befehl werden alle modifizierten Daten (After-Images) zum DBMS übertragen, dort in das Logfile geschrieben und anschliessend in die Datenbank übertragen. Bei einem Crash des DBMS muss bei der Wiederherstellung lediglich der Inhalt des Logfiles seit dem letzten Checkpoint nochmals auf die Datenbank übertragen werden (Redo-Lauf).

Plattenfehler

1. Massnahme: Ausfallrate von Platten verkleinern durch
   - Plattenspiegelung (via Datenbanksystem)
   - RAID System (via Betriebssystem)

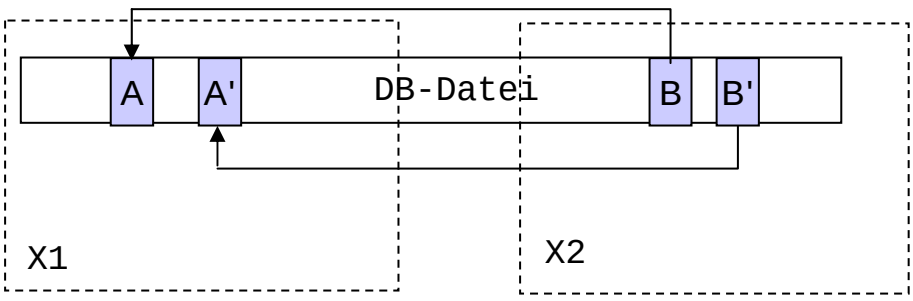
2. Massnahme: On-Line Backup erstellen. Nur On-Line Backups garantieren die Wiederherstellbarkeit einer Datenbank nach Plattenfehlern.

   Betriebssystem-Backups sind ungeeignet!

Plattenspiegelung vom Datenbanksystem selbst (Sybase) zur Verfügung gestellt. Plattenspiegelung ist sehr effizient: Die Zeit für den Schreibzugriff wird nicht erhöht, da parallel geschrieben werden kann. Lesezugriffe werden bei guter DB-Software sogar schneller, da beide Platten für unterschiedliche Lesezugriffe benutzt werden können.

Beim RAID-5 Verfahren werden die Daten parallel auf zwei Disks geschrieben und gleichzeitig wird Zusatzinformation für die Korrektur kleinerer Fehler auf den Platten gespeichert. RAID-5 besitzt eine sehr hohe Performance. Nachteil: Der Ausfall einer Platte verlangsamt den Zugriff bis zur vollständigen Rekonstruktion. Defekte Controller können das beste RAID- System unbrauchbar machen.

Betriebssystem-Backups sind ungeeignet für die Sicherung von Datenbanken, da sie inkonsistente Datenbanken speichern. Mit Dateien aus einem Betriebssystem-Backup kann eine abgestürzte Datenbank in der Regel nicht wieder hochgefahren werden. Beispiel eines inkonsistenten Ablaufes des Betriebssystem-Backups (BB) und des Datenbankservers (DB):
1. BB mach Backup von X1, 2: DB ändert A nach A' und B nach B', 3. BB macht Backup von X2. Beim Zurückladen des Backups würde B' auf nicht mehr existentes A zeigen.



On-Line Backup

- Anforderungen
  - Es wird ein konsistenter Datenbankzustand gesichert oder es kann ein solcher aus dem Backup rekonstruiert werden.
  - Der Datenbankbetrieb muss während dem Backup weiterlaufen.
  - *Im Extremfall gilt: Jede bestätigte* Transaktion muss bei einem Plattfehler rekonstruiert werden können.
- Schritte
  1. Voller Backup
  2. Inkrementeller Backup

Full Backup

1. Beginn des Backups im Logfile markieren.
2. Checkpoint durchführen, danach bis Backup-Ende keinen weiteren Checkpoint durchführen.
3. Alle zur Datenbank gehörenden Seiten (Dateien) sichern.
4. Als letztes die Before-Images aller beim Checkpoint laufenden Transaktionen in den Backup übertragen.
5. Ende des Backups im Logfile markieren.

