"Survival Skript"

Zur Vorlesung:

Einführung in die Wirtschaftsinformatik WS00/01 (bei Prof. Dr. J. Ruhland)

-Teil 2-

Hinweis: Dieses Skript wurde privat erstellt, um für die Wirtschaftsinformatik-Klausur eine Basis zu haben, und wird anderen Studierenden zur *privaten* Vorbereitung auf die Klausur ebenfalls zur Verfügung gestellt. Eine - wie auch immer geartete - kommerzielle Nutzung des "Survival-Skriptes" stellt einen Verstoß gegen die Urheberrechte der div. benützten (und nicht immer explizit ausgewiesenen) Quellen dar. Als "Privat-Papier" kann auch keine Garantie auf Vollständigkeit und/oder Fehlerfreiheit gegeben werden. Man sollte aber ruhig davon ausgehen, das dieses Skript hier als Lerngrundlage einigermaßen zuverlässig ist. Fragen und / oder Anregungen bitte an: bast4u@gmx.net

Inhaltsverzeichnis

2. Datenorganisation und Datenmodellierung

2.0 Vorbemerkung, Begriffserklärung

➟ Was sind Daten? Welche Rolle spielen diese? Wie können Daten gespeichert werden, dass diese möglichst schnell und einfach abrufbar sind? Wie sieht es mit der Aktualität und Kompatibilität[1] der Daten aus? usw.

Daten und Datenverarbeitung

Daten sind Zeichen als (alpha)numerische oder symbolische Werte, die Informationen darstellen; aber nur dann tatsächlich als Information benutzt werden können, wenn sie "verstanden" werden.

Ein kleines Beispiel: Geburtsdaten: ✳ 15.03.75, Köln. Diese Daten sind Zeichen (Symbole, Ziffern, Buchstaben) die als symbolischer ("✳" = Symbol für "geboren am", vgl. "✝" als Symbol für "gestorben am"), numerischer (die Ziffern 1 5 0 3 7 5) und alphanumerischer (die Buchstaben K ö l n) Wert dargestellt sind. [numerisch = "nur Ziffern"; alphanumerisch = "Ziffern & Buchstaben & Sonderzeichen (, ; . : ? ! % & ` ( ) / usw.)] Diese Daten lassen sich a) abspeichern und b) "verstehen" = werden Informationen, wenn der Benutzer weiß, dass die Zeichenfolge "Geburtsdaten" nicht {stern - eins - fünf - punkt - null – drei – punkt – sieben – fünf – komma – k – ö – l – n} bedeutet, sondern {geboren am fünfzehnten März neunzehnhundertfünfundsiebzig in Köln}!

Daten sind also Grundvorraussetzung für Information und damit für Wissen, Kommunikation und Entscheidungsfindung. Somit erklären sich auch einige Begriffe: Gegenüber dem alten System der Karteikarten zur Verarbeitung von z.B. Kundendaten war die Einführung der **EDV** (= Elektronische DatenVerarbeitung) also der Datenverwaltung mittels Computer eine Revolution. Der Begriff "**Informati**k" ließe sich als "Wissenschaft der Informationsverarbeitung und Logik" übersetzen.

Daten sind also die Grundlegende Basis von 1.) betriebswirtschaftlichen Entscheidungen und 2.) dem Funktionieren von Technik. Zum letzteren ein Beispiel: Wenn Töne nicht als Schwingungen (Daten) in einer Rille (Organisation) einer Langspielplatte von einer Nadel (abrufbarkeit) abgetastet werden könnten, könnte die Hifi-Anlage daraus keine elektrischen Impulse (Datentransformation) machen und den Lautsprecher dadurch dazu veranlassen, wieder Töne zu produzieren (Datenausgabe). **Datenverarbeitung** im weiten Sinne gibt es also nicht nur im Bereich der Bits und Bytes.

2.1 Rolle der Datensicht

(?) [Ich muß ganz ehrlich gestehen, ich habe keine Ahnung, was die Schlagworte unter dieser Kapitelüberschrift im "Ruhland-Skript" überhaupt sollen. Ich habe mich daher entschlossen, diesen Punkt einfach wegzulassen, da er mir auch nicht sonderlich "wichtig" erscheint.]

---

[1] **Kompatibilität, kompatibel:** Ließe sich (ungenau) mit „passend" oder „einsetzbar" übersetzen, bzw. „nicht kompatibel" heißt soviel wie „nicht passend". Der Sinn des Begriffs erschließt sich am ehesten durch Beispiele: a) Ein Programm „myone.exe" lässt sich auf einem Windows-Rechner (PC) genauso installieren, wie auf einem Apple-Computer (Mac) ⇒ Man sagt: Das Programm „myone.exe" ist **PC- und Mac- kompatibe**l. b) Es ist nicht möglich, ein Antennenkabel (Fernseher) an eine (Strom-) Steckdose anzuschließen, denn **die Anschlüsse sind nicht kompatibe**l.

➟ Überall wo Technik mit im Spiel ist, tritt die Frage der Kompatibilität der Bauteile / Systeme / Elemente auf.

➟ Kompatibilität von Daten heißt also: Wie kann ich die Daten ablegen, dass jedes beliebige Programm diese Daten nutzen kann?.

