Modul 153

Dieses Dokument stellt eine Zusammenfassung des Lernstoffs von Modul 153 dar, der für die richtugsspezifische Lehrabschlussprüfung 2007 relevant ist. Die Zusammenfassung gliedert sich in die folgenden Teile:

Im ersten Teil wird auf allgemeines Wissen rund um Datenbanken eingegangen. Dabei geht es um den grundlegenden Aufbau eines Datenbanksystems (logische und physische Architektur).

Der zweite Teil dreht sich um das Design von Datenbanken. Dabei wird auf theoretische Aspekte zu Datenbank-Anomalien, auf die Erstellung von ER-Diagrammen und auf die Überführung von ER-Diagrammen in ein relationales Modell eingegangen.

Konkurrierende Zugriffe sind das Thema des dritten Teils. Hier geht es um Transaktionen, Synchronisationsprobleme und Sperren.

SQL ist das Thema der Teile vier, fünf und sechs. Im vierten Teil geht es um allgemeine Informationen zu SQL. Es folgt der fünfte Teil über DDL – die Data Definition Language, genauer geht es darin um den `CREATE TABLE`-Befehl. Der `SELECT`-Befehl (DQL, Data Query Language) ist dann Thema des sechsten Kapitels.

Im siebten Teil, wird kurz auf Trigger eingegangen. Dabei geht es um theoretische Aspekte von Triggern (was ist ein Trigger, wie funktioniert er) und nicht um die Implementierung von Triggern.

Der achte und letzte Teil widmet sich den Grundlagen verteilter Datenbanksysteme.

Freigabe

Version	***Datum***	***Bearbeiter***	***Änderungen***
0.01	07.04.2007	Patrick Bucher	- Neues Dokument aufgesetzt - Inhalte aus altem Dokument eingepflegt
0.02	06.05.2007	Patrick Bucher	- Zusammenfassung gemäss Prüfungsstoff umstrukturiert - Kapitel über den Datenbankentwurf übernommen
0.03	27.05.2007	Flavio Hüsler Kilian Schwarzentruber Patrick Bucher	- Korrekturen gemäss Reviews durchgeführt - Kleinere Umstrukturierungen vorgenommen
0.04	27.05.2007	Patrick Bucher	- Konkurrierende Zugriffe zusammengefasst - Trigger zusammengefasst
0.05	28.05.2007	Flavio Hüsler Patrick Bucher	- Grundlagen von SQL zusammengefasst - Verteilte Datenbanken zusammengefasst
0.06	29.05.2007	Flavio Hüsler Patrick Bucher	- SQL-DDL (`CREATE TABLE`) zusammengefasst - SQL-DQL (`SELECT`) zusammengefasst
1.00	31.05.2007	Michael Egger Patrick Bucher	- Korrekturen durchgeführt - Dokument freigegeben
1.01	02.06.2007	Patrick Bucher	- Korrekturen an Kapitel 6 durchgeführt

Inhaltsverzeichnis

1 Allgemeines

1.1 Datenbanksystem

Die folgenden Abschnitte sollen eine Einführung in den allgemeinen Aufbau von relationalen Datenbanksystemen bieten. Dabei wird nicht speziell auf ein Produkt, sondern auf den allgemeinen Aufbau eingegangen.

Ein Datenbanksystem (kurz *DBS* = **d**ata**b**ase **s**ystem) besteht aus zwei Hauptkomponenten:

1. Dem Datenbankverwaltungssystem (*DBMS* = **d**ata**b**ase **m**anagement **s**ystem)
2. und der eigentlichen Datenbank (*DB* = **d**ata**b**ase)

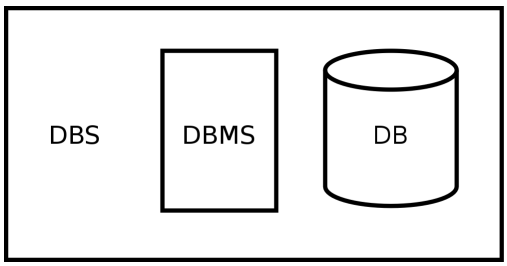


Abbildung 1: Das Datenbanksystem

1.1.1 Datenbank

Die Datenbank ist der Teil eines DBS, in welchem die Daten physisch abgespeichert werden (meist in Form von Tabellen). Neben den eigentlichen Nutzdaten werden auch noch Informationen wie Indizes, Logs und Metadaten in der DB abgespeichert.

1.1.2 Datenbankverwaltungssystem

Das Datenbankverwaltungssystem bietet diverse Unterstützungsdienste und regelt den Zugriff auf die DB. Zusammengefasst hat ein DBMS folgende Aufgaben:

- Zugriff auf die DB ermöglichen
- Verwaltung der Index-Dateien
- Transaktionsverwaltung
- Korrektheit bei Mehrbenutzerbetrieb gewährleisten
- Datensicherung
- Sicherheit

Für den Programmierer stellt ein Datenbankverwaltungssystem folgenden Nutzen dar:

- Die Daten werden abstrahiert, der Zugriff wird somit einfacher
- Die Verwendung eines DBMS spart Zeit und somit Geld gegenüber dem direkten Datenzugriff auf Dateiebene
- Man braucht sich nicht um die *Integrität* der Daten zu kümmern

1.2 Logische Architektur

Das DBMS kann vom Programmierer als *Blackbox* betrachtet werden, d.h. man braucht sich nicht darum zu kümmern, wie ein DBMS aufgebaut ist. Es ist aber dennoch nützlich, wenn man grundlegend über die Arbeitsweise des DBMS Bescheid weiss.

Ein DBMS ist vereinfacht ausgedrückt in drei Ebenen unterteilt:

1. Die externe Ebene (benutzerspezifische Sicht)
2. Die konzeptuelle Ebene (logische Sicht)
3. Die interne Ebene (physische Sicht)

Zum besseren Verständnis eine Darstellung dieser drei Ebenen:

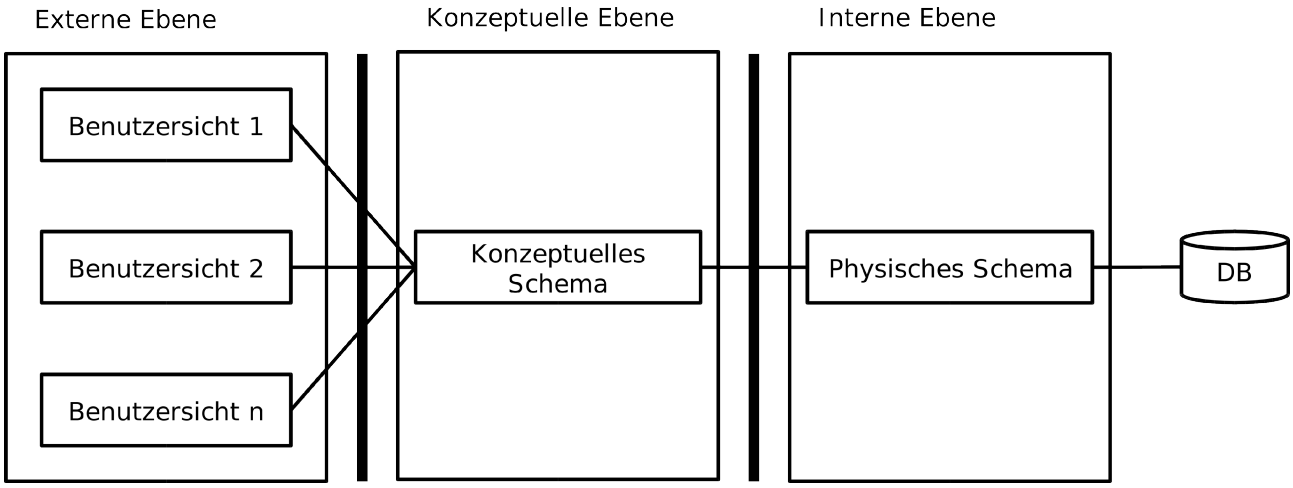


Abbildung 2: Die drei Ebenen eines Datenbanksystems

1.2.1 Externe Ebene

Die Bereitstellung der Daten erfolgt in der externen Ebene durch ein Anwendungsprogramm.

Der Benutzer interagiert nur mit der externen Ebene eines Datenbanksystems, die beiden „tieferen" Ebenen bleiben ihm verborgen. Der Benutzer hat somit eine abstrakte Sicht auf die inneren Ebenen.

1.2.2 Konzeptuelle Ebene

Die konzeptuelle Ebene befasst sich mit der *Abstraktion* der Daten. Die Darstellung der Daten, die im physischen Schema vorliegen, wird also für den Benutzer in eine abstraktere Sicht überführt. Liegen Daten in einer abstrakten Darstellung vor, so müssen diese in das physische Schema der internen Ebene überführt werden. In einem relationalen Datenbanksystem möchte der Benutzer Daten in Form von ganzen Tabellen sehen und nicht in der Form von Seiten, wie sie in der internen Ebene vorliegen.

Die konzeptuelle Ebene hat zusammengefasst die Aufgabe, die Daten für die interne/externe Ebene entsprechend darzustellen.

1.2.3 Interne Ebene

Die interne Ebene kümmert sich um die physische Abspeicherung der Daten. Weiter ist die Verwaltung des Pufferspeichers auf dieser Ebene angesiedelt.

Die kleinsten Einheiten sind dabei sog. *Seiten*, die von der internen Ebene zwischen dem physischen Datenspeicher (Harddisk) und dem Memory hin und her bewegt werden. Eine weitere wichtige Aufgabe der internen Ebene ist die Sicherung der Datenkonsistenz zwischen Datenspeicher und Memory (Synchronisation). Sollten sich Datenobjekte ändern, so verwaltet die interne Ebene, wie die Speicherseiten verändert werden müssen. Ausserdem hat die interne Ebene die Aufgabe, die Indizes zu verwalten.

1.2.4 Ablauf einer Abfrage

Stellt der Benutzer eine Anfrage an ein Datenbanksystem, so sieht dies zunächst nicht nach viel Arbeit aus; das Ergebnis erscheint meist schon nach wenigen Millisekunden. Im Datenbanksystem selber werden jedoch viele Operationen durchgeführt. Der Ablauf einer Abfrage kann (stark vereinfacht) wie folgt aussehen:

1. Das Datenbanksystem empfängt die vom Benutzer gestellte Anfrage.
2. Die Anfrage läuft durch eine Syntaxprüfung und wird validiert.
3. Das DBMS prüft, ob der Benutzer die notwendigen Rechte zum Ausführen der gestellten Anfrage besitzt.

4. Die physischen Daten und deren Zugriffspfade werden ermittelt (Umwandlung der externen Sicht in die interne Sicht).
5. Das DBMS beauftragt das Betriebssystem mit dem Lesen der ermittelten Speicherbereiche.
6. Die gelesenen Daten werden vom Betriebssystem in den Systempuffer des Datenbanksystems geschrieben.
7. Die Daten werden in die externe Sicht überführt und der Anfrage entsprechend zusammengestellt.
8. Das Datenbanksystem sperrt die Daten für andere Benutzer solange, bis deren Bearbeitung abgeschlossen ist.
9. Die transformierten Daten werden dem Anwendungsprogramm übergeben und gelangen somit zurück zum Benutzer.

1.3 Physische Architektur

Die Aufgabe eines Datenbanksystems ist es, verschiedenen Benutzern einen Zugriff auf einen Datenbestand über ein Netzwerk zu bieten. Es gibt verschiedene Ansätze, **wie** dieser Zugriff erfolgen kann. Man kann zwischen folgenden drei Ansätzen unterscheiden:

- Client/Server
- File/Server
- Hostbasiertes System

Dabei spielt es keine Rolle, ob sich der Client und der Server physisch auf dem gleichen Rechner befinden oder über ein Netzwerk miteinander verbunden sind.

1.3.1 Die Client/Server-Architektur

Die meisten modernen DBS bauen auf der Client/Server-Architektur auf. Dabei laufen auf dem Client und dem Server zwei unabhängige Prozesse. Die Kommunikation läuft über eine definierte Schnittstelle nach dem Frage-Antwort-Prinzip. Der Server bietet den Clients also einen Dienst an, die Clients nehmen diesen in Anspruch.

Der Vorteil an dieser Architektur ist derjenige, dass der Server den Datenbestand selbständig verwaltet und somit mehrere Clients über den Server auf diesen Datenbestand zugreifen können.

Client und Server können auch auf dem gleichen Rechner laufen. Auch hier spricht man von einer Client/Server-Architektur, da der Client- und der Serverprozess nach wie vor selbständig agieren. In der Regel wird der Serverprozess aber auf einen leistungsstarken Rechner gelegt, der sonst keinen Client-Aufgaben nachkommen muss und dementsprechend schneller läuft.

Beispiele für Client/Server-DBS:

- Oracle
- DB2
- Microsoft SQL Server
- PostgreSQL
- MySQL

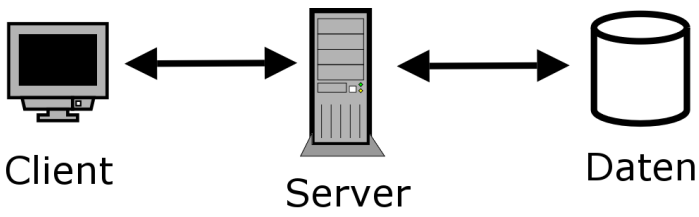


Abbildung 3: Die Client/Server-Architektur

1.3.2 Die File/Server-Architektur

Bei einer File/Server-Architektur agiert ein Rechner zugleich als Client und als Server. Es liegt somit keine physische Trennung zwischen Client und Server vor. Mit anderen Worten; ein File/Server-DBS greift direkt auf den Datenbestand zu. Dieser Datenbestand kann sich entweder lokal oder im Netzwerk auf einem anderen Rechner befinden.

Die Aufgabe der Verwaltung des Datenbestandes kommt also nicht wie bei der Client/Server-

Architektur dem Serverprozess zu, sondern wird direkt von einem Datenbanktreiber übernommen.

Die Datenbanktreiber können verwendete Datensätze mit Locks versehen. Ein Mehrbenutzerbetrieb ist also grundsätzlich möglich, jedoch nicht gerade performant. File/Server-DBS eignen sich eher für die Verwendung von wenigen Benutzern mit einer sehr geringen Anzahl an Transaktionen.

Beispiele für File/Server-DBS sind:

- Jet-Engine (Microsoft Access)
- FoxPro
- dBase
- Paradox

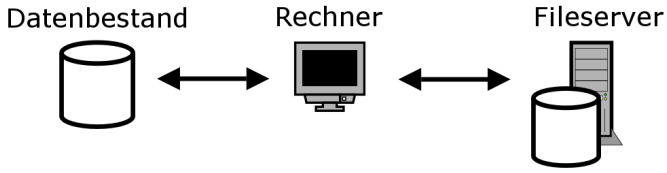


Abbildung 4: Die File/Server-Architektur

Das Konzept der File/Server-Architektur ist auf Abbildung 4 schematisch dargestellt. der Datenbestand kann sich entweder auf dem gleichen Rechner, oder aber auf einem speziellen Fileserver befinden.

1.3.3 Die hostbasierte Architektur

Ein hostbasiertes DBS besteht aus zwei Komponenten; dem *Host* und dem *Terminal*. Wie bei der Client/Server-Architektur greift hier der eine Rechner (Terminal) über den anderen Rechner (Host) auf den Datenbestand zu. Der Unterschied ist jedoch der, dass der Client- und der Serverprozess auf dem Host laufen, das Terminal dient nur zur Ein- und Ausgabe der Daten. Man spricht auch gelegentlich von „dummen" Terminals, da diese keinerlei Logik besitzen.

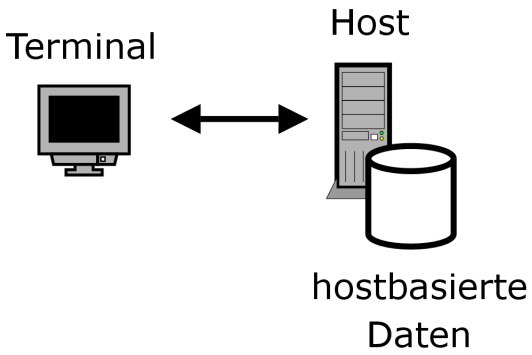


Abbildung 5: Hostbasierte Architektur

2 Datenbankentwurf

Ein sauberer Datenbankentwurf ist sehr wichtig für jedes datenbankgestützte Softwareprojekt. Fehler, die beim Datenbankentwurf geschehen, lassen sich im späteren Projektverlauf nur noch schwer und mit viel Aufwand beheben.

2.1 Anomalien

Ein wichtiger Aspekt am Datenbankdesign ist die Vermeidung von *Anomalien*. Eine Anomalie tritt bei einem unsauberen Datenmodell auf und führt zu inkonsistenten Daten. Dabei wird zwischen Einfüge-, Änderungs- und Lösch-Anomalien unterschieden.