Effekt

Der Backup gilt für den Zeitpunkt des Checkpoints, abzüglich der zu diesem Zeitpunkt noch laufenden Transaktionen, und abzüglich der während dem Backup gestarteten Transaktionen.

Logfile kürzen

Alle vor dem Checkpoint beendeten Transaktionen (commit oder rollback) können nach dem Backup aus dem Logfile gelöscht werden.

Inkrementeller Backup

1. Im Logfile den Beginn des inkrementellen Backups markieren.
2. Checkpoint durchführen.
3. Alle After-Images der beim letzten Backup-Checkpoint laufenden und aller danach gestarteten Transaktionen und bis zum jetzigen Checkpoint abgeschlossenen Transaktionen sichern.
4. Ende des Backups im Logfile markieren.
5. Das Logfile kann anschliessend um die gesicherten Transaktionen gekürzt werden.

...

Ablaufbeispiel On-Line Backup

T1	T2	T3	TB	Logfile	Kommentar
insert x				insert x T1	
			backup database		
				backup ready **BR**	TB wartet auf Commits aller bei BR laufenden Transaktionen.
commit				commit T1	
				checkpoint **CP**	TB führt nun Checkpoint durch. Danach kein Checkpoint mehr bis Backup-Ende.
					Backup läuft nun ...
		insert z		insert z T3	nur im Memory, nicht auf DB Dateien
		commit		commit T3	
	insert y1			insert y1 T2	nur im Memory, nicht auf DB Dateien
				backup end **BE**	
	insert y2			insert y2 T2	
	commit			commit T2	
					TB schreibt die Before-Images der beim Checkpoint laufenden Transaktionen auf den Backup
					Backup ist beendet

...

Restore

Full Backup zurückspielen.

Alle inkrementellen Backups nachspielen.

Sicherung gegen Verluste zwischen zwei Backups (volle oder inkrementelle)

Full- und Incrementell-Backup schützen nicht vor Verlusten *zwischen* zwei Backups. Hier hilft nur das Führen einer gespiegelten oder zusätzlichen Logdatei auf einem zweiten physischen Medium, damit die Logdatei sicher nicht verloren gehen kann. Alternativ gibt es auch Systeme mit Log-Servern. Jede fertige Transaktion wird im Rahmen des commit-Befehles an diesen Log-Server weitergeleitet, der eines oder mehrerer Logfiles auf unterschiedlichen Medien führt.

Varianten

Es gibt zahlreiche Modifikationen und Untervarianten dieses Verfahrens. Beispielsweise können während dem Full-Backup weitere Checkpoints möglich sein, um Memory-Probleme zu verhindern. Dann müssen aber die Before-Images aller bei Backup-Beginn laufenden und aller später eröffneten Transaktionen aufgezeichnet, und später dem Backup mitgegeben werden. Der Backup, überspielt mit diesen Before-Images, wiederspiegelt dann einen konsistenten Zustand zum Zeitpunkt des Backup-Beginnes.

Zugriffsoptimierung

- Zielsetzung
- Indexierung von Daten
- SQL Ausführungsplan
- Optimierungshinweise

Für eine gute Performance sind verschiedenste Software- und Hardware-Komponenten verantwortlich. Oft können Mängel im Software-Bereich durch eine schnellere CPU, grössere Disk, mehr Memory usw. ausgeglichen werden.

Bei Datenbanken ist die Komplexität einer schlecht optimierten Abfrage proportional zum Produkt der Tabellengrössen. Die folgende Join-Abfrage über zwei Tabellen mit je 10'000 Einträgen kann im schlimmsten Fall 10^8 Vergleichs- und Verknüpfungsoperationen zur Folge haben, im besten Fall können es weniger als 10^5. sein.

```
select name, vorname, strasse
from person, adresse
where person.persnr = adresse.persnr
```

Ein zentraler Aspekt bei der Performance-Optimierung ist die Art, wie das Datenbanksystem eine Abfrage durchführt. Dabei spielen Indices eine wichtige Rolle, aber auch gewisse Formulierungen im where-Teil eines SQL-Befehles. Diese beiden Themen werden deshalb im Folgenden besprochen.