2.2 Probleme bei dezentraler, d.h. programmspezifischer Datenhaltung

Auf gut Deutsch: **"Warum es Scheiße ist, wenn jedes Programm die Daten jeweils woanders und in der jeweils Programmspezifischen Art und Weise abspeichert."**

Beispiel: In einem Betrieb gibt es jeweils eine Software für

a) die Lagerverwaltung (lager.exe)
b) die Abteilung Kundenaufträge (auftrag.exe)
c) die Marketingabteilung "Kundenpflege" (kunden.exe)
d) die Personalplanung (personal.exe)
e) die Lieferabteilung (liefer.exe)
(usw.)

Der Kunde Peter Arndt bestellt 1.00 Stück Schrauben des Typs 34X5 zu insgesamt 700 DM. Was passiert?

In der Auftragsabteilung speichert das Programm auftrag.exe folgende Daten{Arndt, Peter / 01.11.00 / 1.000 Stck / 34X5}, die Lagerverwaltung gibt in lager.exe ein {Abgang: Nr. 0274 / 1.000 Stck / 34X5}, Personalplanung notiert in personal.exe {Maier [für] Nr. 0274 / 01.11.00} und die Marketingabteilung speichert mit kunden.exe – da Peter Arndt schon früher div. Dinge gekauft hat – die Daten { Peter Arndt / Meisenweg 6 / 99423 Weimar / Summe aller Bestellungen: 2.500 DM}. Das heißt: Obwohl es sich im Prinzip um einen Vorgang handelt sind jetzt insgesamt vier Datenbanken mit teilweise denselben Informationen gespeichert bzw. geändert worden.

Werden die Schrauben jetzt ausgeliefert, muß auch in der Datenbank der Lieferabteilung ein Datensatz mit { Peter Arndt / Meisenweg 6 / 99423 Weimar / 01.11.00 / 1.000 Stck / 34X5} angelegt werden, obwohl diese Daten z.B. in kunden.exe und auftrag.exe bereits existieren (siehe Unterstreichnung). Müssen bereits vorhandene Daten erneut abgelegt werden, spricht man von **Redundanz der Daten** (überflüssige Datenwiederholung).

Datenredundanz ist nicht nur mit höherem Speicherplatz (und damit –kosten) und höherem Aufwand bei der Dokumentation verbunden, sondern erschwert besonders die Aktualisierung und Absicherung von Daten: Zieht z.B. der Kunde um, kann es vorkommen, dass die Marketingabteilung in kunden.exe schon die neue, die Lieferabteilung in liefer.exe noch die alte Adresse hat. Es liegen also **inkonsistent**e, d.h. logisch widersprüchliche, Datenbestände vor (Meisenweg 6 *vs.* Rosengasse 9).

Solch eine **dezentrale Dantenhaltung** führt aber auch zu Problemen beim Schutz vor Datenvernichtung, dem Crash, (Welche Abteilung soll wie oft von was eine Sicherungskopie machen?) und dem **Schutz der Daten** (Absicherung): Es wäre logischerweise sehr viel einfacher, EINE Datenbank vor Hackerangriffen zu schützen, als fünf Datenbanken (Eine Bank ist auch sicherer als fünf Geldtransporter :-).

Ein weiteres Problem ist der **ungenutzte Informationsgehalt** der Daten (Die Marketingabteilung weiß nicht, dass die Lagerverwaltung weiß, wann der Kunde was bestellt hat, usw.).

2.3 Das Konzept der Datenbank

Die Lösung der in 2.2 beschriebenen Probleme heißt **datenbankorientierte Datenorganisatio**n. Hier wird zwischen **logischen Dateien** und **physischen Dateien** unterschieden. Wichtig ist aber vor allem, daß alle Daten **zentral in einer** Datenbank abgespeichert werden, und nicht mehr jedes Programm seine eigenen "Datenbänkchen" hat.