2.1.1 Einfüge-Anomalie

Eine Einfügeanomalie entsteht dann, wenn nach dem Einfügen von Datensätzen einige Werte mehrfach in der Datenbank vorkommen oder Datenfelder leer bleiben. Beispiel:

PerId	***Vorname***	***Nachname***	***Projekt***	
1	Hans	Meier	**Einkauf**	
2	Martin	Schwegler	Verkauf	
3	*Max*	*Birrer*	**Einkauf**	***Neu***
4	Alice	Huber		***Neu***

Das Projekt „Einkauf" existiert nach dem Hinzufügen des dritten Eintrags doppelt – es ist also eine Redundanz entstanden.

Ein weiterer Aspekt der Einfüge-Anomalie ist bei Datensatz vier ersichtlich. Hier wurden nicht alle Felder ausgefüllt (Projekt), was zu inkonsistenten Zuständen führen kann.

2.1.2 Änderungs-Anomalie

Wird ein redundanter Datensatz in einer Relation geändert, so müssen alle dazugehörigen Datensätze ebenfalls entsprechend angepasst werden. Passiert dies nicht, so ist die Rede einer Änderungs-Anomalie. Beispiel:

SpendenNr	***Vorname***	***Nachname***	***Betrag***
1	Hans	Meier	50.-
2	Martin	Schwegler	30.-
3	Hans	Meier	80.-

Hans Meier hat zweimal gespendet, einmal 50 Franken und einmal 80 Franken. Da aber „Herr Meier" eigentlich „Herr Maier" heisst, wird die Relation nun aktualisiert.

SpendenNr	***Vorname***	***Nachname***	***Betrag***	
1	Hans	**Maier**	50.-	***Aktualisierung***
2	Martin	Schwegler	30.-	
3	Hans	**Meier**	80.-	

Da der dritte Datensatz nicht angepasst wurde, entsteht der Eindruck, dass es drei Spender gibt, die jeweils eine Spende abgegeben haben. Die Relation stimmt mit der realen Welt nicht mehr überein und ist somit inkonsistent.

2.1.3 Löschanomalie

Bei der Löschung eines Datensatzes werden Informationen mit gelöscht, welche eigentlich in der Relation hätten verbleiben sollen. Beispiel:

PerId	***Vorname***	***Nachname***	***Projekt***
1	Hans	Meier	Einkauf
2	Martin	Schwegler	**Verkauf**
3	Max	Birrer	Einkauf

Löschen

Durch die Löschung des zweiten Datensatzes wird das ganze Projekt „Verkauf" gelöscht. Dieses hätte jedoch für eine spätere Verwendung noch in der Datenbank verbleiben sollen.

2.1.4 Anomalien vermeiden

Es gibt verschiedene Möglichkeiten, um die genannten Anomalien zu vermeiden:

- Die Datenbank wird anhand eines *Entity Relationship* Modells erstellt und dann in ein relationales Modell überführt.
- Die Relationen werden *normalisiert,* d.h. in eine Normalform überführt.

Diese beiden Ansätze werden in den folgenden Abschnitten genauer erklärt.

2.2 Das ER-Diagramm

Das Abbilden von Informationsstrukturen aus der realen Welt in das relationale Schema ist ein aufwändiger Prozess. Bei grösseren Informationsstrukturen empfiehlt es sich, in mehreren Arbeitsschritten vorzugehen. Dabei hat sich das *Entity Relationship* (ER) Modell zum de facto Standard entwickelt.

Im ER-Modell wird zwischen folgenden Komponenten unterschieden:

- **Entitäten**
  Eine *Entität* ist die Abbildung eines Objekts aus der realen Welt, z.B. Person, Auto, Haus usw.
- **Attribute/Eigenschaften**
  Jede Entität besitzt bestimmte *Eigenschaften*. So hat eine Person z.B. einen Vor- und einen Nachnamen. Eigenschaften die ein Objekt eindeutig identifizieren werden als *Schlüsseleigenschaften* bezeichnet.
- **Beziehungen**
  Entitäten stehen in bestimmten *Beziehungen* zueinander. So kann z.B. eine Person in einem bestimmten Ort wohnhaft sein oder ein Lastwagen verschiedene Güter transportieren. Beziehungen können auch Eigenschaften besitzen.

2.2.1 Symbolik

Um Entitäten, Eigenschaften und Beziehungen in einem Diagramm darzustellen, wurde folgende Symbolik festgelegt:

Abbildung 6: Entität

Abbildung 7: Beziehung

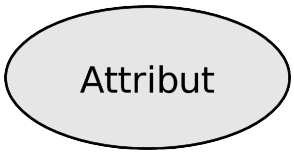


Abbildung 8: Attribut

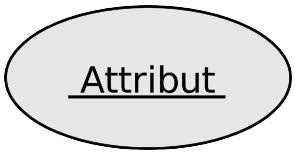


Abbildung 9: Schlüsselattribut

2.2.2 Kardinalitäten

Bei jeder Beziehung gilt eine bestimmte *Kardinalität*. Diese sagt aus, wie viele Instanzen einer Entität einer Instanz der jeweils anderen Entität zugeordnet werden können.

Beispiel: Die Entitäten Mann und Kind stehen in einer Beziehung zueinander. Ein Mann kann kein, ein oder mehrere Kinder haben. Ein Kind hat aber immer nur einen einzigen Mann als Vater.

Symbol	***Bedeutung***
1	Einer Entität wird **genau eine** Instanz der anderen Entität zugeordnet
c	Einer Entität wird **keine** oder **genau eine** Instanz der anderen Entität zugeordnet
m oder n	Einer Entität werden **eine** oder **mehrere** Instanzen der anderen Entität zugeordnet
cn oder cm	Einer Entität werden **keine**, **eine** oder **mehrere** Instanzen der anderen Entität zugeordnet

2.2.3 Erstellung eines ER-Diagramms

Ein ER-Diagramm wird in mehreren Schritten erstellt. Dabei ist es sinnvoll nach folgendem Muster vorzugehen:

1. Bestimmung der Entitäten
2. Bestimmung der Attribute/Eigenschaften
3. Verknüpfung der Entitäten mit Beziehungen
4. Bestimmung der Kardinalitäten

Diese Schritte werden in den folgenden Abschnitten anhand eines Beispiels näher erläutert. Es handelt sich dabei um die Abbildung eines einfachen Bestellwesens.

2.2.3.1 Entitäten

Als erstes werden die notwendigen Entitäten ermittelt und mit dem entsprechenden Symbol abgebildet:

Abbildung 10: Ermittlung der Entitäten

Wie hier zu sehen ist, werden Entitäten immer anhand ihrer Einzahl benannt.

2.2.3.2 Attribute

Sind alle Entitäten ermittelt, so werden ihnen die entsprechenden Eigenschaften zugewiesen. In diesen Schritt fallen sowohl Schlüssel- wie auch normale Attribute.

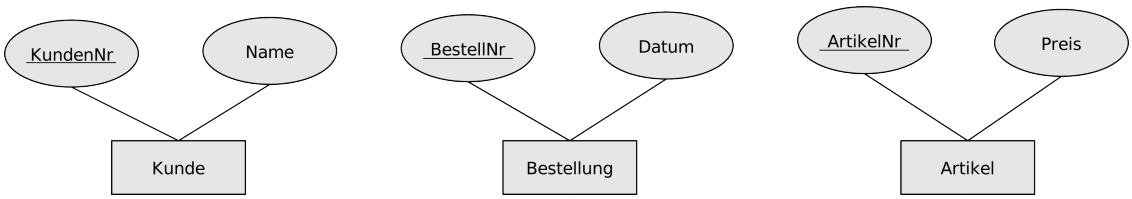


Abbildung 11: Bestimmung der Attribute

Eigenschaften werden immer mithilfe einer geraden Kante mit der zugehörigen Entität verbunden.

2.2.3.3 Beziehungen

Im dritten Schritt werden die Beziehungen zwischen den Entitäten hergestellt.

Auch die Beziehungen werden mithilfe einer geraden Kante mit den zugehörigen Entitäten verbunden. Beziehungen werden von links nach rechts bzw. von oben nach unten gelesen. In diesem Beispiel bedeutet das folgendes:

- Kunde macht Bestellung
- Bestellung beinhaltet Artikel

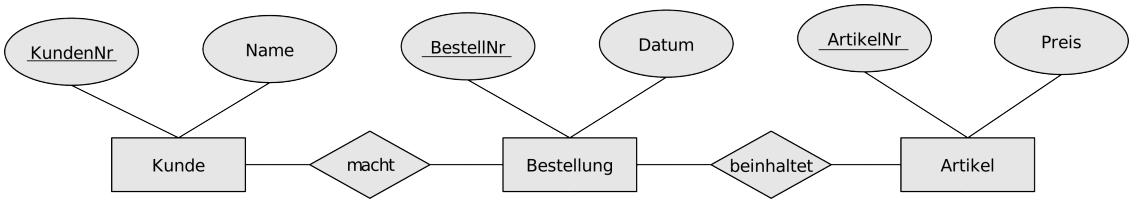


Abbildung 12: Verknüpfung der Entitäten mittels Beziehungen

Das ER-Diagramm enthält nun noch einen Fehler; will man von einem Artikel in der gleichen Bestellung mehrere Einheiten bestellen, so müsste pro Bestellung jeder Artikel mehrmals aufgeführt werden. Dieses Problem kann elegant gelöst werden, indem man der Beziehung „beinhaltet" das Attribut „Anzahl" hinzufügt. Das ER-Diagramm sieht dann schlussendlich so aus:

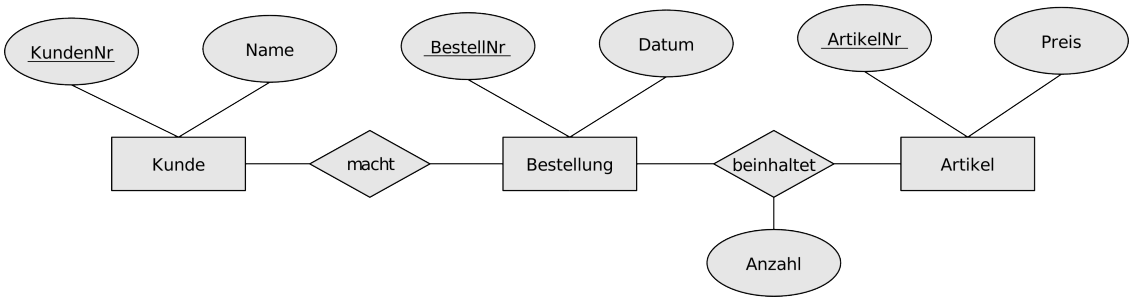


Abbildung 13: Attribute von Beziehungen

Auch rekursive Beziehungen sind erlaubt, dabei wird eine Entität mit zwei Kanten mit der Beziehung verbunden.

2.2.3.4 Kardinalitäten

Sind alle Entitäten mitsamt ihrer Eigenschaften ermittelt und mit Beziehungen entsprechend verknüpft, so gilt es zum Schluss noch die Kardinalitäten korrekt zu bestimmen. Das Vorgehen ist recht simpel; man geht immer von einer Entität aus (Entität A) und denkt sich „Wie viele Instanzen der Entität B können mit einer Instanz der Entität A in Beziehung stehen?". Das gleiche überlegt man sich aus der Sicht von Entität B.

Das Ergebnis aus dieser Überlegung fügt man jeweils in der Nähe der Beziehungskante, direkt neben die entsprechende Entität ein.

Für das verwendete Beispiel ergeben sich folgende Kardinalitäten (die Attribute wurden aus Platzgründen weggelassen):

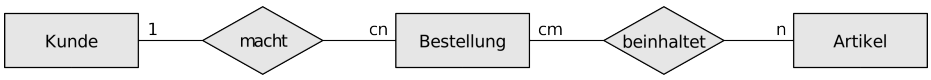


Abbildung 14: Bestimmung der Kardinalitäten

Aus diesen Kardinalitäten ergeben sich folgende Aussagen:

Beziehung *macht*:

- Ein Kunde macht keine, eine oder mehrere Bestellungen (cn)
  - Ein Kunde kann existieren, bevor er eine Bestellung getätigt hat
- Eine Bestellung ist genau einem Kunden zugeordnet (1)
  - Eine Bestellung kann nur existieren, wenn sie einem Kunden zugeordnet ist

Beziehung *beinhaltet*:

- Eine Bestellung beinhaltet einen oder mehrere Artikel (n)
  - Eine Bestellung kann nicht existieren, bevor ihr ein Artikel zugewiesen wurde
- Ein Artikel kann in keiner, einer oder in mehreren Bestellungen vorkommen (cm)
  - Ein Artikel kann existieren, bevor er in einer Bestellung verwendet wird

2.2.3.5 Alternative Darstellung

Die hier beschriebene Methode zur Angabe der Kardinalitäten (und die Darstellung der ER-Diagramme als Gesamtes) wird als „Chen-Notation" bezeichnet. In der Praxis trifft man jedoch häufig auf eine alternative Darstellungsart für die Kardinalitäten, die sog. „Martin-Notation" (auch als „Krähenfuss-Notation" bekannt).

Kardinalität	***Darstellung***	***Kardinalität***	***Darstellung***
c		cn oder cm	
1		n oder m	

2.2.4 Erweitertes ER-Modell

Die „klassische" Chen-Notation genügt oftmals den heutigen Anforderungen nicht mehr vollkommen. Deshalb gibt es verschiedene Erweiterungen am ER-Modell. Diese sind zwar weit verbreitet, gehören aber nicht zur eigentlichen Chen-Notation.

Die zwei wichtigsten Erweiterungen am ER-Modell sind die *Generalisierung* und die *Aggregation* welche im Folgenden kurz erläutert werden.

2.2.4.1 Generalisierungen

Das objektorientierte Paradigma fliesst vermehrt in die Datenbankentwicklung ein. So ist die Bildung von *Untertypen* bzw. das Zusammenfassen gleicher Eigenschaften zu einem Obertyp (= Generalisierung) in das erweiterte ER-Modell miteinbezogen worden.

Die Generalisierung wird mithilfe eines Dreiecks im ER-Diagramm dargestellt. Der Obertyp befindet sich dabei oberhalb dieses Dreiecks und ist mit einer vertikalen Kante zu ihm hin verbunden.

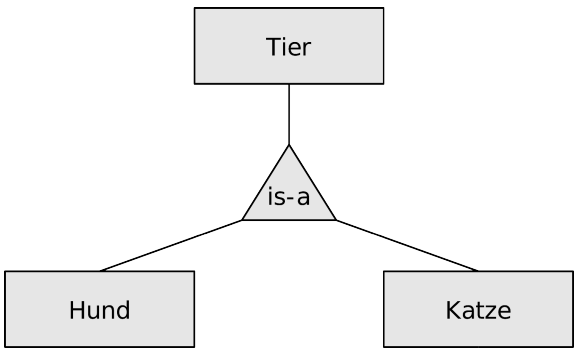


Abbildung 15: Generalisierung

Die Untertypen besitzen jeweils auch eine Kante auf das Dreieck. Diese Kanten werden mit der Grundlinie des Dreiecks verbunden.

Das Dreieck beinhaltet oft die Bezeichnung „is-a", man trifft jedoch auch auf leere Dreiecke oder auf andere Bezeichnungen.

Die Generalisierung auf Abbildung 15 sagt aus, dass die Entität *Tier* eine Generalisierung der beiden Entitäten *Hund* und *Katze* ist. Diese beiden sind ihrerseits ein Untertyp der Entität *Tier*.

2.2.4.2 Aggregation

Das Erstellen einer Beziehung zwischen zwei Beziehungen ist in der Chen-Notation nicht vorgesehen. Da Beziehungen aber im relationalen Modell (bei (c)n:(c)m Beziehungen) als Relationen umgesetzt werden, wurden Beziehungen zwischen Beziehungen mit einem weiteren Symbol eingeführt.

Im Diagramm wird die Aggregation mithilfe einer normalen Beziehungsraute dargestellt, die jedoch von einem Rechteck umgeben ist. Auf diese Weise wird dargestellt, dass die Beziehung zu einer Entität geworden ist. Das Verbinden zweier Beziehungen ist jedoch nicht erlaubt – das umranden mit dem Rechteck ist zwingend!