Weitere wichtige Aspekte der Optimierung sind:

- Clustering von Tabellen: Welche Tabellen sind physisch auf denselben I/O-Pages zu speichern, weil sie häufig in Joins gebraucht werden?
- Füllfaktor von Tabellen: Führen grösser werdende Datensätze zu einer Reorganisation der I/O-Page?
- Parallelisierung von I/O: Können Logfile, Metadaten und eigentliche Datentabellen auf getrennten I/O-Geräten plaziert werden?
- Vermeidung von Wartezuständen im Multiuser-Betrieb!

Zielsetzung

- Ein zentrales Ziel der Abfrageoptimierung ist die Minimierung des Zugriffs auf I/O-Pages.
- Eine I/O-Page ist ein Datenblock fester Grösse, der am Stück von einem Speichermedium gelesen oder darauf geschrieben wird.
- Ein Query Optimizer erzeugt einen Query Execution Plan (QEP), der den Ablauf einer Abfrage festlegt.
- Die Hilfsinformation für die Planung einer Abfrage sind verschiedenste Statistiken.
- Das primäre Hilfsmittel für die Durchführung einer Abfrage ist ein Index.

Die Grösse einer IO-Page ist abhängig vom Produkt oder kann allenfalls bei der Initialisierung einer Datenbank angegeben werden. Die Grösse liegt häufig bei 2 oder 4 KB. Bei konfigurierbaren Produkten kann etwa im Bereich von 2 – 32 KB gewählt werden.

Indexierung von Daten

- Ein Index ist eine Hilfsstruktur zum schnelleren Auffinden von Datensätzen.
- Indices werden immer für ein, allenfalls mehrere Attribute *einer* Tabelle erzeugt.
- Für Primär- und Fremdschlüssel erzeugt das DMBS meist selbstständig einen Index.
- Für Attribute, die in Suchbedingungen oder Sortierklauseln vorkommen, werden zusätzliche Indices erstellt mit:
  `create index` *`ixname`* `on` *`table`* `(`*`attr1`* `[,`*`attrn`*`]...)`
- Indices haben meist die Struktur eines verzweigten, ausgeglichenen Baumes (B*).

Indices werden durch den Datenbankadministrator rsp. Tabellenbesitzer erzeugt. Sie dienen dem schnelleren Zugriff auf Datensätze bei bestimmten Formen der where-Klausel eines select-, update- oder delete-Befehles. Beispielsweise ist ein Index über dem Namen einer Person sehr effizient einsetzbar für das Auffinden von Datensätzen, wenn die Suchbedingung lautet: `where name = "Meier"`. Indices werden auch benützt, um die Eindeutigkeit von Primär- und Sekundärschlüsseln festzustellen. Wenn eine SQL-Tabellendefinition eine `primary key`-Definition enthält, wird bei manchen Datenbanksystemen automatisch ein Index auf den Schlüsselattributen erstellt.

Indices verschnellern unter vielen Umständen die Abfrage von Daten, verlangsamen aber auch Änderungsoperationen, weil bei jeder Änderung die Indexdaten nachgeführt werden müssen.

Indizes werden fast immer als *zusätzliche* Struktur neben den eigentlichen Tabellen erstellt und verwaltet. In grösseren Datenbanksystemen können aber auch die Basistabellen selbst nach *einer* Indexstruktur abgelegt sein.

Achtung: über die Verwendung eines Index entscheidet immer das Datenbanksystem! Der Benutzer setzt lediglich einen SQL-Befehl über *Tabellen* ab.