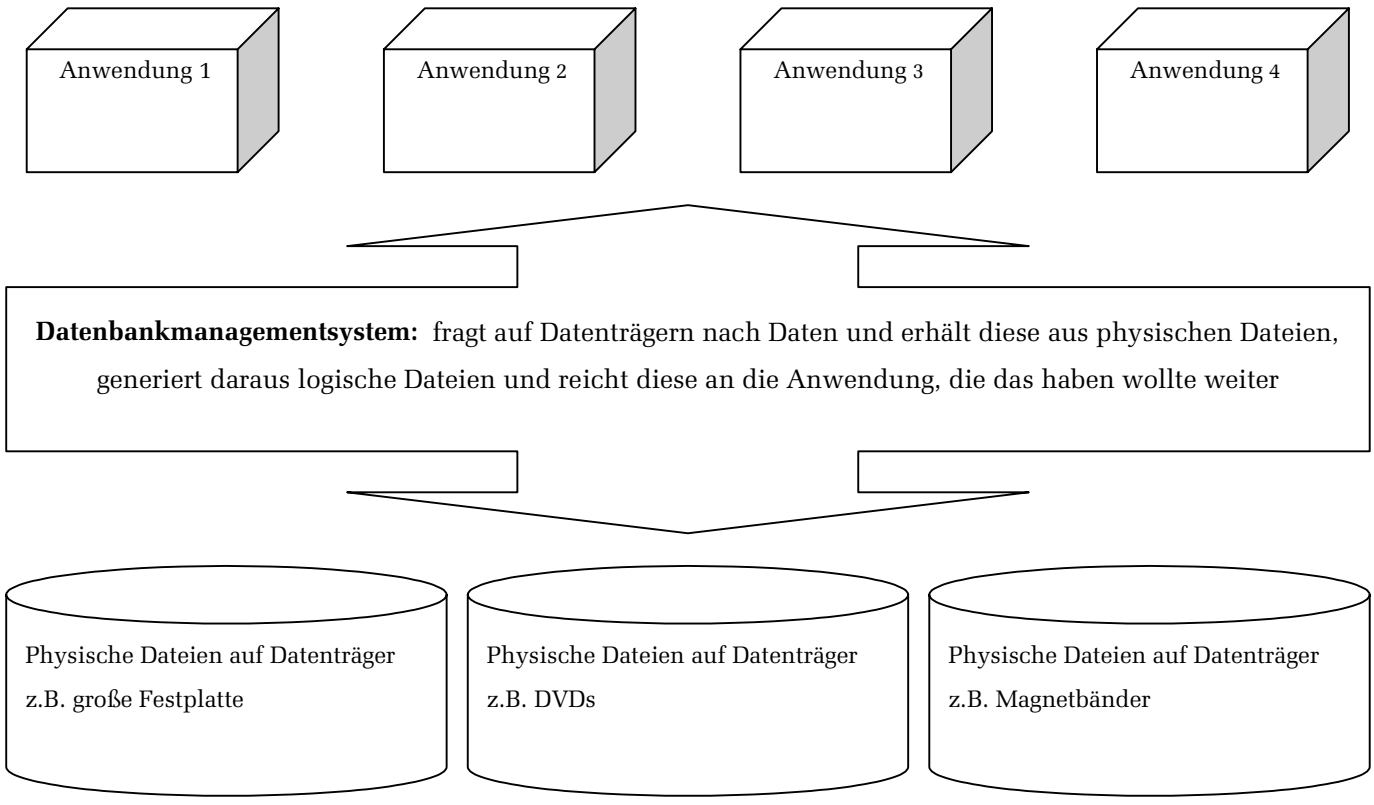
Logische Dateien bedeutet das z.B. für die Versandabteilung die Daten in der logischen Reihenfolge {[Name] / [Anschrift] /

[Artikel] / [Menge]} in der Datei auftrag.doc vorliegen, während bei der Marketingabteilung Kundenpflege eine Datei die Daten {[Name] / [Anschrift] / [Kunde seit] / [Summe Wert aller Bestellungen]} enthält. **Physische Dateien** bedeutet, daß es Dateien gibt, die mit den darin enthaltenen Daten auf einem Datenträger gespeichert – also an einem physischen Ort – sind und evtl. nach ganz anderen Kriterien geordnet werden. (ganz besonders Wichtige & sich nur selten ändernde Daten z.B. auf einer CD-ROM; Daten die sich ständig ändern auf Festplatte oder Diskette.)

Das heißt: Jede Information muß nur einmal an einem ganz bestimmten Platz (in einer physischen Datei) abgespeichert werden. Benötigt nun ein Programm eine ganz bestimmte logische Datei, dann "fragt" dieses Programm beim **Datenbankmanagementsystem (DBMS)** nach. Das **DBMS** erstellt nun anhand der Daten aus den physischen Dateien eine der Anfrage des Programms entsprechenden logische Datei zusammen und schickt diese an das Pragramm.

Oder anders ausgedrückt: Das Konzept der Datenbank läßt sich vergleichen mit dem Prinzip eines Zentralarchives samt Archivar in einem Unternehmen. Fragt nun die Marketingabteilung [~Programm] nach z.B. den Adressdaten der Kunden, die in den letzten zwei Wochen etwas bestellt haben, dann sucht der Archivar [~DBMS] in seinen Ordnern [~physische Dateien] (die er irgendwie, nach einem für ihn logischen System abgelegt hat) nach den entspr. Informationen, schreibt diese auf ein Blatt Papier [~logische Datei] und schickt das an die Marketingabeteilung.

Genauso wie der Archivar in meinem Beispiel, ist auch das DBMS sozusagen eine Art "Vermittler" zwischen der Ebene der physisch gespeicherten Daten (oder zwischen den Datenträgern) und den Programmen. Man sagt auch: Das **DBMS** sitzt als "**Schicht**" zw. den (physisch gehaltenen) **Daten** und den **Anwendungen**. Das Prinzip ähnelt ein wenig der Beziehung von Anwendungen - Betriebssystem - Hardware (vgl. Kapitel 1).



Die Vorteile eines solchen Systems liegen auf der Hand: Die Daten einer Datenbank haben **übergreifende Geltung**, das heisst sie sind **unabhängig** von den einzelnen Programmen, die auf sie zugreifen. Die Daten werden nur **einmal** an **einem** ganz bestimmten Ort **dauerhaft** gespeichert (redundanzfrei und konsistent), eine Änderung (z.B. Adressänderung) muss auch nur noch **einmal** vorgenommen werden und trotzdem haben haben **alle Programme** diese **aktualisierten Daten** zur Verfügung. Das System ist also (insbesondere bei großen Datenmengen) auch sehr effizient. Trotz (oder gerade wegen) der **zentralen Verwaltung** von Daten können mehrere Programme gleichzeitig (**simultan**) die Datenbank nutzen (jeder kriegt seine eigene

logische Datei) und **jedes Programm** kriegt **vollständige Daten** (d.h. nicht weniger - aber auch nicht mehr -, als angefragt wurde = benötigt wird).