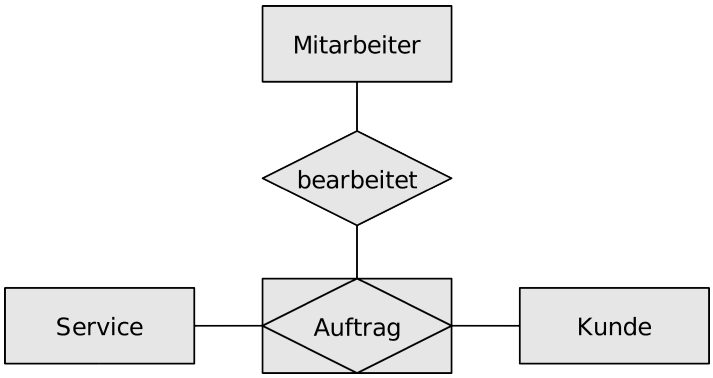


Abbildung 16: Aggregation

2.3 Textuelle Darstellung

Für das relationale Modell gibt es verschiedene Darstellungsarten. Sehr einfach und platzsparend ist die textuelle Darstellung der Relationen, welche folgender Syntax folgt:

```
Tabellenname (Schlüsselattribut, Attribut 1, Attribut 2, ... Attribut n);
```

Als erstes kommt der Tabellenname, gefolgt von einer öffnenden Klammer. Als erstes wird dann das Schlüsselattribut aufgeführt, welches unterstrichen sein muss. Nun folgen, jeweils durch Komma getrennt, die weiteren Attribute. Nach dem letzten Attribut folgt eine schliessende Klammer und ein Semikolon. Beispiele:

```
Person (id, Vorname, Nachname, Geburtsdatum);
Auto (Autonummer, Marke, Modell, Farbe);
Computer (Seriennummer, Prozessortakt, Arbeitsspeicher);
```

2.4 Überführung

Eine einfache Methode zur Überführung eines ER-Diagramms in ein relationales Modell ist die Methode „in drei Schritten". Diese Methode wird nun anhand des folgenden ER-Diagramms demonstriert:

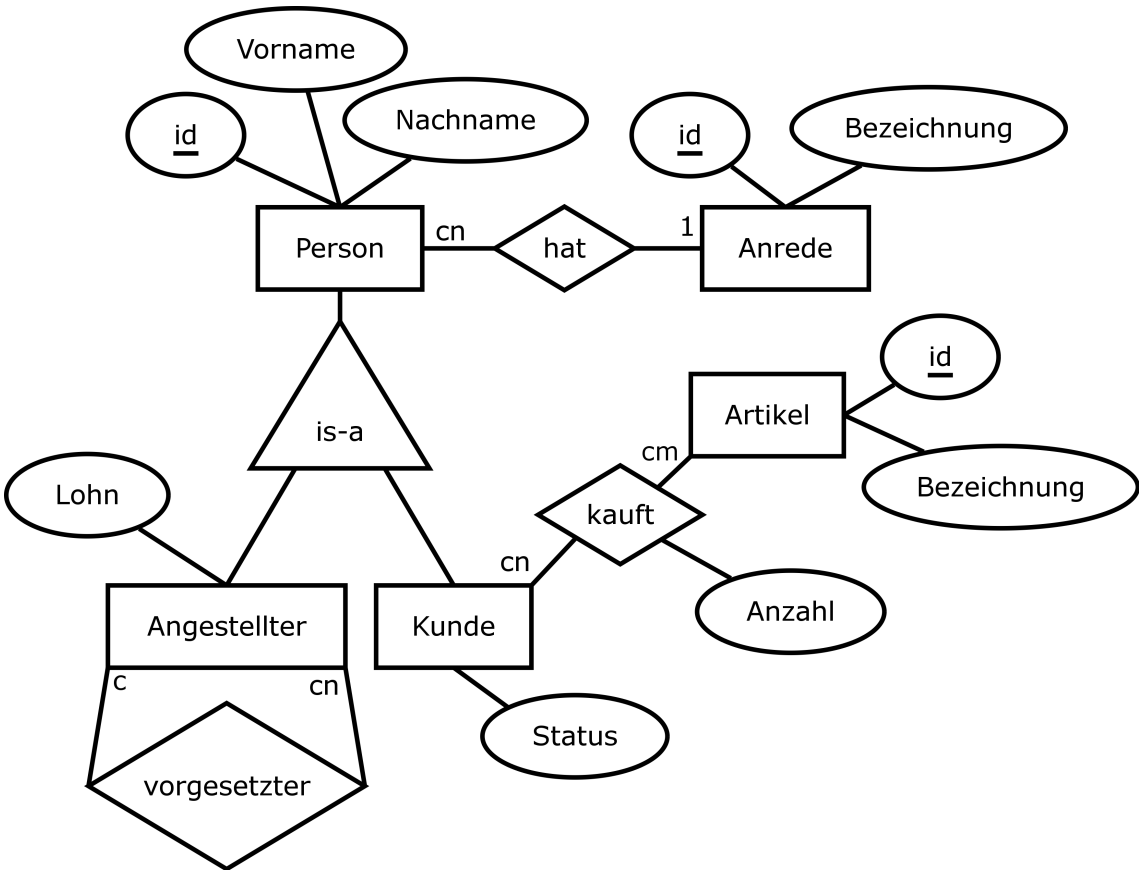


Abbildung 17: Das ER-Modell zur Überführung

2.4.1 Bildung der Entitätsmengen

Als erstes werden sämtliche Entitäten als Relationen abgebildet. Enthält das ER-Diagramm Generalisierungen, so werden diese in einem separatem Teilschritt abgebildet.

Untertypen bekommen als Primärschlüssel immer den Primärschlüssel ihres jeweiligen Obertyps. Da dieser Schlüssel jedoch auf die Entität Person bezogen ist, ist die Rede von einem sog. Fremdschlüssel. Diese werden oft mit einem vorangestellten „fk" Bezeichnet.

Schritt A) „normale" Entitäten

- `Anrede (id, Bezeichnung);`
- `Artikel (id, Bezeichnung);`

Schritt B) Entitäten innerhalb einer Generalisierung

- `Person (id, Vorname, Nachname);`
- `Angestellter (fk_Person, Lohn);`
- `Kunde (fk_Person, Status);`

2.4.2 1:n Beziehungen

Sind alle Entitätsmengen als Relationen abgebildet, so kann man sich nun den 1:(c)n bzw. den c:(c)n Beziehungen widmen.

Dies wird gemacht, indem man der Entität auf der „n-Seite" eine Spalte mit einem Fremdschlüssel hinzufügt, welche dann auf die „1-Seite" referenziert.

Die Entitäten werden also folgendermassen angepasst (Änderungen jeweils **fett** geschrieben):

- `Person (id, Vorname, Nachname, fk_Anrede);`
- `Angestellter (fk_Person, Lohn, fk_Vorgesetzter);`

Bei der Entität *Angestellter* liegt eine Besonderheit vor, es handelt sich dabei um eine *rekursive* Beziehung.

2.4.3 n:m Beziehungen

Zuletzt werden noch sämtliche (c)n:(c)m Beziehungen umgesetzt. Jede Beziehung dieser Kardinalität wird dabei als Entität umgesetzt. Der Primärschlüssel setzt sich dabei aus den beiden Primärschlüsseln beider referenzierten Entitäten zusammen. Dann folgen die weiteren Eigenschaften, die der Beziehung angehängt wurden:

- `kauft (fk_Kunde, fk_Artikel, Anzahl);`

Es ist auch möglich, dass der Primärschlüssel als zusätzliches Feld umgesetzt wird (bei obiger Umsetzung könnte ein Kunde ein bestimmtes Produkt nur ein einziges mal einkaufen).

- `kauft (id, fk_Kunde, fk_Artikel, Anzahl);`

2.5 Normalisierung

In der Datenbankentwicklung steht einem nicht immer ein sauberes ER-Diagramm zur Verfügung, um ein relationales Modell daraus zu schaffen. Weist ein Datenmodell Anomalien auf, muss dieses neu strukturiert werden. Diesen Prozess bezeichnet man als *Normalisierung*.

2.5.1 Abhängigkeiten

Ist durch den Inhalt eines Feldes (A) auf den Inhalt eines weiteren Feldes (B) zu schliessen, so spricht man von einer Abhängigkeit. Beispiel: Es gibt drei Projekte; 1 – Painkiller, 2 – Powerslave und 3 – Pilgrim. Enthält das Feld „Projektnummer" nun den Wert 2, so muss das Feld „Projekt-Bezeichnung" den Wert „Powerslave" enthalten. Das Feld „Projekt-Bezeichnung" ist also abhängig vom Feld „Projektnummer".

Neben dem Eliminieren aller Anomalien strebt man bei der Normalisierung das Erreichen von bestimmten Abhängigkeiten an. Im relationalen Modell unterscheidet man zwischen vier verschiedenen Abhängigkeiten. Diese werden nun anhand der Relation „Anstellung" erläutert (das zugrundeliegende ER-Diagramm ist in Abbildung 18 dargestellt).

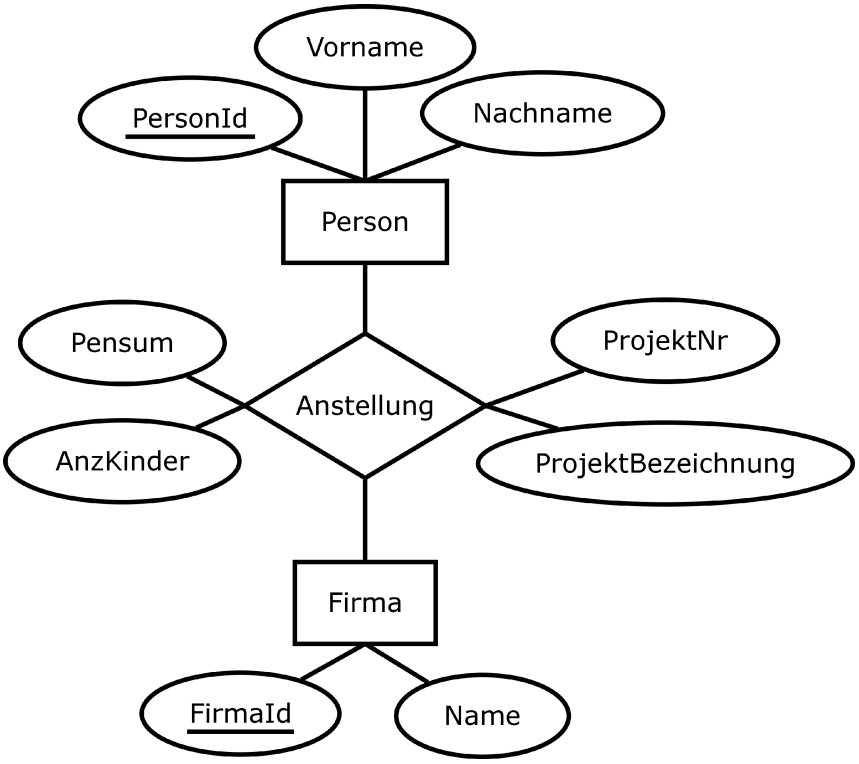


Abbildung 18: Abhängigkeiten

Führt man die Entitäten „Person", „Firma" und die Beziehung „Anstellung" in ein relationales Modell über, so ergeben sich daraus die folgenden Entitäten:

```
Person(PersonId, Vorname, Nachname);
Firma(FirmaId, Name);
Anstellung(fk_PersonId, fk_FirmaId, Pensum, AnzKinder, ProjektNr, ProjektBez);
```

Die Beiden Felder *fk_PersonId* und *fk_FirmaId* bilden gemeinsam den Primärschlüssel der Entität „Anstellung".

In diesem Beispiel lassen sich nun folgende vier Arten von Abhängigkeiten erkennen:

2.5.1.1 Funktionale Abhängigkeiten

Feld A ist von Feld B *funktional abhängig*, wenn der Inhalt des Feldes B den Inhalt des Feldes A eindeutig bestimmt.

Beispiel: *PersonId* bestimmt den Inhalt der Felder *Vorname* und *Nachname* eindeutig

Funktionale Abhängigkeiten sind **erwünscht!**

2.5.1.2 Teilweise funktionale Abhängigkeiten

Ist ein Feld nur von einem Teil eines zusammengesetzten Schlüssels abhängig, so spricht man von einer *teilweisen funktionalen Abhängigkeit*.

Beispiel: Das Feld *AnzahlKinder* ist nur von *fkPersonId* abhängig, nicht aber von *fkFirmaId*.

Funktionale Abhängigkeiten sind **nicht erwünscht**, da in der selben Relation thematisch unterschiedliche Felder vorkommen (die Anzahl der Kinder hat nichts mit der Anstellung zu tun).

2.5.1.3 Vollständig funktionale Abhängigkeiten

Sind alle nicht-Schlüssel-Felder vom gesamten Schlüssel abhängig, so liegt eine *vollständig funktionale Abhängigkeit* vor.

Beispiel: *Pensum* ist von *fk_PersonId* und *fk_FirmaId* abhängig.

Vollständig funktionale Abhängigkeiten sind **erwünscht!**

2.5.1.4 Transitive Abhängigkeiten

Ist ein Nicht-Schlüssel-Feld von einem anderen Nicht-Schlüssel-Feld abhängig, so liegt eine transitive Abhängigkeit vor.

Beispiel: *ProjektBezeichnung* ist abhängig von *ProjektNr*.

Transitive Abhängigkeiten sind **nicht erwünscht**, da beim Löschen eines Datensatzes Daten verloren gehen, die mit dem Löschvorgang nichts zu tun haben (wird eine Anstellung gelöscht, so muss nicht gleich ein Projekt gelöscht werden).

2.5.2 Normalformen

Bei der Normalisierung wird eine Relation Schritt für Schritt so umgewandelt, dass sie am Schluss keine Anomalien mehr aufweist. In der Praxis wird dies in der Regel nur bis zur dritten Normalform getrieben (es existieren noch weitere Normalformen, siehe Abschnitt 2.5.2.4).

Die folgenden Erklärungen der einzelnen Normalformen beziehen sich auf folgende nicht normalisierte Relation:

PerId	***Vorname***	***Nachame***	***AbtNr***	***AbtBez***	***ProjektNr***	***ProjektBez***	***Tätigkeit***
1	Benjamin	Breeg	1	Logistik	1 2 3	Powerslave Painkiller Pilgrim	Leiter Leiter Assistent

2.5.2.1 Die 1. Normalform

Die erste Normalform hat folgende Eigenschaften:

- Sie ist ein 2-dimensionales Gebilde aus Spalten und Zeilen (Tabelle).
- Jede Zelle enthält nur einen Wert.

Um die erste Normalform bei obigem Beispiel erreichen zu können, muss der Datensatz in

mehrere Datensätze unterteilt werden:

PerId	***Vorname***	***Nachame***	***AbtNr***	***AbtBez***	***ProjektNr***	***ProjektBez***	***Tätigkeit***
1	Benjamin	Breeg	1	Logistik	1	Powerslave	Leiter
1	Benjamin	Breeg	1	Logistik	2	Painkiller	Leiter
1	Benjamin	Breeg	1	Logistik	3	Pilgrim	Assistent

2.5.2.2 Die 2. Normalform

Eine Relation befindet sich in der zweiten Normalform, wenn folgende Eigenschaften auf sie zutreffen:

- Die erste Normalform ist gegeben.
- Es kommen keine teilweisen funktionalen Abhängigkeiten darin vor.

Vom Schlüssel *PerId* sind die Spalten *Vorname*, *Nachname*, *AbtNr* und *AbtBez* abhängig. Die anderen Spalten beziehen sich nicht auf die Person!

Die beiden projektspezifischen Spalten werden also in eine separate Relation ausgelagert:

ProjektNr	***ProjektBez***
1	Powerslave
2	Painkiller
3	Pilgrim

Die Tätigkeiten ergeben sich aus der Verbindung zwischen einer Person und einem Projekt – eine Person hat in einem Projekt eine bestimmte Tätigkeit. Diese Zuordnung wird in eine eigene Relation ausgelagert:

PerId	***ProjektNr***	***Tätigkeit***
1	1	Leiter
1	2	Leiter
1	3	Assistent

Schlussendlich bleibt von der Personentabelle noch folgendes übrig:

PerId	***Vorname***	***Nachame***	***AbtNr***	***AbtBez***
1	Benjamin	Breeg	1	Logistik

2.5.2.3 Die 3. Normalform

Um die dritte Normalform zu gewährleisten, müssen folgende Bedingungen erfüllt sein:

- Die zweite Normalform ist gegeben.
- Es kommen keine transitiven Abhängigkeiten darin vor.

Eine transitive Abhängigkeit lässt sich noch finden; das Nicht-Schlüssel-Feld *AbtBez* ist abhängig von dem Feld *AbtNr*. Als letzer Schritt kann man also nun noch die Abteilung in eine separate Relation überführen und von der Personen-Relation darauf verweisen:

PerId	***Vorname***	***Nachame***	***AbtNr***
1	Benjamin	Breeg	1

AbtNr	***AbtBez***
1	Logistik

Der Vollständigkeit halber nun die Relationen *Projekt* und *Tätigkeit*:

ProjektNr	***ProjektBez***
1	Powerslave
2	Painkiller
3	Pilgrim

PerId	***ProjektNr***	***Tätigkeit***
1	1	Leiter
1	2	Leiter
1	3	Assistent

Die Relationen befinden sich nun in der dritten Normalform. Dies ist für die Praxis in den meisten Fällen genügend. Es existieren jedoch noch weitere Normalformen.

2.5.2.4 Weitere Normalformen

- Boyce-Codd-Normalform (BCNF)
- 4. Normalform
- 5. Normalform

Diese Normalformen werden in der Praxis kaum eingesetzt, da sie sich auf das Verhältnis Normalisierung/Performance negativ auswirken. So müssen in der vierten Normalform alle Nicht-Schlüsselfelder von einem Schlüssel eindeutig abhängig sein. Dies führt dazu, dass Tabellen (neben den Fremdschlüssel-Feldern) nur zwei Spalten haben; eine eindeutige ID und den dazugehörigen Wert.