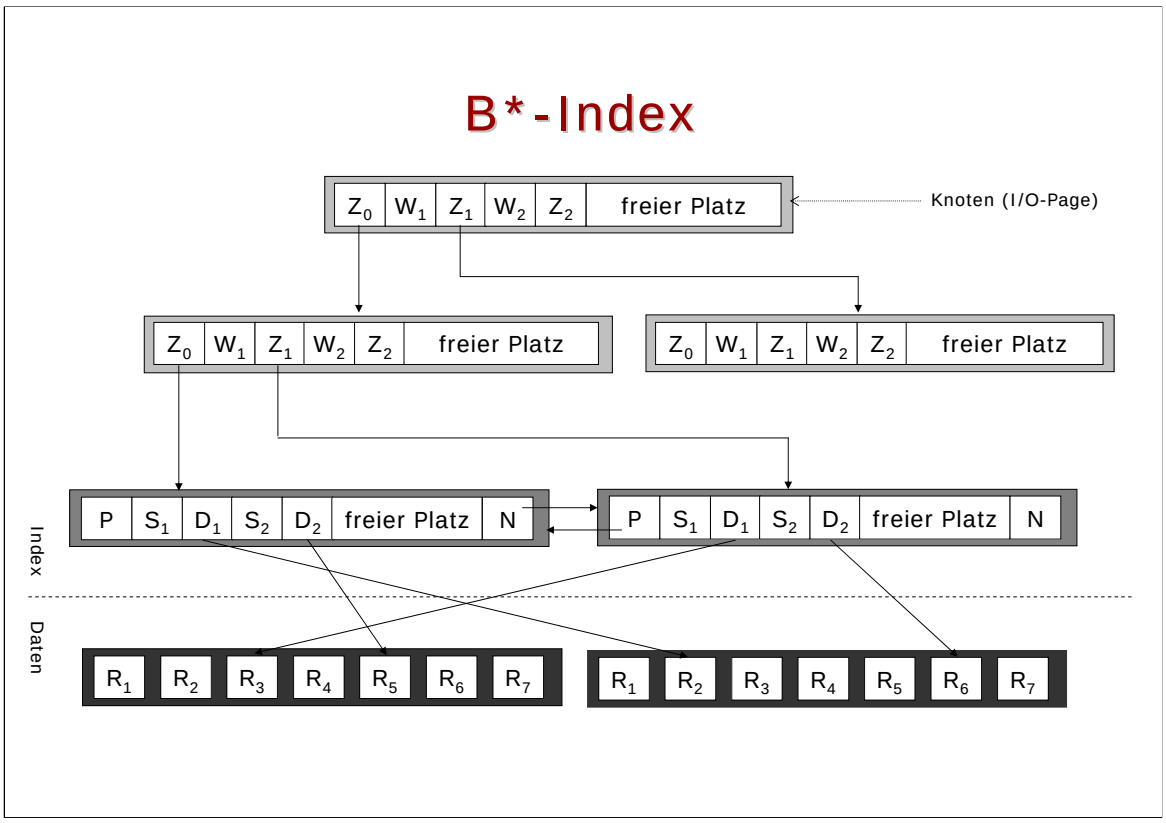
Indices können wesentlich mehr physikalischen Platz beanspruchen als die eigentlichen Nutzdaten. Ein Faktor 10 ist durchaus nicht ungewöhnlich.

Indices können nur auf "kurzen" Datentypen wie varchar, integer, float, date etc erstellt werden.

Index-Typen

B*-Bäume sind heute der häufigste Indextyp in Datenbanksystemen. Die Zugriffszeit auf bestimmte Datensätze wächst nur logarithmisch mit der Anzahl vorhandener Datensätze und nicht proportional wie das ohne Index der Fall wäre. B*-Bäume erlauben die indexunterstützte Suche nach Daten, wenn die Suchbedingungen einen oder mehrere Vergleichsoperatoren =, >, <, >=, <= und `like` enthält. Ausserdem können sie für die sortierte Ausgabe von Datensätzen eingesetzt werden.

Hash-Tabellen als Indices erlauben einen extrem schnellen Zugriff auf einzelne Datensätze. Die Zugriffszeit ist konstant und unabhängig von der Anzahl Datensätze! Jedoch können Hash-Tabellen nur für Suchbedingungen mit dem Operator = eingesetzt werden und Sie sind nicht für die sortierte Ausgabe von Daten geeignet.