Zusätzlich lassen sich sehr einfach ein paar **Sicherheitsvorkehrungen** treffen:

In der Regel dokumentiert das DBMS alle Vorgänge in einer zentralen log-Datei. (D.h. jeder Abfragevorgang aber auch jeder Speicherungs- und Änderungsvorgang ist dokumentiert.) Solch eine Dokumentation erlaubt:

1.) Eine **erhöhte Konsistenzsicherung** der Daten. Bsp. Soll-Haben-Buchung: "A an B, Schrauben 10.000 DM". Nun stürzt das Buchungssystem & das DBMS mitten im Vorgang ab. Nachdem es wieder eingschaltet ist, merkt das DBMS - aufgrund des Eintrags in der Dokumentation - dass da gerade ein Vorgang nur "halb" ausgeführt wurde (nur Abbuchung, Gegenbuchung fehlt). Es liegt eine "inkonsistenz" vor = "logischer Widerspruch" (Weil das abgebuchte Geld "muss" ja irgendwo "draufgebucht" sein, kann sich ja nicht in Luft auflösen). Das DBMS macht - da es merkt "hoppla Vorgang war nicht vollständig abgeschlossen - den ganzen Vorgang wieder Rrückgängig. Der Buchhalter kann nun die Buching nochmals - und diesmal vollstänig - ausführen
2.) **Einfache Persistenzgarantie**, also Sicherungsmechanismen gegen Crash (Es reicht eine Sicherungskopie der DB zu erstellen, anstatt für jede einzelne Anwendung eine eigene wie in 2.2) und die Dateien können anhand der log-Datei evtl. wieder rekonstruiert werden.
3.) Datensicherung gegen "**Malice**", also Angriffe von aussen ("Hacker"). Da alles protokolliert / dokumentiert wird, sind Angriffe besser nachvollziehbar => Gefahr für Hacker enttarnt zu werden steigt. Aber schon allein die zentrale Verwaltung der Daten durch das DBMS erhöht die Datensicherung gegen "Malice": Die Daten können **einheitlich verschlüsselt** abgelegt werden, und der Zugang (bzw. die Anfrage nach log. Dateien) kann einheitlich durch **Zugangskonzepte** gesichert werden (also Passwörter, Zugriff nur vom Intranet aus möglich, jedes Programm darf nur ganz bestimmte Dateien abfragen, etc.).