3 Konkurrierende Zugriffe

Dateninkonsistenzen können während der Entwurfsphase einer Datenbank unter Berücksichtigung der Normalformen vermieden werden. Dies wurde im vorhergehenden Kapitel ausführlich behandelt.

Im laufenden Betrieb reicht eine sauber entworfene Datenbank aber nicht aus, um eine dauerhafte Datenkonsistenz zu wahren. Diese Aussage trifft jedenfalls dann zu, wenn mehrere Benutzer gleichzeitig mit einer Datenbank arbeiten und somit gleichzeitig auf die selben Datenobjekte zugreifen wollen.

Die folgenden Abschnitte beschäftigen sich mit Mechanismen, mit deren Hilfe man Dateninkonsistenzen zur Laufzeit verhindern kann.

3.1 Transaktionen

Sobald ein Benutzer (lesend oder schreibend) auf eine Datenbank zugreift, wird eine Transaktion gestartet. Eine Transaktion kann aus einer oder beliebig vielen Operationen bestehen, die als Ganzes betrachtet werden. Können alle diese Operationen erfolgreich durchgeführt werden, so ist diese Transaktion erfolgreich. Schlägt auch nur eine einzige Operation dieser Transaktion fehl, so ist die ganze Transaktion fehlgeschlagen. Die bereits durchgeführten Änderungen müssen somit wieder rückgängig gemacht werden.

Eine Transaktion ist eine logische Operation, die in ihrer Gesamtheit entweder erfolgreich verlaufen muss oder in ihrer Gesamtheit scheitert. Eine teilweise Ausführung von Transaktionen kann zu Inkonsistenzen führen und ist somit nicht zulässig.

Eine Transaktion führt die Datenbank immer von einem konsistenten Zustand in einen anderen konsistenten Zustand über. Startet die Transaktion aus einem inkonsistenten Zustand, so wird auch der Zustand der Datenbank nach der Transaktion inkonsistent sein. Um diese Inkonsistenzen zu vermeiden, übernimmt das Datenbanksystem die Verwaltung sämtlicher Transaktionen.

3.1.1 Beispiel

Es gibt zwei Konti, Konto A und Konto B. Auf Konto A befindet sich ein Betrag von 5'000 sFr., auf Konto B sind 2'000 sFr. hinterlegt.

Konto	***Betrag (in sFr.)***
A	5'000
B	2'000

Von Konto A sollen nun 1'000 sFr. auf Konto B überwiesen werden. Dazu sind zwei Teiloperationen notwendig:

1. Abzug von 1'000 sFr. von Konto A
2. Gutschrift von 1'000 sFr. auf Konto B

Die Datenbank präsentiert sich nach der ersten Operation folgendermassen:

Konto	***Betrag (in sFr.)***
A	**4'000**
B	2'000

Der Betrag von 1'000 sFr. wurde offenbar von Konto A abgebucht.

Nun soll die Gutschrift dieser 1'000 sFr. auf Konto B erfolgen. In diesem Moment stürzt die Anwendung jedoch ab, die zweite Operation kann also nicht durchgeführt werden. Es sind also

1'000 sFr. verloren gegangen. Dies darf keinesfalls passieren!

Wären diese beiden Operationen zusammen in einer Transaktion ausgeführt worden, so hätte der Abzug der 1'000 sFr. von Konto A rückgängig gemacht werden können. Die Datenbank würde sich somit selbst nach einem Absturz noch in einem konsistenten Zustand befinden, die Transaktion könnte nach dem Neustart der Anwendung erneut durchgeführt werden. Das Ergebnis sähe dann folgendermassen aus:

Konto	***Betrag (in sFr.)***
A	**4'000**
B	**3'000**

Die beiden Operationen wurden gemeinsam in einer Transaktion ausgeführt, die Überweisung von 1'000 sFr. hat korrekt funktioniert.

Dieses Beispiel zeigt, wie wichtig die Verwendung von Transaktionen ist!

3.1.2 ACID

Eine Transaktion muss über bestimmte Eigenschaften verfügen, einige dieser Eigenschaften wurden in den bisherigen Abschnitten bereits genannt und erläutert. In der Fachliteratur fasst man diese Eigenschaften oftmals unter dem Begriff *ACID* zusammen.

ACID ist eine Abkürzung, die einzelnen Buchstaben bedeuten folgendes:

- **A**tomicity (Atomarität)
  - Eine Transaktion muss als Ganzes, Unteilbares betrachtet werden.
  - Es ist nicht zulässig, dass nur einige Operationen einer Transaktion erfolgreich durchgeführt werden, andere Operationen jedoch scheitern.
  - Eine Transaktion kann als Ganzes erfolgreich durchgeführt werden oder als Ganzes scheitern.
- **C**onsistenty (Konsistenz)
  - Wird eine Transaktion aus einem konsistenten Zustand gestartet, muss sie auch einen konsistenten Zustand auf der Datenbank hinterlassen.
  - Eine Transaktion führt die Datenbank von einem konsistenten in einen anderen konsistenten Zustand über.
- **I**solation (Isolation)
  - Transaktionen müssen voneinander isoliert ablaufen.
  - Gleichzeitig ablaufende Transaktionen dürfen sich gegenseitig nicht beeinflussen.
- **D**urability (Dauerhaftigkeit)
  - Wird eine Transaktion erfolgreich ausgeführt, müssen die vorgenommenen Änderungen dauerhaft in der Datenbank abgespeichert werden.

3.1.3 `COMMIT` und `ROLLBACK`

SQL stellt für den Programmierer die beiden Transaktionsbefehle `COMMIT` und `ROLLBACK` zur Verfügung.

Ein Programmierer möchte beispielsweise drei SQL-Operationen durchführen. Diese Operationen bilden zusammen eine Transaktion. Sind alle diese Einzel-Operationen erfolgreich durchgeführt worden, kann die Transaktion als gesamtes mit dem `COMMIT`-Befehl bestätigt werden – die Änderungen können in der Datenbank abgespeichert werden:

```
[Operation 1]
[Operation 2]
[Operation 3]
```

```
COMMIT
```

Tritt innerhalb einer Transaktion ein Fehler auf, so kann die gesamte Transaktion mit dem `ROLLBACK`-Befehl wieder rückgängig gemacht werden:

```
[Operation 1]
[Operation 2]
>> Fehler
ROLLBACK
```

Die dritte Operation wird nun nicht mehr durchgeführt, die bisherigen Änderungen werden wieder rückgängig gemacht.

3.1.4 Das Transaktionsprotokoll

Ein Datenbanksystem muss immer wissen, welche Operationen es im Falle eines abgesetzten `ROLLBACK`-Befehls wieder rückgängig machen muss. Zu diesem Zweck verfügt jede Datenbank über ein Transaktionsprotokoll. In diesem werden sämtliche bisher ausgeführten Operationen festgehalten. Rein technisch wird dieses Transaktionsprotokoll als Tabelle abgespeichert. Diese kann beispielsweise folgendermassen aussehen:

id	*trans_id*	*prev*	*next*	*operation*	*table*	*row_id*	*attribute*	*pre_val*	*post_val*
...	...	...	...	...	...	...	...	...	...
756	563	NULL	757	START					
757	563	756	758	UPDATE	Konto	1	Betrag	5'000	4'000
758	563	757	759	UPDATE	Konto	2	Betrag	2'000	3'000
759	563	758	NULL	COMMIT					
...	...	...	...	...	...	...	...	...	...

Erklärung der einzelnen Spalten:

- `id`
  - Primärschlüssel (automatischer Zähler)
- `trans_id`
  - Nummer der Transaktionen
- `prev` und `next`
  - Diese beiden Felder stellen eine doppelt verkettete Liste zwischen den einzelnen Operationen einer Transaktion dar. Dabei verweist `prev` auf die `id` der vorhergehenden Operation (innerhalb der gleichen Transaktion) und `next` auf die `id` der nachfolgenden Operation.
  - Da möglicherweise mehrere Transaktionen gleichzeitig ablaufen, kann nicht garantiert werden, dass die Operationen einer Transaktion immer schön nacheinander stehen. Aus diesem Grund wird diese doppelt verkettete Liste benötigt.
- `operation`
  - In diesem Feld steht der SQL-Befehl, die innerhalb der Operation durchgeführt wurde.
- `table`
  - Hier steht der Name der Tabelle, welche durch die Operation betroffen wurde.
- `row_id`
  - Der Primärschlüssel des Datensatzes, der durch die jeweilige Operation verändert wurde.
- `attribute`

  - Der Name der Spalte, in der ein Wert verändert wurde.
- `pre_val` und `post_val`
  - In `pre_val` wird der Wert gespeichert, der das geänderte Feld **vor** der Operation hatte.
  - In `post_val` wird der Wert gespeichert, der das geänderte Feld **nach** der Operation hatte.

Neben den einzelnen Teiloperationen werden auch der Start und das Ende einer Transaktion im Transaktionsprotokoll festgehalten. Mithilfe dieser Angaben kann jede Teiloperation einer Transaktion wieder rückgängig gemacht werden, sodass sich die Datenbank danach wieder in einem konsistenten Zustand befindet.

Ein Transaktionsprotokoll stellt für das Datenbanksystem eine zusätzliche Verwaltungsaufgabe und somit eine Performanceeinbusse dar (Overhead). Bei der Massendatenverarbeitung sollte die Möglichkeit geprüft werden, dieses Transaktionsprotokoll zu umgehen.

Das Transaktionsprotokoll ist eine sehr kritische Komponente eines jeden Datenbanksystems. Darum sollte es keinesfalls von Hand mutiert oder sogar gelöscht werden. Da der Zugriff auf das Transaktionsprotokoll sehr schnell erfolgen muss, wird dieses oftmals im Arbeitsspeicher gehalten. Datenbankserver sind somit sehr empfindlich gegen Stromausfälle.

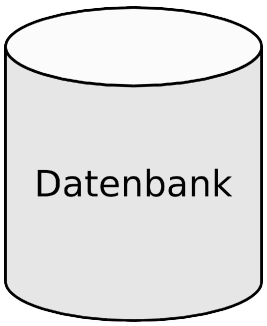
3.2 Synchronisationsprobleme

In den vorhergehenden Abschnitten wurde detailliert auf Transaktionen eingegangen und darauf, was Transaktionen für Eigenschaften haben müssen. Arbeitet eine Datenbank nicht nach dem ACID-Prinzip, so kann dies zu Synchronisationsproblemen führen. Diese werden in den folgenden Abschnitten erläutert.

3.2.1 Lost Update

Greifen mehrere Benutzer gleichzeitig schreibend auf den selben Datenbestand zu, kann das „*Lost Update*"-Problem auftreten.

Benutzer A
Lese Wert aus Tabelle
Wert = 8
Puffer = Wert 8 + 2 (10)
Aktualisiere Wert (Wert = 10)
Lese Wert aus Tabelle (13)



Benutzer B
Lese Wert aus Tabelle
Wert = 8
Puffer = Wert + 5 (13)
Aktualisiere Wert (Wert = 13)
Lese Wert aus Tabelle (13)

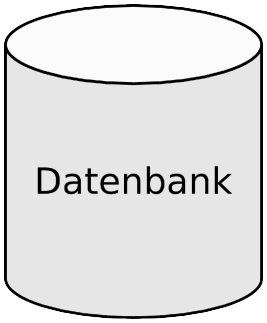
Benutzer A erwartet, dass der Wert nach seiner Änderung 10 beträgt. Benutzer B hat die Daten jedoch in der Zwischenzeit geändert, sodass der Wert auf der Datenbank tatsächlich 13 beträgt. Die Änderungen von Benutzer A wurden also „vergessen" (Lost Update). Dieses Phänomen wird auch als „wer zuletzt speichert, gewinnt" bezeichnet.

Würden die beiden Transaktionen atomar ablaufen, so würde jeder Benutzer nach seiner Aktualisierung den korrekten Wert in der Datenbank vorfinden.

3.2.2 Dirty Read

Die Problematik beim „*Dirty Read*" ist recht ähnlich zur „*Lost Update*"-Problematik. In diesem Fall geht es jedoch darum, dass beim Herauslesen von Werten auch unbestätigte Transaktionen berücksichtigt werden.

Benutzer A
Lese Wert aus Tabelle
Wert = 16
Puffer = Wert + 5 (21)
Aktualisiere Wert (Wert = 21)
Absturz (`ROLLBACK`)
Lese Wert aus Tabelle
Wert = 41



Benutzer B
Lese Wert aus Tabelle
Wert = 21
Puffer = Wert + 20 (41)
Aktualisiere Wert (Wert = 41)
Lese Wert aus Tabelle
Wert = 41

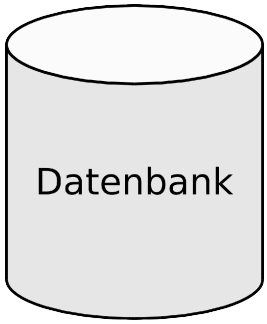
Benutzer B liest den Wert 21 aus der Datenbank, dieser wurde jedoch innerhalb der Transaktion von Benutzer A noch nicht bestätigt. Benutzer B arbeitet aber dennoch mit diesen Daten und aktualisiert den Wert auf der Datenbank. Benutzer A führt nun einen `ROLLBACK` vor, sodass der Wert eigentlich wieder 16 (Anfangswert) betragen müsste. Da die Transaktion von Benutzer B den Wert jedoch aktualisiert hat und somit die unbestätigte Änderung von Benutzer A beinhaltet, findet Benutzer A am Ende nicht den Wert 16, sondern den Wert 41 vor.

Das Problem besteht darin, dass Benutzer B Werte aus der Datenbank lesen kann, die gerade von Benutzer A bearbeitet werden. Die Transaktion läuft somit nicht isoliert ab.

3.2.3 Nonrepeatable Read

Wird innerhalb einer Transaktion ein Wert mehrmals nacheinander abgefragt, so muss immer das gleiche Ergebnis zurückgeliefert werden.

Benutzer A
Lese Wert aus Tabelle
Wert = 35
Lese Wert aus Tabelle
Wert = 50



Benutzer B
Aktualisiere Wert (Wert = 50)

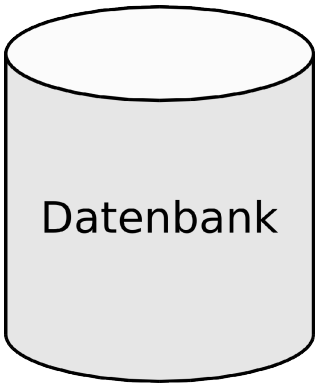
Solange Benutzer A lesend auf die Daten zugreift, dürfen diese nicht von einer anderen Trans-

aktion geändert werden. Die Transaktionen von Benutzer A und B laufen somit nicht isoliert voneinander ab.

3.2.4 Phantome

„*Phantome*" treten auf, wenn eine Transaktion laufend Daten aus der Datenbank heraus liest, mit diesen Daten vorweg Berechnungen anstellt und manche Daten in der Zwischenzeit von einer anderen Transaktion geändert werden.

Benutzer A
[Wert I = 31, Wert II = 43]
Lese Wert I aus Tabelle
Wert I = 31
Lese Wert II aus Tabelle
Wert II = 22
Puffer = Wert I + Wert II (53)



Benutzer B
Wert II = 22
Aktualisiere Wert II

Am Anfang der Transaktion beträgt Wert I 31, Wert II 43. Eine Addition aus diesen Werten würde somit 74 ergeben. Im Laufe der Transaktion wird Wert II jedoch aktualisiert und beträgt nun 22. Am Ende der Transaktion von Benutzer A beträgt die Summe dieser beiden Werte somit 53. Solange die Transaktion von Benutzer A lesend auf den Datenbestand zugreift, darf keine andere Transaktion schreibend darauf zugreifen.

Wie bei der „*Nonrepeatable Read*"-Problematik, laufen die Transaktionen nicht isoliert ab.

3.3 Sperrmechanismen

Damit Transaktionen nach den ACID-Prinzip ablaufen können und somit die geschilderten Probleme wie „Lost Updates", „Dirty Reads" usw. gar nicht auftreten, müssen Datenobjekte gesperrt werden können.

Eine Sperre bietet einer Transaktion exklusiven Zugriff auf einen Teil der Datenbank. Bevor eine Transaktion ihre Operationen ausführen kann, müssen die betreffenden Datenobjekte gesperrt werden. Erst dann können die Operationen einer Transaktion sequenziell abgearbeitet werden. Am Ende einer Transaktion (ganz egal, ob diese durch `COMMIT` oder `ROLLBACK` beendet wurde), müssen diese Sperren dann wieder entfernt werden, damit andere Transaktionen ebenfalls mit diesen Datenobjekten arbeiten können.

Moderne Datenbanksysteme verwalten Sperren automatisch, sodass der Programmierer sich nicht um diese Feinheiten kümmern muss.

3.3.1 Granularität

Unter dem Begriff *Granularität* versteht man, auf welcher Ebene die Daten von einer Transaktion gesperrt werden. Dazu gibt es verschiedene Möglichkeiten, welche im Folgenden einzeln betrachtet werden:

- Sperren auf Datenbankebene
  - Eine Transaktion sperrt die gesamte Datenbank, sodass keine andere Transaktion irgendwelche Operationen auf die Datenbank anwenden kann, solange die eine Transaktion die Datenbank nicht wieder freigegeben hat.