Z_0, Z_1, Z_2	Zeiger auf Knoten
W_1, W_2	Wegweiser-Schlüssel (fiktive Schlüssel)
S_1, S_2	Effektive Schlüssel (entsprechen Daten)
D_1, D_2	Zeiger auf Datensätze
P, N	Zeiger auf Vorgänger- resp. Nachfolge-Knoten
R_1, R_2	Datensätze

Ein Knoten entspricht einer I/O-Page. Jeder Knoten enthält eine Anzahl Wegweiserschlüssel. Innerhalb des Knotens sind die Wegweiserschlüssel sortiert, und auf jeder Ebene von Knoten liegen die Wegweiserschlüssel von links nach rechts ebenfall sortiert vor. Vor jedem Wegweiserschlüssel W_i weist ein Zeiger auf einen Knoten, dessen Wegweiserschlüssel kleiner als W_i sind. Nach dem letzten Wegweiserschlüssel W_l in einem Knoten weist ein Zeiger auf einen Knoten, dessen Wegweiserschlüssel grösser oder gleich als W_l sind. In den Blättern des Baumes sind die tatsächlich in der Datenbank existierenden Schlüsselwerte abgelegt, zusammen mit einem Zeiger auf den eigentlichen Datensatz. Die Blattknoten sind untereinander als lineare Liste verbunden.

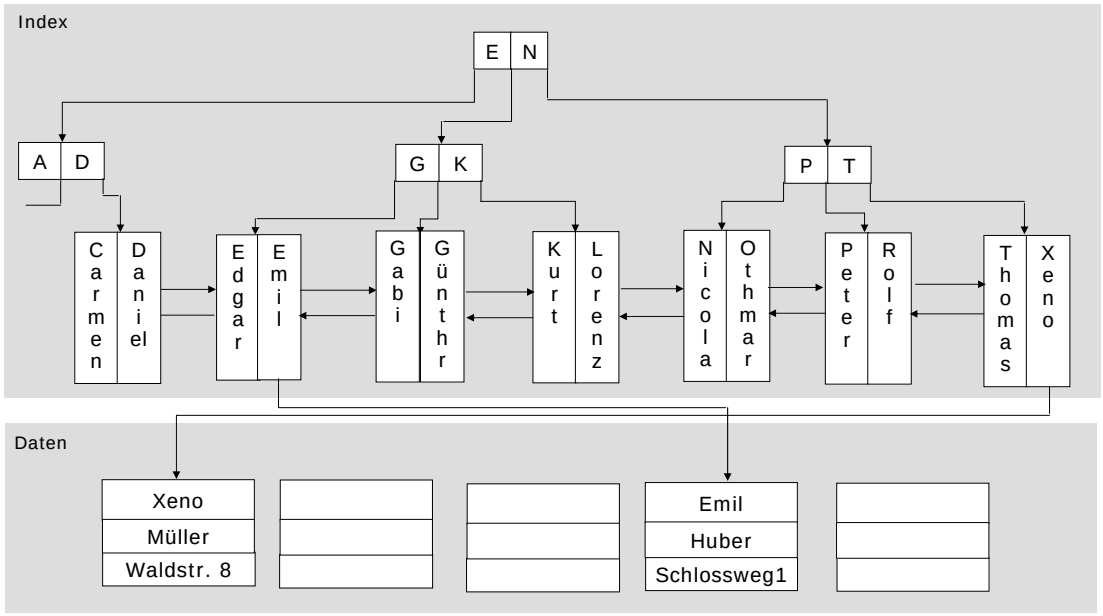
Die Wegweiserschlüssel können tatsächlichen Schlüsselwerten entsprechen oder Abkürzungen sein.

Der Zugriff auf einen Datensatz mit einem bestimmten Schlüsselwert benötigt rund $_{k+1}\log(N/k) + 1 + 1$ Zugriffe auf I/O-Pages. k bedeutet hier die Anzahl Schlüssel pro Knoten, N die Anzahl Schlüssel (rsp. Datensätze in der zugehörigen Basistabelle). Für das Auffinden jedes weiteren Datensatzes der dem Schlüsselwert genügt sind dann höchstens noch zwei Zugriffe notwendig, weil die Schlüsselwerte über alle Blattknoten sortiert vorliegen.