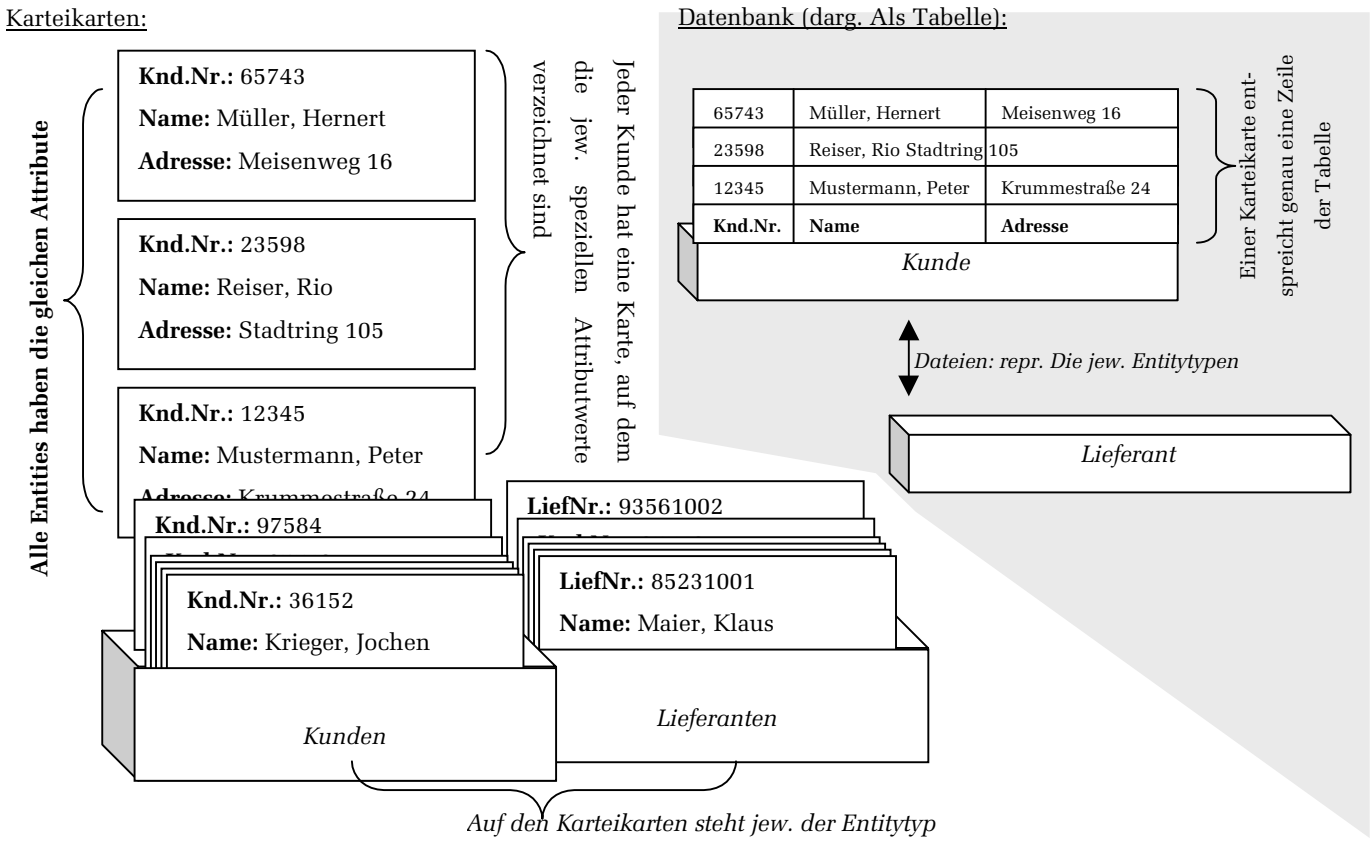
2.4 Datenmodellierung

Nachdem klar ist, **wie** Daten **abgefragt** und **wo** die Daten **gespeichert** sind; ist zu klären **wie** die Daten **gespeichert** werden sollen. Also nach welcher logischen Systematik werden die Daten organisiert (im Sinne von verteilt) und abgelegt? Vgl. System von Karteikartenarchiv: Wie sind die Karten (~Datei) organisiert, was steht auf den Karten (~Daten einer Datei). Bleiben wir beim Beispiel Karteikartensystem, und zwar speziell für Kundendaten und kommen zum ersten wichtigen Begriff: Objekte bzw. Entities!

Herr Dr. Ruhland definiert dies in bestechend eindeutiger Weise als"datenmäßige Abstraktion eines Objekts der realen Welt für das jeweilige Anwendungsgebiet". Dies ist nichts anderes als eine komplizierte Definition für einenvöllig simplen Sachverhalt: Ein Objekt, bzw. ein **Entity** ist z.B. im Karteikartenbeispiel der Kunde "Herr Arnold"; der **Entitytyp** nach dem die Karteikarten organisiert sind ist "Kunde". Anders ausgedrückt: "Objekte [**Entities**] sind (...) individuelle oder identifizierbare Exemplare von Dingen, Personen oder Begriffen der realen (oder der Vorstellungs-)Welt. (...) Objekte mit ähnlichen Eigenschaften lasen sich zu **Klassen**, sogenannten Objekt**typen** - oder **Entitytypen** - zusammenfgassen (z.B. alle Lieferanten) Die [wichtigen] Eigenschaften des einzelnen Entities oder eines Entitytyps werden mit **Attributen** charakterisiert. (Der **Kunde**(Entitytyp) **Mayer**(EntitY) hat den **Namen**(Attribut) ...) Zwischen Entities können Beziehungen bestehen (Lieferant Y liefert Artikel X), die wiederum zu **Beziehungstypen** zusammengefasst werden können."

Zurück zum **Beispiel Karteikarten**: Die Daten des **Entitytyps** "Kunde" sind in Karteikarten erfasst; und zwar so, dass **jedem Entity** (jedem einzelnen Kunden) **eine** Karteikarte zugewiesen ist, auf dem die **jew**. **Werte** der **Attribute** {Name}{Adresse}{Knd.Nr.} aufgedruckt sind. **Alle** Kunden (also alle Entities eines Entitytyps) haben die **gleichen Attribute** (= von mir erfassten Merkmale), nämlich {Name}{Adresse}{Knd.Nr.}; aber logischerweise haben die Kunden **nicht** die **gleichen** Attributs**werte**. (= Ausprägungen der Attribute). *Hätten sie alle die gleichen Merkmale, wären's ja dieselben ;-)*

Diese Logik der "Karteikartenorganisation" muss man jetzt nur noch eins-zu-eins in die Welt der "elektronischen Datenorganisation" übertragen" und schon hat man das Grundprinzip der "Datenmodellierung als Basis des Datenbankdesigns" begriffen:



Und nun lass ich wieder ein wenig eine **Quelle** "für mich" erklären: "(Zitat) Bei der Entwicklung einer Datenbank wird das Ziel angestrebt, einen Ausschnitt der realen Welt **abzubilden**, d.h. Teile unserer Umgebung mit Daten zu beschreiben. In diesem kleinen Weltausschnitt existieren eine Vielzahl von Objekten, wie Personen, Unternehmungen, Produkte oder Dienstleistungen. Diese stehen in unterschiedlichen Beziehungen zueinander. Ein Beispiel für eine **Beziehung** wäre, ein Student „entleiht“ sich ein Buch aus der Bibliothek. Oder ein anderes Beispiel wäre, ein Kunde bestellt Waren. Um die Bedeutung dieser Objekte und Beziehungen erfassen zu können, d.h. abstrakt zu beschreiben, bedient man sich den sogenannten semantischen **Datenmodellen**.

Ein Beispiel, wie so ein semantisches Netz bzw. ein **Entity-Relationship-Modell** aufgebaut sein könnte, wäre folgende Darstellung: Das Buch „Harry Potter“ hat einen Verfasser, die Bibliothek hat dieses Buch in der Ausleihe und das Buch ist ein Bestseller. Das Entity-Relationship-Modell spricht in seinem Ansatz von **Entitytypen** wie z.B. *Student* und *Buch* zwischen denen eine **Beziehung** besteht die hier den Vorgang der Entleihe wiederspiegelt.