- Sperren auf Tabellenebene
  - Eine Transaktion sperrt nur die Tabellen, auf die sie auch wirklich zugreifen muss.
  - Alle anderen Tabellen können weiterhin von anderen Transaktionen verwendet werden.
- Sperren auf Seitenebene
  - Datenobjekte werden auf einem Datenbanksystem in sog. „Seiten" abgelegt. Eine Seite enthält einen Teil eines Datensatzes, einen Datensatz oder mehrere Datensätze.
  - Eine Transaktion sperrt nur die Seiten, auf deren Datensätze sie auch wirklich zugreifen muss.
  - Datensätze der gleichen Tabelle, die sich jedoch auf einer anderen Seite befinden, können weiterhin von anderen Transaktionen bearbeitet werden.
- Sperren auf Datensatzebene
  - Eine Transaktion sperrt nur die Datensätze, die sie auch wirklich bearbeiten muss.
  - Auf alle anderen Datensätze kann weiterhin von anderen Transaktionen aus zugegriffen werden.
- Sperren auf Feldebene
  - Eine Transaktion sperrt nur die Felder eines Datensatzes, auf die auch wirklich zugegriffen werden muss.
  - Dies ist die flexibelste aller Sperren, da in diesem Fall sogar mehrere Transaktionen gleichzeitig auf den gleichen Datensatz zugreifen können.

Die Sperre auf die ganze Datenbank ist für normale Anwendungen viel zu restriktiv. Auch die Sperre auf eine ganze Tabelle bremst das System oftmals unnötig aus, da die Transaktionen möglicherweise auf ganz unterschiedliche Bereiche der Tabelle zugreifen müssen.

Einen guten Kompromiss stellt die Sperre auf Seitenebene dar.

Die Sperre auf Datensatzebene stellt schon einen beträchtlichen Overhead dar, bei der Sperre auf Feldebene muss sich das Datenbankverwaltungssystem für jedes Feld merken, ob und von welcher Transaktion es gesperrt ist.

3.3.2 Sperrtypen

Die Granularität sagt aus, auf welcher Ebene Datenobjekte gesperrt werden. Es gibt aber auch verschiedene Möglichkeiten, **wie** diese Sperren errichtet werden.

3.3.2.1 Binäre Sperren

Bei der binären Sperre ist ein Datenobjekt entweder gesperrt, oder nicht gesperrt. Möchte eine Transaktion, egal ob lesend oder schreibend, auf ein Datenobjekt zugreifen, so wird dieses gesperrt. Während dieser Sperre dürfen andere Transaktionen weder lesend, noch schreibend auf die gesperrten Datenobjekte zugreifen.

Die binäre Sperre ist sehr restriktiv. Sie erlaubt es nicht, dass mehrere Transaktionen gleichzeitig lesend auf die gleichen Datenobjekte zugreifen können, obwohl dies keinerlei negative Auswirkung auf die Datenkonsistenz haben kann.

3.3.2.2 Exklusive und nicht-exklusive Sperren

Ein weiterer Ansatz ist die Unterscheidung in exklusive und nicht exklusive Sperren. Ein Datenobjekt kann dabei drei Zustände haben:

1. **Nicht gesperrt**
   - Transaktionen können exklusive oder nicht-exklusive Sperren auf diese Datenobjekte anlegen.

2. **Exklusiv gesperrt**
   - Diese Sperre wird errichtet, wenn **schreibend** auf Datenobjekte zugegriffen werden muss.
   - Ist ein Datenobjekt exklusiv gesperrt, dürfen keine anderen Sperren darauf eingerichtet werden (weder exklusive, noch nicht-exklusive).
3. **Nicht-exklusiv gesperrt**
   - Diese Sperre wird errichtet, wenn nur **lesend** auf Datenobjekte zugegriffen werden muss.
   - Ist ein Datenobjekt nicht-exklusiv gesperrt, so dürfen darauf beliebig weitere nicht-exklusive Sperren eingerichtet werden, jedoch keine exklusiven.

Diese Art des Lockings ist viel flexibler als die binären Sperren und wird daher auch häufiger eingesetzt.

3.3.3 Deadlock

Möchten zwei Transaktionen auf die gleiche Ressource eine Sperre errichten und tun sie dies in verschiedener Reihenfolge, so kommt es zu einem Stillstand. Dieser Stillstand wird als *Deadlock* bezeichnet.

Beispiel: Transaktion A und Transaktion B möchten die Tabellen „customer" und „address" sperren, tun dies aber in unterschiedlicher Reihenfolge.

- Transaktion A sperrt die Tabelle „customer"
- Transaktion B sperrt die Tabelle „address"
- Transaktion A möchte nun ebenfalls die Tabelle „address" sperren
  - „address" ist jedoch schon durch Transaktion B gesperrt
- Transaktion B möchte nun ebenfalls die Tabelle „customer" sperren
  - „customer" ist jedoch schon durch Transaktion A gesperrt

Die beiden Transaktionen warten nun so lange, bis die jeweils andere Transaktion die Tabelle wieder freigegeben hat. Dies ist jedoch nicht möglich, da keine der beiden Transaktionen abgearbeitet werden kann, bevor die andere ihre Sperren nicht wieder aufgehoben hat. Die Transaktionen stehen also still – ein klassischer Deadlock.

3.3.3.1 Verhinderung von Deadlocks

Es gibt drei Möglichkeiten, wie Deadlocks aufgehoben oder verhindert werden können:

1. **Timeout**
   - Muss eine Transaktion eine bestimmte Zeit auf eine Ressource warten (z.B. länger als 5 Sekunden), wird diese abgebrochen.
   - Die andere Transaktion kann ihre Sperren nun errichten und so erfolgreich abgearbeitet werden.
2. **Eine Transaktion abbrechen**
   - Sobald das Datenbanksystem einen Deadlock erkennt, sucht es sich unter den involvierten Transaktionen ein Opfer. Diese Transaktion wird dann abgebrochen.
   - Die andere Transaktion kann mit der Arbeit fortfahren.
3. **Die Sperren werden mit dem 2-Phasen Locking erstellt**
   - Eine Transaktion wird in drei Phasen aufgeteilt:
     1. Wachstumsphase: Erstellung der Sperren
     2. Gesperrte Phase: Die Transaktion kann ihre Operationen durchführen
     3. Schrumpfphase: Die Sperren werden wieder abgebaut

- Die Transaktion kann somit nur Operationen ausführen, wenn sie alle benötigten Datenobjekte sperren konnte.
- Die Datenobjekte werden in umgekehrter Reihenfolge zur Sperrung wieder freigegeben.
- Somit können zwei Transaktionen keine Sperren besitzen, die in Konflikt zueinander stehen.
- Es werden keine Daten verändert, bevor nicht sämtliche Sperren erfolgreich errichtet werden konnten.

4 SQL – Allgemeines

Die Structured Query Language (SQL) ist eine Sprache, die für den Entwurf und die Verwaltung von relationalen Datenbanken sowie die Manipulation der darin enthaltenen Daten verwendet wird.

Alle Anbieter relationaler Datenbankverwaltungsysteme haben ihre eigene Implementation von SQL, die sich mehr oder weniger vom Standard *SQL-92* unterscheiden. SQL-Server nennt sein erweitertes SQL „*Transact-SQL*", bei Oracle spricht man von „*PL/SQL*". Diese „erweiterten" Sprachen beinhalten Standard-SQL und besitzen erweiterte Funktionalitäten. In diesem Dokument wird bevorzugt der Standard-SQL-92 verwendet.

Eine enge Anlehnung an den ANSI SQL-Standard ist besonders dann wichtig, wenn die Datenbankapplikation auf unterschiedlichen Datenbanksystemen funktionieren soll. In diesem Fall sollte man auf die Verwendung der erweiterten Komponenten (z.B. Transact-SQL) verzichten.

SQL kann in folgende vier Komponenten unterteilt werden:

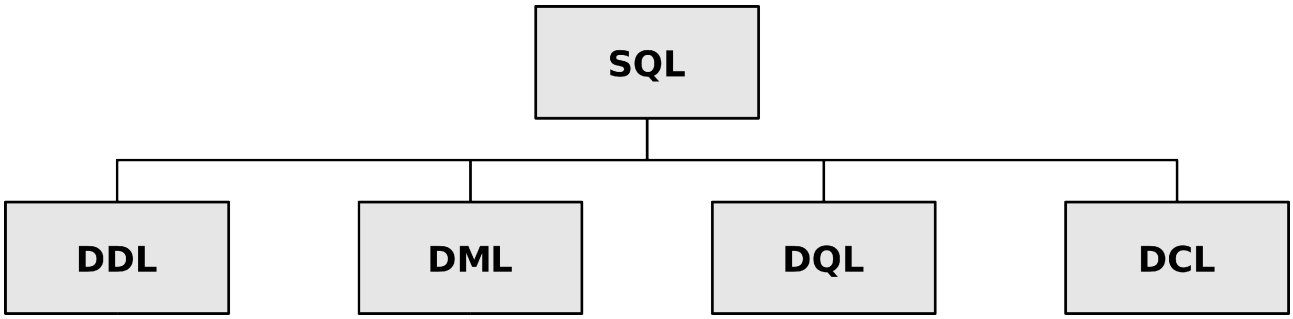


Abbildung 19: Die vier Komponenten von SQL

SQL: Structured Query-Language:

DDL: Data Definition Language:

Die Data Definition Language stellt alles zur Verfügung, was man benötigt, um eine Datenbank und deren Elemente wie Relationen und Beziehungen zu definieren, zu ändern und zu löschen.

DML: Data Manipulation Language:

Die Data Manipulation Language verfügt über Befehle mit welchen man Daten (Informationen) in eine Relation einfügen und wieder löschen kann.

DQL: Data Query Language:

Die Data Query Language verfügt über Befehle mit welchen man Informationen nach den verschiedensten Kriterien aus einer Datenbank abrufen (betrachten) kann

DCL: Data Control Language:

Die Data Control Language verfügt über Befehle um die Datenbank vor unerwünschten Einflüssen zu schützen (Berechtigungen).

4.1 Übergreifende Elemente von SQL

SQL besteht aus einer begrenzten Anzahl von Befehlen zur Datenverwaltung. Einige dieser Befehle dienen zur Datendefinition (DDL), andere zur Datenmanipulation (DML) und wieder andere zur Datenkontrolle (DCL).

Die folgende Tabelle enthält sämtliche Befehle des SQL-92 Standards:

ADD	GET DIAGNOSTICS	DEALLOCATE DESKRIPTOR
ALTER	GOTO	DECLARE CURSOR
ALTER TABLE	HAVING	DEFAULT
AVG	MAX	DELETE FROM
CHECK	OPEN	DESCRIBE INPUT
COMMIT	PREPARE	ESCAPE
CONTINUE	REVOKE	EXECUTE IMMEDIATE
COUNT(*)	SELECT	FOREIGN KEY
CREATE CHARACTER SET	SUM	GET DESKRIPTOR
CREATE DOMAIN	ALLOCATE DESKRIPTOR	GO
CREATE TABLE	ALTER DOMAIN	GRANT
CREATE VIEW	AUTHORIZATION	INSERT INTO
DEALLOCATE PREPARE	BEGIN	MIN
DECLARE CURSOR FOR	CLOSE	ORDER BY
DELETE	CONNECT	REFERENCES
DESCRIBE	COUNT	ROLLBACK
DROP	CREATE ASSERTION	SET
EXECUTE	CREATE COLLATION	UPDATE
FETCH	CREATE SCHEMA	
GET	CREATE TRANSLATION	

5 SQL-DDL

DDL ist der Teil der Sprache SQL, welcher für die Erstellung und Zerstörung von Objekten zuständig ist. In diesem Dokument werden nur SQL-Befehle berücksichtigt, die dem SQL-Standard (SQL-92) entsprechen.

5.1 SQL Datentypen

Abhängig von ihrer Herkunft unterstützen verschiedene SQL-Implementierungen eine Vielzahl von Datentypen. Die SQL-92-Spezifikation kennt jedoch nur sechs allgemeine Datentyp-Gruppen:

- genaue Zahlen
- annähernd genaue Zahlen
- Zeichenketten
- Bit-Strings
- Datetimes
- Intervalle

Innerhalb dieser allgemeinen Typen kann es mehrere Untertypen geben. Wenn man mit einer SQL-Implementierung arbeitet, die einen oder mehrere Datentypen unterstützt, die nicht in der SQL-92-Spezifikation enthalten sind, kann man die Portabilität der Datenbank erhöhen, indem man diese nicht definierten Datentypen vermeidet.

5.1.1 Numerische Datentypen

Numerische Datentypen werden für die Speicherung von Zahlen verwendet.

5.1.1.1 Numerische Datentypen für ganzzahlige Werte

Datentyp	***Wertbereich***	***Grösse***
`TINYINT`	0..255	1 Byte
`SMALLINT`	-32'768..32'767	2 Byte
`INTEGER`	-2'147'483'648..2'147'483'647	4 Byte

5.1.1.2 Numerische Datentypen für Fliesskommazahlen

Datentyp	***Wertbereich***	***Grösse***
`FLOAT`	7 signifikante Stellen	4 Byte
`DOUBLE PRECISION`	15 signifikante Stellen	8 Byte

5.1.1.3 Numerische Datentypen für Festkommazahlen

Datentyp	***Grösse***
`NUMERIC(Präzision, Skalierung)`	variiert
`DECIMAL(Präzision, Skalierung)`	variiert

5.1.2 Datentyp für Datumswerte

Mit dem Datentyp `DATETIME` kann man ein Datum und/oder eine Uhrzeit abspeichern.

Datentyp	*Wertbereich*	*Grösse*
DATETIME	Datumswerte	8 Byte

5.1.3 Datentypen für Zeichen und Text

Für Zeichenketten (Strings) gibt es Datentypen fester Länge (CHAR), Datentypen mit variabler Länge und Maximallänge (VARCHAR) und Datentypen mit flexibler Länge (TEXT, BLOB).

Datentyp	*Grösse*
CHAR(Länge)	Länge * 1 Byte
VARCHAR(Maximallänge)	Verwendete Länge * 1 Byte + 1 Byte (Speicherung des verwendeten Länge)
TEXT	variiert
BLOB	variiert

5.2 Der „CREATE TABLE"-Befehl

Die Anweisung „CREATE TABLE" dient zur Erstellung von Relationen (Tabellen) in einer Datenbank. Die Struktur der neuen Tabelle wird in Form von Spalten mit jeweils spezifischem Datentyp und Länge festgelegt. Es wird eine leere Basistabelle erzeugt und bestimmte Daten werden in die Systemtabellen des Datenbanksystems eingetragen.

Nach SQL-92 können folgende Konsistenzbedingungen für eine Tabelle festgelegt werden:

- Primärschlüssel der Tabelle (PRIMARY KEY)
- Kandidatenschlüssel (UNIQUE)
- Fremdschlüssel (FOREIGN KEY)
- Einschränkungen des Wertebereichs der Spalten (CHECK)
- Verbot von NULL-Werten in Spalten (NULL / NOT NULL)
- Spaltenübergreifende Integritätsbedingungen (CHECK)

Es folgt eine minimal-Definition eines „CREATE TABLE"-Befehls:

```
CREATE TABLE Person (
  PersonenNr INTEGER,
  Name VARCHAR(30),
  Vorname VARCHAR(30)
)
```

```
CREATE TABLE Kind (
  KinderName VARCHAR(30),
  GebDatum DATETIME,
  fk_Eltern INTEGER
)
```

Die Anweisung links erzeugt eine Tabelle namens Person, in der rechten Anweisung lautet der Name der Tabelle Kind.

5.2.1 Primärschlüssel

Auf jeder Tabelle sollte bei der Erstellung definiert werden, welches Feld den Datensatz eindeutig identifiziert (Primärschlüssel). Dies erfolgt mithilfe des Parameters PRIMARY KEY:

```
CREATE TABLE Person (
  PersonenNr INTEGER
    PRIMARY KEY,
  Name VARCHAR(30),
  Vorname VARCHAR(30)
)
```

```
CREATE TABLE Kind (
  KinderName VARCHAR(30),
  GebDatum DATETIME,
  fk_Eltern INTEGER
  CONSTRAINT pk_Kind
    PRIMARY KEY (KinderName, fk_Eltern)
)
```

Im Beispiel rechts wird ein zusammengesetzter Schlüssel erstellt. Dieser besteht aus den beiden Attributen `KinderName` und `fk_Eltern`.