In praktischen Fällen, wo die Anzahl Schlüssel pro Knoten typischerweise grösser als 10 ist (häufig sogar grösser als 100), kann einfach gerechnet werden:

Anzahl Knotenzugriffe, um einen Datensatz zu finden = $_k\log(N/k) + 1 + 1$, bei 1'000'000 Datensätzen und 100 Schlüsseln pro Indexknoten also beispielsweise 4 Zugriffe.

B*-Index, Beispiel



Dieser Index enthält 2 Schlüsselwerte pro Knoten.

Von jedem Knoten gehen daher 3 Zeiger auf weitere Knoten aus. Der Verzweigungsgrad ist daher 3. Pro Ebene multipliziert sich die Anzahl Knoten um 3. Die unterste Ebene mit den Blattknoten muss breit genug sein, um alle vorkommenden Schlüsselwerte aufzunehmen. In diesem Beispiel gäbe es also für eine Tabelle mit 1000 Einträgen gesamthaft 6 Ebenen: 3^6 Knoten * 2 Einträge pro Knoten = 1458 Einträge.

Der Zugriff auf einen Datensatz via Index benötigt also: 3log(1000 / 2) + 2 = 8 Zugriffe auf Knoten.

Verzweigungsgrad = Anzahl ausgehende Zeiger von jedem Knoten = Anzahl Schlüssel pro Knoten + 1.

Anwendungsmöglichkeiten B*

- Auffinden einzelner Datensätzen eines bestimmten Schlüsselwertes. Beispiel:
  `where name = 'Meyer'`
- Auffinden von Datensätzen in einem bestimmten Schlüsselbereich.Beispiel:
  `where gebdatum between '1.1.2000' and 1.1.2001`
- Sortierte Ausgabe von Daten. Beispiel:
  `order by name`
- Auffinden von Datensätzen wenn nur der Anfang des Schlüssels bekannt ist. Beispiel:
  `where name like 'M%'`

Mit einem B*-Indices auf dem Attribut `att` können Datensätze gesucht werden, für die Bedingungen der folgenden Art gelten

`where att op wert` (`op` ist dabei einer der Operatoren `=`, `>`, `<`, `>=`, `<=`, `like`)

```
where att > wert1 and att < wert2
order by att
```

Weil die Zeiger auf die eigentlichen Daten und die effektiven Schlüsselwerte erst in den Blattknoten vorkommen, können die inneren Knoten wesentlich stärker gepackt werden, also mehr Zeiger und Wegweiser-Schlüssel enthalten. Damit ist die Tiefe des Baumes kleiner und das Auffinden einzelner Datensätze ist noch effizienter als bei der in Algorithmen-Büchern häufig beschriebene B-Bäumen. Letztere enthalten in *jedem* Knoten vollständige Schlüsselwerte und Zeiger auf Datensätze.

Join-Bildung

- Joins sind ein zentrales Konstrukt in SQL. Sie werden hauptsächlich in drei Varianten durchgeführt.
  - Kartesisches Produkt
    - Doppelte Schlaufe zur Abarbeitung der Tabellen
    - Aufwand $M/k_1 * N/k_2$
  - Lookup Join
    - Äussere Tabelle durchlaufen, für innere Tabelle Index verwenden.
    - Aufwand $M/k_1 * ({}_{k2}\log(N/k_2)+2)$
  - Sort-Merge
    - Beide Tabellen sortieren, dann abgleichen
    - Aufwand $M*{}_{k1}\log(M) + N*{}_{k2}\log(N) + (M/k_1+N/k_2)$

Kartesisches Produkt
: Beide Tabellen werden in einer doppelten Schlaufe abgearbeitet. Es werden keine bestehenden Indices benötigt. Das kartesische Produkte wird angewendet bei sehr kleinen Tabellen (wenige I/O-Pages), wo eine Sortierung oder temporäre Indexbildung nicht lohnenswert ist.