Diese Methoden helfen uns die Komplexität, welche die Realität mit sich bringt zu reduzieren. Mit anderen Worten, sie helfen uns, uns auf die Gegenstände und ihre Eigenschaften zu konzentrieren, die für unsere Aufgabenstellung wesentlich sind. Zum Beispiel sind die Augenfarbe, die Haarfarbe und das Körpergewicht eines Buchautors für den begeisterten Leser sicherlich interessante Merkmale, aber nicht gerade die Eigenschaften die in einem Bibliotheksprogramm gespeichert werden sollten, um über eine Autorenrecherche bestimmte Buchtitel ausfindig zu machen. Nützliche Eigenschaften wären hier eher der Titel des Buches, der Verlag oder die ISBN-Nummer.

Es gibt auch Hilfsmittel in der semantischen Datenmodellierung. Sie helfen uns, die Objekte die uns interessieren abzugrenzen und durch relevante Eigenschaften zu charakterisieren. Dies geschieht meist mit Hilfe einer **grafischen Unterstützung**. Das Ergebnis der semantischen Datenmodellierung soll als Grundlage für das logische Datenbankmodell bzw. für das konzeptionelle Schema dienen.

Die Erstellung einer Datenbank vollzieht sich in drei Schritten. Erstens in der **Datenmodellierung**, zweitens im logischen **Datenbankmodell** und drittens in der **Datenbeschreibung** in dem zur Anwendung kommenden Datenbanksystem.

Im ersten Schritt werden die Objekte der realen Welt, die für unsere Aufgabenstellung relevant sind, mit ihren **Beziehungen** untereinander in **abstrakter** Weise beschrieben, d.h. **modelliert**. Um diesen Vorgang **visuell** zu unterstützen, werden z.B. semantische Datenmodelle mit Hilfe des Entity-Relationship-Modells erstellt.

Im **zweiten** Schritt wird aus der Datenmodellierung das **logische Datenbankmodell** erstellt. Man spricht hier auch von der sogenannten konzeptionellen Phase. Auf **welches** Datenbankmodell man dabei abzielt, liegt daran, welches Datenbankverwaltungssystem später zum Einsatz kommen soll. Bei der Konzeptionierung des logische Datenbankmodells gibt es schon Ausrichtung auf das Datenbankverwaltungssystemen das man später verwenden will. Man spricht hier je nach verwendeten Datenbankverwaltungssystem oder auch Database Management Systems von einem Hierarchischen Modell, einem Netzwerkmodell oder von einem **Relationenmodell**.

Im **dritten** Schritt wird das logische Datenbankmodell dann in der Datenbeschreibungssprache des Zielsystems oder auch **Data Description Language** genannt, **umgesetzt**. Das Modell wird zum Beispiel in einem Datenbankprogramm wie Access implementiert. Hierbei werden **Tabellen** angelegt, wobei durch die Felddefinition die Datenbeschreibung erfolgt.

Im ER-Modell wird von Entitytypen, bzw. von Objekttypen gesprochen. Dies könnte zum Beispiel ein Student sein, oder ein Buch. Dieser Entitytyp hat wiederum bestimmte Ausprägungen, sogenannte Entities. Im Fall des Objekttyps Student wäre das nun ein bestimmter Student. Zusätzlich wird jeder dieser **Entitytypen** durch eine Menge von **Attributen** bzw. Eigenschaften **beschrieben**. Das Objekt Student wird somit z.B. anhand des Vornamens, des Nachnamens, einer Matrikelnummer und des eingeschriebenen Studiengang **beschrieben**. Nun kommt jedoch noch hinzu, dass **jedes** einzelne **Attribut** eines **bestimmten Entities** wiederum einen bestimmten **Attributswert** oder eine Attributsausprägung besitzt. Der Vorname des Entities Huber ist z.B. Daniel und seine Matrikelnummer ist 1234567.

Dazu kommt noch, das wir neben den Objektbeschreibungen im ER-Modell auch die **Beziehungen**, die sogenannten **Relationen** zwischen den Objekttypen, beschrieben werden. Beispiel wäre hier wiederum der Student, der ein Buch aus der Bibliothek entleiht. Auch hier kann selbstverständlich die Beziehung „entleiht“ wiederum Attribute besitzen die den Vorgang näher beschreiben, z.B. das Datum der Ausleihe und Titel des Buches.

Damit die abstrakten Objekte und Beziehungen noch deutlicher werden, wird dieser Modellansatz durch ein **ER-Diagramm** unterstützt. Mittlerweile gibt es in der Literatur die unterschiedlichsten Darstellungsansätze solcher ER-Diagramme. Weit verbreitet ist, das **Objekttypen** durch **Rechtecke**, **Beziehungstypen** durch **Rauten** und deren **Attribute** durch **Ovale** dargestellt werden. Attribute sind durch **Linien** mit den Objekttypen und den Beziehungstypen verbunden, während die Objekt- und Beziehungstypen selbst wiederum durch **Linien** miteinander verbunden sind. Beziehungen können eine unterschiedliche Komplexität besitzen. Darunter versteht man, wie häufig eine Ausprägung eines Objekttyps mit einer Entität des in Beziehung stehenden anderen Objekttyps stehen kann.