5.2.2 Pflichtfelder (NULL und NOT NULL)

Der Parameter `NULL` bzw. `NOT NULL` gibt an, ob eine Eingabe für ein Feld erforderlich ist, oder nicht. Primärschlüssel müssen immer mit `NOT NULL` definiert werden.

```
CREATE TABLE Person (
  PersonenNr INTEGER
    PRIMARY KEY,
  Name VARCHAR(30) NOT NULL,
  Vorname VARCHAR(30) NULL
)
```

```
CREATE TABLE Kind (
  KinderName VARCHAR(30),
  GebDatum DATETIME NULL,
  fk_Eltern INTEGER
  CONSTRAINT pk_Kind
    PRIMARY KEY (KinderName, fk_Eltern)
)
```

5.2.3 Vorgabewerte (DEFAULT)

Weist man einem Attribut einen Standardwert zu, wird dieser automatisch beim Hinzufügen eines Datensatzes in das entsprechende Feld eingetragen. Natürlich gilt das nur, wenn die entsprechende `INSERT`-Anweisung nicht bereits einen Wert für das jeweilige Attribut übergibt.

```
CREATE TABLE Person (
  PersonenNr INTEGER
    PRIMARY KEY,
  Name VARCHAR(30) NOT NULL
    DEFAULT ('Neu'),
  Vorname VARCHAR(30) NULL
)
```

```
CREATE TABLE Kind (
  KinderName VARCHAR(30),
  GebDatum DATETIME NULL,
  Geschlecht CHAR(1)
    DEFAULT ('W'),
  fk_Eltern INTEGER
  CONSTRAINT pk_Kind
    PRIMARY KEY (KinderName, fk_Eltern)
)
```

5.2.4 Eingabe-Einschränkungen (CHECK)

Mit der `CHECK`-Klausel kann man bestimmte Einschränkungen definieren. Im folgenden Beispiel soll das Feld `PersonenNr` nur Werte zwischen 100 und 200 erlauben. Wie solche Bedingungen zu formulieren sind, ist dem Abschnitt 6.3 zu entnehmen.

```
CREATE TABLE Person (
  PersonenNr INTEGER CHECK (PersonenNr > 100 and PersonenNr < 200) PRIMARY KEY,
  Name VARCHAR(30) NOT NULL DEFAULT ('Neu'),
  Vorname VARCHAR(30) NULL
)
```

Eine `CHECK`-Klausel kann auch als `CONSTRAINT` definiert werden:

```
CREATE TABLE Person (
  PersonenNr INTEGER PRIMARY KEY,
  Name VARCHAR(30) NOT NULL DEFAULT ('Neu'),
  Vorname VARCHAR(30) NULL
  CONSTRAINT ck_bereich CHECK (PersonenNr BETWEEN 100 and 200)
)
```

5.2.5 Kandidatenschlüssel (`UNIQUE`)

Der Parameter `UNIQUE` dient dazu, Kandidatenschlüssel zu definieren. Ein Kandidatenschlüssel ist ein Attribut oder eine Attributkombination, das/die dazu geeignet ist, einen Datensatz eindeutig zu identifizieren. Zum Beispiel könnte man vorschreiben, dass in der Relation `Person` nie zwei Personen mit demselben Namen und Vornamen existieren dürfen:

```
CREATE TABLE Person (
  PersonenNr INTEGER PRIMARY KEY,
  Name VARCHAR(30) NOT NULL UNIQUE,
  Vorname VARCHAR(30) NULL
)
```

5.2.6 Fremdschlüssel

Mit der „`FOREIGN KEY`"-Klausel wird ein Fremdschlüssel von einer Tabelle auf eine andere Tabelle definiert. Dies ist vor allem zur Wahrung der referenziellen Integrität wichtig.

5.2.6.1 Referenzielle Integrität

Die referenzielle Integrität befasst sich mit der Korrektheit zwischen Attributen von Relationen und der Erhaltung der Eindeutigkeit ihrer Schlüssel.[1]

Betrachten wir dies anhand eines konkreten Beispiels. Wir haben die Tabelle Person und die Tabelle Kind. Die Tabelle Kind besitzt einen Fremdschlüssel auf die Tabelle Person. Pro Person können mehrere Kinder existieren, aber für jedes Kind kann genau eine Person existieren (c zu cn-Beziehung).

Ruedi und Mäxchen verweisen auf Peter Duss:

Tabelle „Kind"

id	name	fk_vater
1	Heidi	NULL
2	Ruedi	36
3	Gretli	NULL
4	Mäxchen	36
5	Rösali	NULL

Tabelle „Person"

id	name
34	Hans Huber
35	Max Meier
36	Peter Duss
37	David Kunz
38	Franz Dobmann

Abbildung 20: Referenzielle Integrität

Was passiert nun mit den beiden Kindern, wenn Peter Duss aus der Personen-Tabelle gelöscht wird?

Verweisen die zwei Einträge in der Kinder-Tabelle (Ruedi und Mäxchen) weiter hin auf den Personen-Eintrag mit der `id` 36, so sind inkonsistente Daten entstanden. Dies stellt eine Verletzung der referenziellen Integrität dar.

SQL-92 definiert eine Syntax für Regeln, die bei schreibenden Zugriffen auf referenzierte Primärschlüsselwerte die referenzielle Integrität gewährleisten. Darin können folgende Aktionen festgelegt werden:

- `NO ACTION` / `RESTRICT`
  - Die Veränderung kann ganz verboten werden
- `CASCADE`
  - Die Veränderung kann an den Fremdschlüsselwert weitergegeben werden
- `SET NULL`
  - Die Veränderung kann den Fremdschlüsselwert auf `NULL` setzen
- `SET DEFAULT`

1 Quelle: Wikipedia - http://de.wikipedia.org/wiki/Referenzielle_Integrit%C3%A4t

- Die Veränderung kann den Fremdschlüsselwert auf einen Defaultwert setzen

5.2.6.2 *NO ACTION / RESTRICT*

Alle Änderungen an den referenzierten Schlüsseln sind untersagt. Diese Regel gilt auch dann, wenn keine Änderungsaktion angegeben wird (Standard-Verhalten).

Definition mit dem Standard-Verhalten (implizit):

```
CREATE TABLE Kinder (
  KinderNr INTEGER,
  KinderName VARCHAR(30),
  GebDatum DATETIME NULL,
  fk_Eltern INTEGER FOREIGN KEY REFERENCES Personen (PersonenNr),
  CONSTRAINT pk_Kinder PRIMARY KEY (KinderNr)
)
```

Explizite Definition:

```
CREATE TABLE Kinder (
  KinderNr INTEGER,
  KinderName VARCHAR(30),
  GebDatum DATETIME NULL,
  fk_Eltern INTEGER,
  CONSTRAINT pk_Kinder PRIMARY KEY (KinderNr)
  CONSTRAINT fk_Eltern FOREIGN KEY (fk_Eltern) REFERENCES Personen (PersonenNr)
    ON UPDATE NO ACTION
    ON DELETE RESTRICT
)
```

Möchte man bei einer Person, die Kinder hat, die `PersonenNr` ändern, wird dies vom DBMS nicht zugelassen.

5.2.6.3 *CASCADE (Aktualisierungs- und Löschweitergabe)*

Änderungen an referenzierten Schlüsseln (`UPDATE` und/oder `DELETE`) werden an die untergeordnete Tabelle weitergegeben.

```
CREATE TABLE Kinder (
  KinderNr INTEGER,
  KinderName VARCHAR (30),
  GebDatum DATETIME NULL,
  fk_Eltern INTEGER,
  CONSTRAINT pk_Kinder PRIMARY KEY (KinderNr),
  CONSTRAINT fk_Eltern FOREIGN KEY (fk_Eltern) REFERENCES Personen (PersonenNr)
    ON UPDATE CASCADE
    ON DELETE CASCADE
)
```

Ändert man bei einer Person, auf welche Kinder verweisen, die PersonenNr, so wird der Fremdschlüssel auf die neue `PersonenNr` gesetzt. Löscht man eine Person, dann werden auch die Kinder gelöscht, die auf den entsprechenden Eintrag in der Tabelle Person verweisen.

5.2.6.4 *SET NULL / SET DEFAULT*

Die Syntax ist analog dem vorgehenden Beispiel, bis auf die Angabe `SET NULL` bzw. `SET DEFAULT`. Wird die Person gelöscht, werden die Fremdschlüssel der Kinder auf `NULL` bzw. auf den `DEFAULT`-Wert gesetzt.

```
CREATE TABLE Kinder (
  KinderNr INTEGER,
  KinderName VARCHAR (30),
  GebDatum DATETIME NULL,
  fk_Eltern INTEGER,
  CONSTRAINT pk_Kinder PRIMARY KEY (KinderNr),
  CONSTRAINT fk_Eltern FOREIGN KEY (fk_Eltern) REFERENCES Personen (PersonenNr)
    ON DELETE SET NULL
    ON UPDATE SET DEFAULT
)
```

5.2.7 Bestehende Tabellen ändern (ALTER)

Die Struktur bestehender Tabellen kann mit dem `ALTER`-Befehl angepasst werden. Dabei können Spalten hinzugefügt (`ADD COLUMN`), angepasst (`ALTER COLUMN`, `ADD CONSTRAINT`, `DROP CONSTRAINT` usw.) oder gelöscht (`DROP COLUMN`) werden:

```
1. ALTER TABLE Person ADD Anrede VARCHAR(20) NOT NULL DEFAULT('unbekannt');
2. ALTER TABLE Person DROP CONSTRAINT uq_name;
3. ALTER TABLE Person ALTER COLUMN Vorname CHAR(30) NOT NULL;
```

Der erste Befehl erweitert die Tabelle `Person` um die Spalte `Anrede`. Diese erhält sogleich einen `DEFAULT`-Wert.

Der zweite Befehl entfernt das `CONSTRAINT` namens `uq_name` von der Tabelle `Person`.

Mit dem dritten Befehl wird die Spalte Vorname dahingehend angepasst, dass ihr Datentyp fortan `CHAR(30)` lautet.

6 SQL-DQL: Der SELECT-Befehl

Die DQL (Data Query Language) ist der Teil von SQL, mit dem sich Daten aus Tabellen oder Views abfragen lassen. Die DQL umfasst den SELECT-Befehl, der im Folgenden näher erläutert wird.

Syntax:

```
SELECT Ausdrucksliste
  FROM Datenquelle
  WHERE Prädikate
  GROUP BY Prädikate
  HAVING Prädikate
  ORDER BY Sortierkriterien
```

Eine SELECT-Anweisung liefert eine Menge aus Zeilen und Spalten zurück. Diese Menge wird als *Ergebnismenge* bezeichnet.

Die einzelnen Syntaxelemente werden in den nächsten Abschnitten einzeln erläutert.

6.1 Die Ausdrucksliste

Jeder Ausdruck in der Ausdrucksliste wird bei erfolgter Abfrage zu einer Spalte in der Ergebnismenge. Es können beliebig viele Ausdrücke angegeben werden, diese werden dann jeweils durch ein Komma voneinander getrennt.

Beispiel:

```
SELECT vorname, nachname, wohnort FROM Datenquelle
```

Möchte man sämtliche Spalten einer Datenquelle abfragen, so kann dies mit der Wildcard * bewerkstelligt werden:

```
SELECT * FROM Datenquelle
```

6.1.1 Tabellennamen qualifizieren

Möchte man gleichnamige Spalten aus verschiedenen Datenquellen abfragen, so sind diese Spalten mit dem Tabellennamen zu qualifizieren. Diese Angabe hat der Syntax [tabelle.attribut] zu folgen. Gehen wir von folgender Entitätsstruktur aus:

- mitarbeiter (m_id, name, vorname, ort);
- firma (f_id, name, standort);

Die beiden Entitäten mitarbeiter und firma verfügen beide über ein Attribut (bzw. über eine Spalte) mit der Bezeichnung name. Eine Selektion über sämtliche Kriterien dieser beiden Entitäten erfordert eine Qualifikation der gemeinsamen Attribute mit dem Tabellennamen:

```
SELECT m_id, mitarbeiter.name, vorname, ort, f_id, firma.name, standort
FROM mitarbeiter, firma
```

Dabei müssen nur diejenigen Attribute mit dem Tabellennamen qualifiziert werden, die auch wirklich in beiden Tabellen vorkommen. Eine Angabe des Tabellennamen ist für die anderen Attribute zwar zulässig, jedoch nicht zwingend.

6.1.2 Ausdrücke

Oftmals ist es erwünscht oder sogar zwingend nötig, dass die Ergebnismenge nicht eins zu eins mit den Werten, die physisch auf der Datenbank abgespeichert sind, übereinstimmen sollen. Beispiele dafür wären Berechnungen, Verkettungen von Zeichenketten usw. SQL stellt für diesen Zweck Ausdrücke zur Verfügung:

```
SELECT name, preis * 1.10, abs(marge), ltrim(beschreibung) FROM artikel
```

In diesem Beispiel wird der Preis der selektierten Artikel mit dem Faktor 1.1 multipliziert (`preis * 1.10`), von der Marge des Artikels wird der Absolutwert genommen (`abs(marge)`) und bei der Beschreibung werden sämtliche führende Leerzeichen entfernt (`ltrim(beschreibung)`).

Neben den Rechenoperatoren (`+`, `-`, `*`, `/`) können verschiedene Funktionen für die einzelnen Spalten verwendet werden. Eine Übersicht über die wichtigsten Funktionen befindet sich im Abschnitt 6.8, Aggregatsfunktionen.

Spaltennamen und Ausdrücke dürfen in der Ausdrucksliste nach Belieben durchmischt werden, so lange sie mit einem Komma voneinander getrennt sind.

6.1.3 Alias-Namen für Spalten und Ausdrücke

Oftmals möchte man in der Ergebnismenge nicht die gleichen Spaltennamen sehen, wie sie auf der Datenbank definiert sind. Dieses Bedürfnis kommt vor allem bei Ausdrücken zum tragen, da der Titel einer Ausdrucksspalte gleich lautet wie der Ausdruck selbst. Im obigen Beispiel würden die Spaltennamen in der Ergebnismenge beispielsweise „`preis * 1.10`", „`abs(marge)`" oder „`ltrim(beschreibung)`" lauten.

SQL erlaubt es darum, dass für Spalten und Ausdrücke Alias-Namen vergeben werden können. Dazu wird das Schlüsselwort `AS` verwendet. Hier das vorherige Beispiel unter der Berücksichtigung von Alias-Namen:

```
SELECT name, (preis * 1.10) AS preis, (abs(marge)) AS marge,
  (ltrim(beschreibung)) AS beschreibung
FROM artikel
```

Die Spalten in der Ergebnismenge würden in diesem Fall „`name`", „`preis`", „`marge`" und „`beschreibung`" lauten. Die Ausdrücke sollten bei der Vergabe von Alias-Namen immer mit runden Klammern umgeben werden.

6.1.4 `ALL` und `DISTINCT`

Oftmals enthält eine Datenmenge Zeilenduplikate. Dazu ein Beispiel einer Tabelle mit CDs:

Tabelle „cd"

id	*bezeichnung*	*preis*
1	Painkiller	29.90
2	The Number of the Beast	15.90
3	A Matter of Live and Death	29.90
4	Tyranny of Souls	23.30

Auf den ersten Blick sind keine identischen Datensätze zu ermitteln. Selektiert man jedoch nur die Spalte `preis` aus dieser Tabelle, enthält diese Duplikate (Eintrag 1 und 3 haben den gleichen Preis). Ob diese Duplikate ausgegeben werden sollen oder nicht, kann mit dem Parameter `ALL` bzw. `DISTINCT` angegeben werden:

```
SELECT ALL preis FROM cd
```

In diesem Falle lautet die Ergebnismenge `{29.90, 15.90, 29.90, 23.30}`. Es werden also auch Duplikate ausgegeben. Die Angabe des Parameters `ALL` ist optional. Wird nichts angegeben, so werden sämtliche Duplikate selektiert.

```
SELECT DISTINCT preis FROM cd
```

Hier lautet die Ergebnismenge `{29.90, 15.90, 23.30}`, die Duplikate wurden unter Angabe des Parameters `DISTINCT` nicht selektiert.

6.2 Datenquelle (FROM)

Die Datenquelle einer Selektion ist in der Regel eine Tabelle oder eine View. Diese kann einfach mit ihrem Namen angegeben werden:

```
SELECT * FROM artikel
```

Möchte man Spalten von mehreren Tabellen/Views selektieren, so müssen selbstverständlich sämtliche verwendeten Datenquellen angegeben werden.

```
SELECT vorname, nachname, bezeichnung, preis FROM person, artikel
```

Wie bei den Spaltennamen und Ausdrücken können auch Datenquellen mit einem Alias-Namen selektiert werden:

```
SELECT * FROM TBL_ARTICLES AS artikel, TBL_PERSON AS person
```

Dies ist vor allem in der WHERE-Klausel von Nutzen, da fortan mit den Alias-Namen auf die Spalten zugegriffen werden kann (siehe Folgeabschnitt).

6.3 Die WHERE-Klausel

Bisher wurde die Auswahl der Daten nur anhand der zurückgelieferten Spalten (vertikal) eingeschränkt. Es ist jedoch auch möglich, die Auswahl horizontal einzuschränken und somit nur die Datensätze auszuwählen, die bestimmten Kriterien genügen.

Dazu ein Beispiel:

```
SELECT preis
FROM TBL_ARTICLES AS artikel
WHERE artikel.preis = 29.90
```

Es werden sämtliche Artikel selektiert, bei denen der Preis genau dem Wert 29.90 entspricht. Hier sieht man auch gleich den Vorteil eines Alias-Namens für die Tabelle. Statt den Vergleich mit TBL_ARTICLES.preis zu formulieren, kann die besser lesbare Variante artikel.preis verwendet werden. Diese Qualifikation bei der WHERE-Klausel ist nur dann von Nöten, wenn mehrere Spalten mit dem Namen „preis" zurückgeliefert werden.