Lookup-Join
: Für jeden Datensatz in der äusseren Tabelle wird via einen Index der inneren Tabelle der zugehörige Datensatz ermittelt. Lookup-Joins sind effizient, wenn die äussere Tabelle wesentlich kleiner ist als die innere. Die äussere Tabelle kann dabei einer realen Tabelle entsprechen, oder das Zwischenresultat einer bereits ermittelten Abfragebedingung auf dieser sein.

Sort-Merge Join
: Beide zu verbindenden Tabellen werden zuerst nach dem Join-Attribut sortiert. Anschliessend werden die sortierten Tabellen gegeneinander abgeglichen (Merge). Die Sortierung der einen oder anderen Tabelle kann entfallen, wenn für das zu sortierende Attribut ein Index besteht. Sort-Merge Joins sind sehr effizient, wenn beide beteiligten Tabellen etwa gleich gross sind. Wenn für beide Tabellen ein Index auf dem Join-Attribut vorhanden ist, besteht der Aufwand lediglich im Mischen der Blätter der beiden Indexe. Das Verfahren ist dann extrem effizient.

Es gibt viele weitere Algorithmen und Varianten der obengenannten Verfahren zur Join-Bildung. Zu erwähnen ist hier noch der in letzter Zeit häufiger anzutreffende Hash-Join: Aus jedem Datensatz wird für den Wert der Join-Attribute ein Hashwert berechnet. Datensätze mit gleichem Hashwert aus den beiden Tabellen kommen in einen Bucket pro Hash-Adresse. Am Schluss wird der Inhalt jedes Buckets durch Verknüpfen der Datensätze der linken und rechten Tabelle verarbeitet und ausgegeben.

Ausführungsplan

- Eine Ausführungsplan (QEP) bestimmt, mit welchen Indexzugriffen, mit welchen Join-Methoden usw. eine Abfrage durchgeführt wird.
- Das DBMS kann für den ermittelten Ausführungsplan Informationen anzeigen über
  - Verwendete Basistabellen, Indices, Hilfstabellen
  - Angewendete Operationen
  - Benötige Anzahl I/O-Operationen und CPU-Zeit
- Anhand des Ausführungsplanes können kritische Teile einer Abfrage identifiziert und ev. umformuliert oder neue Indices definiert werden.

Die Ermittlung des Ausführungsplanes ist eine sehr komplexe Aufgabe, die viel Zeit in Anspruch nehmen kann. Für die Anwendungspraxis ist es meist ausreichend, die wichtigsten Faktoren zu kennen, welche eine Abfrage schnell oder langsam machen. Entsprechend sind dann die Abfragen zu formulieren oder Indices auf Tabellen zu setzen. Die Besichtigung des Ausführungsplanes mit den effektiven benötigten I/O-Operationen dient der Überprüfung der Performance.

Ein Datenbanksystem ist ohne weiteres in der Lage, eine Abfrage über eine grosse Anzahl Tabellen (>10) hinweg vernünftig zu optimieren.

In Sybase wird die Anzeige des Ausführungsplanes und der I/O-Statistik mit `set showplan on`, `set statistics time on` und `set statistics io on` aktiviert.

Optimierungshinweise

- Auf Primär- und Fremdschlüsselattributen einen Index erstellen.
- Für Attribute, die häufig in der order by Klausel auftreten, einen Index erstellen.
- Indices beschleunigen Abfragen, aber verlangsamen Änderungen.
- Der Boolsche Operator NOT und der Vergleichsoperator <> sind nicht optimierbar.
- Ausdrücke mit dem Vergleichsoperator LIKE sind nur optimierbar, wenn *allfällige Wildcards nicht am Anfang* des Suchmusters stehen.
- Ausdrücke mit einem Funktionsaufruf über einem Attribut sind nicht optimierbar.

Where-Klausel als optimierbaren Ausdruck gestalten:

Optimierbarer Ausdruck = Indexiertes Attribut VglOp Einfacher Ausdruck

Ein einfacher Ausdruck enthält Attribute, Konstanten und Funktionen davon.

Optimierbarer Ausdruck = Optimierbarer Ausdruck BoolOp Ausdruck

VglOp ✍ { <, >, <=, >=, =, LIKE, BETWEEN }

BoolOp ✍{ AND, OR }