Beispiel: Sehen wir uns z.B. eine typische Abteilung in einem Unternehmens an. So eine Abteilung hat im allgemeinen genau einen Leiter, m.a.W. **ein** Abteilungsleiter für **jede** Abteilung, das wäre eine klassische **1:1** Beziehung. Eine typische Abteilung besteht wiederum aus mehreren Mitarbeitern. Nun spricht man von einer **1:n** Beziehung, denn **eine** Abteilung sind **mindestens ein** *oder* **mehrere** Mitarbeiter zugeordnet. Eine dritte Überlegung wäre nun, wenn diese Mitarbeiter an verschiedenen Projekten arbeiten. Dies ergäbe dann eine **n:m** Beziehung, d.h. ein **Mitarbeiter** ist z.B. **an verschiedenen Projekten** beteiligt oder aus der Sicht der Projekte wird ein **Projekt** von **mehreren Mitarbeitern** betreut.

Bei verzweigten und komplizierten Anwendungen können die ER-Diagramme und die grafische Aufbereitung der Entity-

Relationship-Modelle **ganze Räume tapezieren**. Zur Modellierung stehen daher unterschiedliche Softwareprogramme zur Verfügung. Diese helfen bei der Entwicklung solcher Modelle. Je nach gewünschten Datenbankmodell, hierarchisch, Netzwerk oder relational wird jetzt das logische Datenbankmodell erstellt. (Zitat Ende)"

2.5 Design Relationaler Datenbanken (oder: vom ERD zur Tabelle)

"(Zitat) **Hierarchische** und **netzwerkorientierte Datenbankmodelle** können wir uns in Form einer Baumstruktur vorstellen. Stellen Sie sich den Baum auf den Kopf gestellt vor, dann können Sie von der Wurzel ausgehend jedes Blatt des Baumes über einen Ast erreichen. Der Unterschied liegt darin, dass bei dem hierarchischen Datenbankmodell der Baum nur einen sogenannten Wurzelknoten hat, von dem aus jedes Blatt des Baumes erreicht werden kann. Das Netzwerkmodell hingegen kann zwei oder mehrere solcher Wurzelknoten haben.

Das am häufigsten verwendete Datenbankmodell ist das von Codd in den 70´er Jahren entwickelte **relationale Datenbankmodell** das die hierarchischen und netzwerkorientierten Datenbankmodelle nahezu verdrängt hat und das nun ihrerseits vom objektorientierten Ansatz sukzessiv abgelöst wird. Das Relationenmodell, das die Datenbankentwicklung der letzten 30 Jahre stark beeinflusst hat, ist mit Abstand das dominierende Datenbankmodell. Codd hat die **im Entity-Relationship-Modell** beschriebenen **Objekte** und **Beziehungen** in seinem relationalen Datenbankmodell **in Form von Tabellen** umgesetzt.

Ansehen kann man sich so eine Tabelle z.B. in einer Accessdatenbank. An diesem Beispiel kann man auch leicht sehen, das die vorhin beschriebenen Objekttypen nun in Form einer Tabelle wieder auftauchen. Die einzelnen Spalten solch einer Tabelle benennen die Attribute bzw. Eigenschaften der Objekttypen. Die Zeilen einer Tabelle, die sogenannte Tupel, repräsentieren die Entities, m.a.W. eine konkrete Ausprägung des Objekttyps.

Ein Teilgebiet der Datenbanktheorie beschäftigt sich mit den sogenannten Normalformen. Die Theorie spricht z.B. in der 5. Normalform von trivalen Verbundabhängigkeiten. Wir sollten uns hier erstmal merken, dass Normalformen helfen Redundanzen und Anomalien in Tabellen zu beseitigen. Alle Normalformen sind von großer theoretischer Bedeutung. In der Praxis beschränkt man sich jedoch auf die **ersten drei**. Man hat erkannt, dass eine zu starke Normalisierung zu sehr vielen kleinen Tabellen führt. Eine Konsequenz daraus ist ein schlechtes Laufzeitverhalten der Datenbankanwendung, was sich z.B. in langen Antwortzeiten bei Datenbankabfragen wiederspiegelt. (Zitat Ende)"

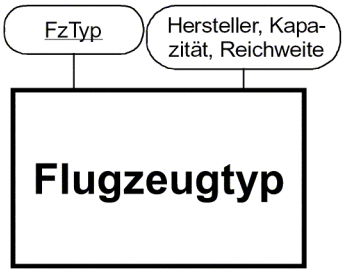
"(Zitat)

2.5.1 Umsetzung ER-Modell in Relationenschema

Die Umsetzung eines ER-Modells in ein Relationenschema erfolgt in zwei Schritten. Zunächst werden die Entity-Typen umgesetzt, dann folgen die Realtionship-Typen.

2.5.1.1. Behandlung der Entity-Typen

Aus jedem Entity-Typ wird eine Tabelle (Relation). Die Entity-Attribute werden zu den Spalten der Tabelle.
Beispiel:



wird zu:

Flugzeugtyp	FzTyp	Hersteller	Kapazität	Reichweite
	747	Boeing	340	5200
	A300	Airbus	300	4800
	...	...	...	...

Das bzw. die Identifikationsschlüssel-Attribut(e) werden in der Regel links stehend und unterstrichen notiert.