6.3.1 Logische Verknüpfungen

Selbstverständlich kann man nicht nur eine einzige Bedingung definieren. Es ist auch möglich, verschiedene Bedingungen mit den Operatoren AND und OR zu verknüpfen. Die Auswertungsreihenfolge kann mit runden Klammern beeinflusst werden.

6.3.2 Vergleichsoperatoren

Die folgende Tabelle listet mögliche Vergleichsoperatoren auf:

Operator	Beschreibung	Operator	Beschreibung
!=, <>	Prüft auf Ungleichheit	=	Prüft auf Gleichheit
<	Prüft auf kleiner als	>	Prüft auf grösser als
<=	Prüft auf kleiner gleich	>=	Prüft auf grösser gleich
LIKE	Prüft, ob ein Wert einem Muster entspricht (Zeichenketten)	BETWEEN	Prüft, ob ein Wert im angegebenen Bereich liegt
IN	Prüft, ob der Wert in der angegebenen Wertmenge vorhanden ist	IS [NOT] NULL	Prüft, ob ein Wert NULL bzw. nicht NULL entspricht

Dazu einige Beispiele:

```
SELECT lagerbestand, preis
FROM artikel
WHERE lagerbestand >= 1 AND preis < 30
```

In diesem Beispiel werden sämtliche Artikel selektiert, bei denen der Lagerbestand grösser oder gleich 1 und der Preis kleiner als 30 ist.

```
SELECT bezeichnung, lagerbestand, preis
FROM artikel
WHERE bezeichnung LIKE '%bier%' AND lagerbestand BETWEEN 100 AND 200
```

Es werden sämtliche Artikel selektiert, bei denen die Bezeichnung die Zeichenkette „bier" enthält. Dazu wird der LIKE-Operator mit den Wildcards % verwendet. Diese steht für keines, eines oder beliebig viele Zeichen. Weiter existiert die Wildcard _ (underscore), welche stellvertretend für kein oder ein Zeichen steht. Zudem muss der Lagerbestand der selektierten Artikel zwischen 100 und 200 liegen.

```
SELECT bezeichnung, lagerbestand, preis
FROM artikel
WHERE bezeichnung IN ('Korn', 'Bier', 'Schnaps', 'Wein', 'Aspirin')
```

In diesem Beispiel werden sämtliche Artikel selektiert, deren Bezeichnung einem Element in der definierten Menge ('Korn', 'Bier', usw.) entspricht. Es ist auch möglich, zu diesem Zweck eine Unterabfrage zu definieren:

```
SELECT bezeichnung, lagerbestand, preis
FROM artikel
WHERE bezeichnung IN (SELECT bezeichnung FROM Trinkutensilien)
```

In diesem Fall muss die Artikelbezeichnung einem Wert (genauer der Bezeichnung) eines Datensatzes aus der Tabelle Trinkutensilien entsprechen.

Die Operatoren LIKE, BETWEEN und IN (wie auch IS NULL, IS NOT NULL) können mit dem NOT-Operator negiert werden:

```
SELECT bezeichnung, lagerbestand, preis
FROM artikel
WHERE bezeichnung NOT LIKE '%bier%'
  AND lagerbestand NOT BETWEEN 200 AND 500
  AND preis NOT IN (SELECT preis FROM wucherei)
```

Für Operatoren wie >, <, >= usw. ist die Verwendung von NOT nicht zulässig bzw. nicht sinnvoll, da es für diese jeweils ein entsprechendes Pendant gibt (!=, <> und =, > und <, >= und <=).

6.4 Gruppierungen (GROUP BY)

SQL erlaubt es, Zeilen in verschiedenen Gruppen zusammenzufassen und dann nur eine dieser Zeilen zurückzuliefern. Um eine Ergebnismenge nach bestimmten Kriterien zu gruppieren, bestehen folgende Möglichkeiten:

1. Ein Gruppierungs-Kriterium wird in der „GROUP BY"-Klausel angegeben.
2. Weitere Spalten können anhand einer Aggregatsfunktion (siehe Abschnitt 6.8, Aggregatsfunktionen) „zusammengeklappt" werden.

Eine Spalte muss sich entweder in der „GROUP BY"-Klausel befinden oder es muss eine Aggregatsfunktion (siehe Abschnitt 6.8) auf sie angewendet werden. Ansonsten ist das Ergebnis der Gruppierung nicht definiert.

Zum besseren Verständnis ein Beispiel. Wir gehen von einer Tabelle mit Personen aus (die Spalte mit dem Alter der Person wurde absichtlich mit dem englischen Wort „age" benannt, da das deutsche Wort „Alter" ein SQL-Schlüsselwort darstellt und somit nicht für Spaltennamen verwendet werden kann):

Tabelle „person"

id	*vorname*	*nachname*	*age*
1	Alice	Meier	87
2	Barbara	Huber	42
3	Claudio	Meier	12
4	Daniel	Meier	65
5	Elvira	Huber	33
6	Fabian	Birrer	56

Wir erstellen eine Selektion, um das älteste Familienmitglied zu ermitteln (wir gehen davon aus, dass der gleiche Nachname eine Verwandschaft darstellt). Dazu soll nach der Spalte `nachname` gruppiert werden. Die Selektion dazu sieht folgendermassen aus:

```
SELECT nachname, MAX(age) AS age
FROM person
GROUP BY nachname
```

In der ersten Zeile wird mit `MAX(age)` angegeben, dass jeweils der grösste Alterswert ausgegeben wird. Dieser Ausdruck macht erst im Zusammenhang mit der Gruppierung in Zeile 3 Sinn. Dabei werden sämtliche Datensätze mit dem gleichen Nachnamen zu einer Gruppe hinzugefügt. Neben dem Nachnamen wird auch das Alter selektiert. Wir erhalten als Ergebnismenge somit eine Liste aller Nachnamen mit dem jeweiligen Höchstwert aus der Spalte `age` zu diesem Nachnamen. Die Ergebnismenge sieht dann folgendermassen aus:

nachname	*age*
Birrer	56
Huber	42
Meier	87

6.5 Die *HAVING*-Klausel

Die `HAVING`-Klausel folgt der gleichen Syntax wie die `WHERE`-Klausel. Beide Klauseln haben die Funktion, die zu selektierende Datenmenge anhand bestimmter Kriterien einzuschränken. Im Unterschied zur `WHERE`-Klausel schränkt die `HAVING`-Klausel die Datenmenge nach einer vollzogenen Gruppierung ein.

1. `WHERE` schränkt die Datenmenge ein
2. Die Gruppierung (`GROUP BY`) reduziert die Ergebnismenge anhand von Gruppierungsattributen und Aggregatsfunktionen
3. `HAVING` schränkt die gruppiete Ergebnismenge weiter nach bestimmten Kriterien ein

Dazu betrachten wir am besten noch einmal das Beispiel aus dem vorhergehenden Abschnitt. Wir möchten diese Ergebnismenge nun nach Familiennamen einschränken, deren ältestes Familienmitglied das 50 Altersjahr schon hinter sich gelassen hat. Die Abfrage dazu sieht dann so aus:

```
SELECT nachname, MAX(age) AS age
FROM person
GROUP BY nachname
HAVING age > 50
```

Zu beachten ist in diesem Fall, dass für den Ausdruck `MAX(age)` ein Alias-Name definiert wurde (`age`). Der `HAVING`-Klausel erfolgt dann anhand dieses Alias-Namens. In diesem Fall wird die Ergebnismenge um den Eintrag mit dem Nachnamen „Huber" und dem entsprechenden Alter 42 reduziert, was folgende Ergebnismenge zur Folge hat:

nachname	***age***
Birrer	56
Meier	87

Nach Möglichkeit sollten so viele Kriterien wie möglich bereits in der `WHERE`-Klausel aufgeführt werden. Dies reduziert die Datenmenge, die das Datenbanksystem anschliessend gruppieren muss. In diesem Beispiel hätte die Bedingung [`age > 50`] auch schon in der `WHERE`-Klausel definiert werden können, die Ergebnismenge wäre die Gleiche geblieben.

6.6 Sortierung (`ORDER BY`)

Der `SELECT`-Befehl stellt die „`ORDER BY`"-Klausel zur Verfügung, um damit eine bestimmte Sortierung in der Ergebnismenge zu erreichen. Es können beliebig viele Spalten angegeben werden, welche wahlweise aufsteigend (`ASC`) oder absteigend (`DESC`) sortiert werden können. Betrachten wir dazu wieder das Beispiel mit der Personen-Tabelle:

Tabelle „person"

id	***vorname***	***nachname***	***age***
1	Alice	Meier	87
2	Barbara	Huber	42
3	Claudio	Meier	12
4	Daniel	Meier	65
5	Elvira	Huber	33
6	Fabian	Birrer	56

Als erstes Kriterium soll nach dem Nachnamen gruppiert werden. Da aber der gleiche Nachname bei verschiedenen Personen eingetragen ist, lässt sich die Sortierung weiter verfeinern. Die Personen sollen zusätzlich nach ihrem Alter sortiert werden. Der `SELECT`-Befehl dafür sieht dann folgendermassen aus:

```
SELECT * FROM person
ORDER BY nachname ASC, age DESC
```

Die „`ORDER BY`"-Klausel besagt in diesem Fall, dass die Spalte `nachname` aufsteigend (d.h. anhand des Alphabets, von A bis Z) sortiert werden soll. Weiter wird absteigend nach der Spalte `age` sortiert, sodass innerhalb der gleichen Familie zuerst die Familienmitglieder höheren Alters aufgelistet werden. Das Ergebnis dieser Abfrage präsentiert sich dann folgendermassen:

id	vorname	nachname	age
6	Fabian	Birrer	56
2	Barbara	Huber	42
5	Elvira	Huber	33
1	Alice	Meier	87
4	Daniel	Meier	65
3	Claudio	Meier	12

Die Angabe der Sortierreihenfolge `ASC` ist optional. Wird keine Sortierreihenfolge angegeben, so wird automatisch `ASC` angenommen. Somit ist nur die explizite Angabe von `DESC` erforderlich.

6.7 Abfragen über mehrere Tabellen (*JOIN*)

Nicht immer befinden sich alle relevanten Daten für eine Abfrage in einer einzigen Tabelle. Eine Abfrage muss somit über mehrere Tabellen hinweg erfolgen. Um die Daten der verschiedenen Tabellen aber dennoch zu einer einzigen Ergebnismenge zusammenfügen zu können, müssen die Tabellen anhand bestimmter Kriterien verknüpft werden. Zu diesem Zweck eignen sich Fremdschlüsselfelder, Datensätze verschiedener Tabellen können über Fremdschlüssel miteinander „verbunden" werden.

In den folgenden Abschnitten wird von der folgenden Tabellenstruktur ausgegangen:

Tabelle „auto"

id	marke	modell	fk_besitzer
1	Opel	Kadett	2
2	Audi	A3	3
3	VW	Golf	2
4	BMW	3er	NULL
5	Fiat	Panda	5

Tabelle „person"

id	name
1	Hans Meier
2	Sepp Birrer
3	Alice Huber
4	Barbara Kunz
5	Martin Duss

Die Assoziation zwischen den einzelnen Datensätzen ist in diesem Falle durch den Fremdschlüssel `fk_besitzer` der Tabelle `auto` auf das Feld `id` der Tabelle `person` zu erstellen.

Die einfachste Möglichkeit, diese beiden Tabellen miteinander zu verknüpfen, ist die Angabe beider Tabellennamen in der `FROM`-Klausel:

```
SELECT * FROM autos, person
```

Auf die Angabe der Ergebnismenge möchte ich in diesem Fall verzichten, da diese Selektion nichts anderes macht, als ein karthesisches Produkt zwischen den Datensätzen dieser beiden Tabellen herzustellen. Somit wird jeder Datensatz aus der Tabelle `autos` mit jedem Datensatz aus der Tabelle `person` angezeigt. Diese Ergebnismenge hat kaum eine praktische Verwendung.

Erinnern wir uns an die `WHERE`-Klausel. Diese erlaubt es nicht nur, Datensätze anhand bestimmter Bedingungen zu selektieren, wir können damit auch Datensätze „zuordnen". Betrachten wir dazu folgendes Beispiel:

```
SELECT auto.marke, auto.modell, person.name AS besitzer FROM auto, person
WHERE auto.fk_besitzer = person.id
```

Die WHERE-Klausel besagt, dass die Datensätze der Tabelle auto mit den Datensätzen der Tabelle person assoziiert werden sollen. Die Bedingung dafür lautet, dass das Feld auto.fk_besitzer jeweils dem Feld person.id entsprechen muss. Wir erhalten somit als Ergebnis eine Zuordnung der Autos zu ihren Besitzern:

marke	***modell***	***besitzer***
Opel	Kadett	Sepp Birrer
Audi	A3	Alice Huber
VW	Golf	Sepp Birrer
Fiat	Panda	Martin Duss

Diese Abfrage lässt sich auch noch anders formulieren, mit einem sog. JOIN. Die Syntax dafür lautet folgendermassen:

```
SELECT Spalten FROM Datenquelle
INNER JOIN Datenquelle ON Bedingung
```

Übertragen auf unser Beispiel ergibt dies folgenden SQL-Befehl:

```
SELECT auto.marke, auto.modell, person.name AS besitzer FROM auto
INNER JOIN person ON auto.fk_besitzer = person.id
```

Wichtig ist, dass die FROM-Klausel nur eine einzige Tabelle umfasst (Tabelle auto). Die weiteren Tabellen werden dann mit der JOIN-Klausel assoziiert. Die Ergebnismenge bleibt die Gleiche.

Dieser JOIN kann über beliebig viele Tabellen weiter gezogen werden. So könnte man z.B. Datensätze der Tabelle ort in die Abfrage miteinbeziehen (nehmen wir an, die Tabelle person enthält ein Feld namens fk_ort, das auf die Tabelle ort verweist):

```
SELECT auto.marke, auto.modell, person.name AS besitzer, ort.name AS ort
FROM auto
INNER JOIN person ON auto.fk_besitzer = person.id
INNER JOIN ort ON person.fk_ort = ort.id
```

Neben dem INNER JOIN gibt es noch weitere Möglichkeiten für JOINs. Erwähnenswert sind folgende JOINs:

- LEFT JOIN
  - Es werden sämtliche Datensätze der Tabelle selektiert, die als erstes aufgeführt wird.
  - Kann zu einem Datensatz der „linken" Tabelle eine Assoziation zu einem Datensatz der „rechten" Tabelle erstellt werden, so wird dieser ebenfalls selektiert.
- RIGHT JOIN
  - Es werden sämtliche Datensätze der Tabelle selektiert, die als zweites aufgeführt wird.
  - Kann zu einem Datensatz der „rechten" Tabelle eine Assoziation zu einem Datensatz der „linken" Tabelle erstellt werden, so wird dieser ebenfalls selektiert.

6.8 Aggregatsfunktionen

Der SQL-Standard stellt folgende Aggregatsfunktionen zur Verfügung:

Funktion	***Beschreibung***
`COUNT()`	Liefert die **Anzahl** der **Werte** in der Ergebnismenge einer `SELECT`-Abfrage bzw. einer Gruppierung.
`COUNT(DISTINCT)`	Liefert die **Anzahl** der **unterschiedlichen Werte** in der Ergebnismenge einer `SELECT`-Abfrage bzw. einer Gruppierung.
`AVG()`	Liefert den **Durchschnittswert** eines Datenfeldes einer `SELECT`-Abfrage bzw. einer Gruppierung.
`MIN()`	Liefert den **kleinsten Wert** eines Datenfeldes einer `SELECT`-Abfrage bzw. einer Gruppierung.
`MAX()`	Liefert den **grössten Wert** eines Datenfeldes einer `SELECT`-Abfrage bzw. einer Gruppierung.
`SUM()`	Liefert die **Summe der Werte** eines Datenfeldes einer `SELECT`-Abfrage bzw. einer Gruppierung.

7 Trigger

Trigger sind Funktionen, die das Datenbanksystem automatisch beim Eintreffen von definierten Ereignissen ausführt. Ein Trigger besteht aus einer oder mehreren SQL-Anweisungen und gehört immer zu genau einer Tabelle. Pro Tabelle können jedoch mehrere Trigger existieren.

7.1 Verwendung

Unterstützt ein Datenbanksystem zwar Trigger, jedoch keine automatische Lösch- oder Aktualisierungsweitergabe, so kann letzteres mit Triggern gelöst werden.

Weitere Verwendungsmöglichkeiten von Triggern:

- Zur Wahrung der Datenintegrität
  - Operationen jenseits von einfachen Lösch- oder Aktualisierungsweitergaben
- Aktualisierung von berechneten Feldern
  - Summen
  - Durchschnittswerte
- Auslösung von externen Aktionen
  - Versand von E-Mails
  - Anstoss einer Datenbanksicherung

7.2 Auslöser

Ein Trigger gehört immer zu einer Tabelle und reagiert auf bestimmte Ereignisse:

- `INSERT` (Einfügen von Datensätzen)
- `UPDATE` (Aktualisierung von Datensätzen)
- `DELETE` (Löschung von Datensätzen)

Pro Tabelle und Ereignis können beliebig viele Trigger definiert werden.

Bei der Ausführung von mehreren Triggern eines bestimmten Ereignisses ist keine bestimmte Sequenz für deren Ausführung festgelegt, man kann also nie wissen, welcher Trigger als erstes, zweites oder als letztes ausgeführt wird. Somit sollte man Operationen, die bestimmte vorhergehende Operationen erfordern, immer im gleichen Trigger definieren.

7.3 Schattentabellen

Bevor Änderungen an einem Datenbestand persistent in der Datenbank abgespeichert werden, durchlaufen die zu mutierenden Daten zunächst noch sog. *Schattentabellen*. Jede Tabelle hat zwei Schattentabellen; eine für Lösch- und eine für Einfügevorgänge. Sobald eine Transaktion mit `COMMIT` bestätigt wird, gelangen die Daten von den Schattentabellen in die eigentliche Tabelle, die Schattentabellen werden wieder geleert. Im Falle eines `ROLLBACK` werden die Daten nur aus den Schattentabellen gelöscht, ohne an der eigentlichen Tabelle etwas zu verändern.

7.3.1 Der Einfügevorgang

In eine Tabelle soll ein Datensatz eingefügt werden. Der einzufügende Datensatz wird zunächst in die Schattentabelle „`inserted`" der eigentlichen Tabelle eingefügt.

Innerhalb des Triggers kann man nun über die Tabelle `inserted` sämtliche Daten ermitteln, die mit der Transaktion in die eigentliche Tabelle eingefügt werden sollen. Der Trigger kann nun darüber entscheiden, ob die Daten definitiv eingefügt werden sollen (`COMMIT`), oder ob die Änderungen rückgängig gemacht werden sollen (`ROLLBACK`).

7.3.2 Der Löschvorgang

Aus einer Tabelle soll ein Datensatz gelöscht werden. Der zu löschende Datensatz wird zunächst in die Schattentabelle „`deleted`" der eigentlichen Tabelle eingefügt.

Innerhalb des Triggers kann man nun über die Tabelle `deleted` sämtliche Daten ermitteln, die mit der Transaktion aus der eigentlichen Tabelle gelöscht werden sollen. Der Trigger kann nun darüber entscheiden, ob die Daten definitiv gelöscht werden sollen (`COMMIT`), oder ob die Änderungen rückgängig gemacht werden sollen (`ROLLBACK`).

7.3.3 Der Aktualisierungsvorgang

Beim Aktualisierungsvorgang werden beide Schattentabellen (`inserted` und `deleted`) benötigt. Der alte Stand der geänderten Datensätze wird dabei in die Tabelle `deleted` geschrieben, die geänderten Datensätze werden vor dem eigentlichen Aktualisierung in die Schattentabelle `inserted` geschrieben.

Auch hier kann der Trigger entscheiden, ob die Änderungen an der eigentlichen Tabelle definitiv übernommen (`COMMIT`) oder rückgängig gemacht werden sollen (`ROLLBACK`).

8 Verteilte Datenbanken

Bei der klassischen Client/Server-Architektur läuft das Datenbanksystem auf einem Server, auf den dann mehrere Clients zugreifen.

Das Problem an dieser Architektur ist, dass man so einen sog. „single point of failure" hat, also eine einzige Fehlerquelle. Fällt dieser Server aus, so steht die gesamte Datenbank nicht mehr zur Verfügung.

Ausserdem genügt diese Architektur sehr grossen Unternehmen oftmals nicht, da diese dezentral organisiert sind (mehrere Filialen). Die Client/Server-Architektur spiegelt diesen Sachverhalt nicht korrekt wider. Hierzu sind verteilte Datenbanken besser geeignet.

8.1 Vor und Nachteile verteilter Datenbanken

Bei bestimmten Anwendungen haben verteilte Datenbanken klare Vorteile gegenüber der Datenverwaltung nach der Client/Server-Architektur:

+ Durch eine Verteilung der Daten kann man erreichen, dass die benötigten Daten näher beim Anwender sind, als das es bei einer monolithischen Datenbank der Fall ist.
+ Manche Daten sind ortsspezifisch. Es macht beispielsweise keinen Sinn, dass in der Berner Filiale die Kundendaten von der Zürcher Filiale einsehbar sind. So halten die einzelnen Teile der Datenbank nur die Daten, die sie auch wirklich benötigen. Dies reduziert die zu durchsuchende Datenmenge.
+ Stürzt ein einzelner Datenbankserver ab, so bedeutet dies nur für die betreffende Filiale einen Ausfall. Die anderen Filialen können wie gewohnt weiter arbeiten.
+ Der Netzwerkverkehr kann unter Umständen reduziert werden. Zudem wird keine enorme Bandbreite für einen einzigen zentralen Server benötigt.

Selbstverständlich hat eine verteilte Datenbank gewisse Nachteile gegenüber einer zentralen:

– Das System wird deutlich komplexer in der Planung, in der Wartung und auch in der Implementierung.
– Daten werden an verschiedenen Orten gehalten und auch verändert. Diese Daten müssen miteinander abgeglichen werden, was zu Konflikten führen kann. Damit keine Inkonsistenzen auftreten, muss das verteilte Datenbanksystem Mechanismen zur Behebung dieser Konflikte zur Verfügung stellen.
– Erlaubt man, dass die Anwender der verschiedenen Filialen auf allen Teil-Datenbanken Änderungen vornehmen können, so kann es zu Deadlocks über mehrere Rechner hinweg kommen.
– Es fehlen noch Standards für die Kommunikation verteilter Datenbanken untereinander. Dies stellt vor allem bei verteilten Datenbanken ein Problem dar, die aus verschiedenen Produkten bestehen (MS SQL Server, Oracle usw.).

8.2 Das Distributed Database Management System

Ein verteiltes Datenbanksystem besteht aus mehreren DBMS (Database Management Systems) und wird durch ein sog. **D**DBMS (**Distributed** Database Management System) zusammengehalten. Dieses steuert sowohl die verteilte Datenverarbeitung als auch die verteilte Datenhaltung.

Aufgaben des DDBMS:

- Optimierung der Datenbankabfragen
- Ermittlung des optimalen Zugriffsweg zu den Daten
- Sicherheitsfunktionen, Backup und Recovery
- Transaktionsverwaltung über die einzelnen DBMS hinweg
- Abstraktion der verteilten Datenbank (der Benutzer soll nur eine Datenbank „sehen")

8.2.1 Datenmanager und Transaktionsmanager

Ein DDBMS enthält sämtliche Funktionen eines „normalen" DBMS. Dazu kommen jedoch noch zwei Komponenten; der Datenmanager und der Transaktionsmanager.

Der Transaktionsmanager ist ein Programm, dass auf jedem Rechner ausgeführt wird, der auf das verteilte Datenbanksystem zugreifen muss. Dieser kümmert sich darum, dass die Anfragen der einzelnen Clients an das richtige Datenbanksystem weitergeleitet werden. Weiter muss er die Antworten der verschiedenen Datenbanksysteme entgegennehmen und für den Client entsprechend zusammenstellen.

Der Datenmanager wird auf jedem DBMS ausgeführt. Dieser kümmert sich darum, dass die Antworten (Abfrageergebnisse) an die richtigen Clients bzw. an den richtigen Transaktionsmanager gesendet werden.

8.2.2 Vollständig verteiltes Datenbanksystem

Bei einem vollständig verteilten Datenbanksystem ist sowohl die Datenhaltung als auch die Datenverarbeitung auf mehrere Rechner verteilt.

Dabei unterscheidet man zwischen homogenen und heterogenen Datenbanksystemen. Bei einem homogenen System wird auf jedem Rechner das gleiche Datenbanksystem eingesetzt (alle Rechner mit MS SQL Server, alle Rechner mit Oracle usw.). Dies steht im Gegensatz zum heterogenen Datenbanksystem, bei welchem auf den verschiedenen Rechner unterschiedliche Datenbanksysteme zum Einsatz kommen (MS SQL Server, Oracle, PostgreSQL usw. gemischt).

In der Praxis werden häufiger homogene als heterogene Datenbanksysteme eingesetzt, da derzeit noch einen Mangel an standardisierten Kommunikationsprotokollen zwischen den einzelnen Produkten besteht.

8.2.3 Replikation

Um den Zugriff auf bestimmte Daten zu beschleunigen, können diese auf verschiedene Teilsysteme *repliziert* (d.h. kopiert) werden. Replizierte Daten stellen eine weitere Gefahr für die Datenkonsistenz dar und müssen somit vom DDBMS synchronisiert werden.

Ob sich dieser Performancegewinn im Hinblick auf mögliche Dateninkonsistenzen und den Overhead beim DDBMS lohnt, ist in jedem Fall neu abzuwägen.

8.3 Transparenz

Transparenz bedeutet, dass ein Benutzer keine Kenntnis von den internen Abläufen eines Systems haben muss, um dieses anwenden zu können.

Ein Beispiel dafür wäre, dass ein Autofahrer nicht wissen muss, wie ein Motor funktioniert um eine Auto fahren zu können.

8.3.1 Stufen der Transparenz

Bei der Datenverteilung wird zwischen drei Transparenzstufen unterschieden:

1. **Vollständige Transparenz**
   - Der Datenbankanwender weiss nicht, wo sich die Daten, mit den er arbeitet, physisch befinden.
   - Der Datenbankanwender kann das verteilte Datenbanksystem so verwenden, als ob es sich um eine einzige Datenbank handeln würde.
2. **Ortstransparenz**
   - Der Datenbankanwender muss wissen, dass sich die Daten an verschiedenen Standorten befinden.
   - Wo sich diese Daten physisch befinden, muss der Datenbankanwender jedoch nicht wissen.

3. **Keine Transparenz**
   - Der Datenbankanwender muss wissen, wie die Daten verteilt sind.
   - Zudem muss der Datenbankanwender wissen, wo sich welche Daten befinden.

Die vollständige Transparenz ist nicht nur in der Anwendung am komfortabelsten, sondern auch in der Wartung. Angenommen, die verteilte Datenbank wird um einen Standort erweitert, so muss bei der Ortstransparenz sowie auch bei fehlender Transparenz jeder SQL-Befehl an die neuen Gegebenheiten angepasst werden.

Eine vollständige Transparenz kann nur durch ein sog. „*distributed Database Dictionary*" erreicht werden. In diesem ist enthalten, welche Daten sich an welchem Standort befinden. Auf sämtlichen Teilsystemen muss sich eine aktuelle Kopie des distributed Database Dictionaries befinden. Bei einer vollständigen Transparenz erhält das DDBMS somit noch eine weitere wichtige Aufgabe dazu.

8.3.2 Transaktionsverwaltung

Transparenz ist nicht nur bei der Datenverteilung wünschenswert, auch die Transaktionsverwaltung soll für den Benutzer komplett transparent ablaufen.

Werden bei einer Datenänderung die Daten von verschiedenen Teilsystemen geändert, so muss die Transaktion nicht nur für die einzelnen Teilsysteme, sondern über sämtliche Teilsysteme hinweg erfolgreich sein. Eine Transaktion ist somit bei verteilten Datenbanksystemen nur erfolgreich, wenn die Änderungen an sämtlichen Standorten durchgeführt werden können.

Im Falle eines Fehlers auf einem einzigen Teilsystem müssen die durchgeführten Operationen auf sämtlichen Teilsystemen wieder rückgängig gemacht werden.

Die Transaktionsverwaltung ist somit bei verteilten Datenbanken wesentlich komplexer als bei der zentralen Datenhaltung. Verteilte Datenbanken arbeiten darum mit einer erweiterten Transaktionsverwaltung, mit dem sog. *2-Phasen-Commit-Protokoll*.

8.3.2.1 Das 2-Phasen-Commit-Protokoll

Beim 2-Phasen-Commit-Protokoll gibt es zwei Transaktionsstufen:

1. Die Transaktionen auf den einzelnen Teilsystemen
2. Die Transaktion über das gesamte System hinweg

Weiter stellt dieses Protokoll drei neue Mechanismen zur Verfügung:

- **`DO`**
  - Die Operation (`INSERT`, `UPDATE`, `DELETE`) wird ausgeführt.
  - Die Werte vor und nach der Operation werden ins Transaktionsprotokoll geschrieben.
- **`REDO`**
  - Die Operation, die mit `DO` in das Transaktionsprotokoll geschrieben wurde, wird endgültig ausgeführt (in die Datenbank geschrieben) und bestätigt (`COMMIT`).
- **`UNDO`**
  - Die Änderungen werden mithilfe der Einträge aus dem Transaktionsprotokoll wieder rückgängig gemacht.

Sämtliche Änderungen werden zunächst ins Transaktionsprotokoll geschrieben, bevor sie in die eigentliche Datenbank aufgenommen werden. Das Transaktionsprotokoll wird jeweils persistent abgespeichert, um die Ausfallsicherheit zu erhöhen (dies wird als „*write-ahead*"-Modus bezeichnet).

Beispiel eines Ablaufs anhand eines `UPDATE`-Befehls:

1. Ein Client sendet einen `UPDATE`-Befehl, der sämtliche Teilsysteme betrifft.
2. Das DDBMS leitet diesen Befehl an sämtliche Teilsysteme weiter.

3. Die Teilsysteme schreiben die Änderungen mit `DO` in das Transaktionsprotokoll.
4. Nach einer Weile sendet das DDBMS den „`PREPARE TO COMMIT`"-Befehl an alle Teilsysteme, um diese auf den `COMMIT`-Befehl vorzubereiten.
5. Die einzelnen Systeme beantworten diesen Befehl mit `YES` oder `NO`.
   - Antwortet mindestens ein Teilsystem mit `NO`, so sendet das DDBMS den Befehl `ABORT` an sämtliche Teilsysteme. Die Teilsysteme machen die vorgenommenen Änderungen mit `UNDO` wieder rückgängig. **Die Transaktion ist fehlgeschlagen.**
   - Antworten alle Systeme mit `YES`, so wird der Vorgang fortgesetzt.
6. Das DDBMS sendet den `COMMIT`-Befehl an sämtliche Teilsysteme.
7. Die Teilsysteme schreiben die Änderungen aus dem Transaktionsprotokoll nun in der Datenbank nieder (`REDO`).
   - Hat dies funktioniert, antwortet das jeweilige Teilsystem mit `COMMITED`. **Die Transaktion wurde erfolgreich durchgeführt und ist somit abgeschlossen.**
   - Ist die Aktualisierung fehlgeschlagen, antwortet das jeweilige Teilsystem mit `NOT COMMITED`.
     - ➔ Antwortet auch nur ein Teilsystem mit `NOT COMMITED`, so sendet das DDBMS den `ABORT`-Befehl an sämtliche Teilsysteme.
     - ➔ Jedes Teilsystem nimmt seine Änderungen wieder mit dem `UNDO`-Befehl zurück.
     - ➔ **Die Transaktion ist fehlgeschlagen.**

Mit dem 2-Phasen-Commit-Protokoll kann die Transaktionssicherheit auch über verteilte Datenbanken hinweg gewährleistet werden.

8.4 Datenfragmentierung

Einzelne Tabellen können auf verschiedene Standorte verteilt werden. Dies bezeichnet mal als *Datenfragmentierung*, ein Teilstück einer Tabelle wird als *Datenfragment* bezeichnet. Hier kommt das distributed Database Dictionary zum Zuge. Dieses enthält Informationen darüber, welche Datenfragmente sich auf welchen Teilsystemen befinden.

Es gibt drei Möglichkeiten, eine Tabelle zu fragmentieren:

1. Horizontale Fragmentierung
   - Eine Tabelle wird anhand von Datensätzen aufgeteilt.
   - An jedem Standort befindet sich die gleiche Tabellenstruktur (die gleichen Spalten), jedoch nicht sämtliche Datensätze.
2. Vertikale Fragmentierung
   - Jedes Tabellenfragment enthält alle Datensätze, jedoch nicht alle Informationen (Spalten) jedes Datensatzes.
   - Der Primärschlüssel muss bei sämtlichen Fragmenten vorhanden sein.
3. Gemischte Fragmentierung
   - Eine Tabelle wird sowohl horizontal als auch vertikal fragmentiert.

Referenzen

- SQL – Grundlagen und Datenbankdesign
  - Autoren: Ulrike Böttcher und Peter Teich
  - Verlag: Herdt
- Datenbanken – Grundlagen und Design
  - Autor: Frank Geisler
  - Verlag: MITP
  - ISBN-13: 978-3-8266-1689-1
- SQL kurz und gut
  - Autor: Jonathan Gennick
  - Verlag: O'Reilly
  - ISBN: 3-89721-268-4
- Wikipedia
  - Anomalien
    - ➔ http://de.wikipedia.org/wiki/Anomalie_%28Informatik%29
  - Trigger
    - ➔ http://de.wikipedia.org/wiki/Datenbanktrigger

Abbildungsverzeichnis