2.5.1. 2. Behandlung der Relationship-Typen

Die Umsetzung erfolgt je nach Kardinalität und Grad der Beziehung. Unterschieden wird bei der Kardinalität zwischen hierarchischen (1:1, 1:n) und netzwerkartigen Beziehungen (n:m); bei dem Grad der Beziehung zwischen Beziehungen zwischen zwei oder mehr Entity-Typen.

2.5.1.2.1. Umsetzung von Beziehungen zwischen zwei Entity-Typen

Bei hierarchischen Beziehungen wird die Relation, deren Tupel mit der einfachen Kardinalität in die Beziehung eingeht, als übergeordnete Relation bezeichnet, die andere als untergeordnete Relation. So ist z.B. bei einer Abteilung, zu der mehrere Mitarbeiter gehören, die Abteilungsrelation die übergeordnete. Bei einer 1:1-Beziehung sind für beide Relationen beide Rollen möglich.

Weiterhin wird der Begriff des **Fremdschlüssels** eingeführt:

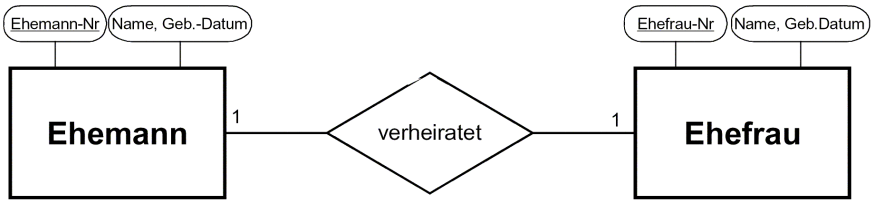
> Ein Fremdschlüssel in einer untergeordneten Relation R2 ist ein Attribut (oder eine Attributkombination), welche auch in einer übergeordneten Relation R1 vorkommt und dort Identifikationsschlüssel ist.
> Als Werte dürfen im Fremdschlüssel nur diejenigen Werte auftreten, die in der übergeordneten Relation R1 aktuelle vorhandene Tupel identifizieren (dynamischer Wertebereich).

Fremdschlüssel werden im Relationenschema durch gestrichelte Unterstreichung gekennzeichnet.

2.5.1.2.1.1. Umsetzung von 1:1-Beziehungen

Die Abbildung erfolgt durch Aufnahme des Identifikationsschlüssels der einen Relation als Fremdschlüssel in die andere Relation.

Beispiel:



Wird abgebildet als:

Ehemann	Ehemann-Nr	Name	Geb-Datum
	1	Klaus Markert	10.02.63
	2	Jürgen Schilke	11.07.58
...	...	...	

Ehefrau	Ehefrau-Nr	Ehemann-Nr	Name	Geb-Datum
	1	1	Sabine Markert	23.06.63
	2	13	Heike Sommer	22.12.67
...	...	...	...	

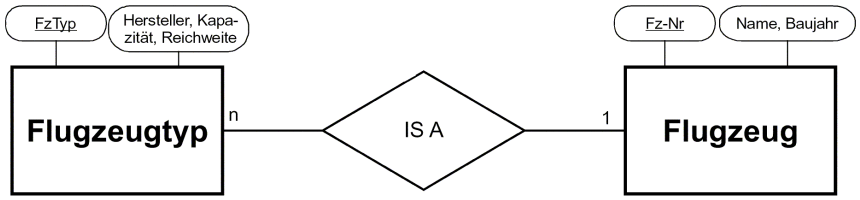
Umgekehrt ist auch eine Einbettung von Ehefrau-Nr in die Relation Ehemann möglich.

Prinzipiell sind 1:1-Beziehungen in ER-Modellen kritisch zu betrachten. Insbesondere ist zu prüfen, ob sie nicht zu einer Relation vereinigt werden können. Dies ist z.B. dann möglich, wenn eines der beiden Enitity-Typen einer 1:1-Beziehung keine weitere Beziehung hat. Dann können dessen Attribute als Spalten in die andere Relation eingeführt werden.

2.5.1.2.1.2. Umsetzung von 1:n-Beziehungen

Die Umsetzung erfolgt wiederum durch Aufnahme eines Fremdschlüssels. Allerdings ist bei 1:n-Beziehungen die Richtung vorgegeben: der Identifikationsschlüssel der übergeordneten Relation (bei der im ERM das “n” steht) wird als Fremdschlüssel in die untergeordnete Relation (bei der im ERM die “1” steht) aufgenommen.

Beispiel:



Wird abgebildet als:

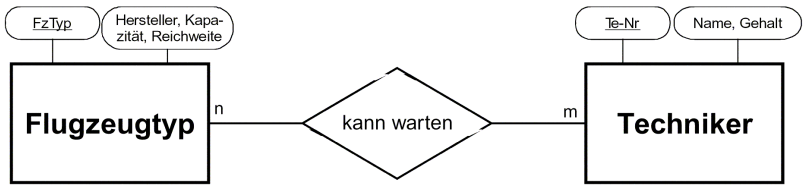
Flugzeugtyp	FzTyp	Hersteller	Kapazität	Reichweite
	747	Boeing	340	5200
	A300	Airbus	300	4800
...	...	...	...	

Flugzeug	Fz-Nr	FzTyp	Name	Baujahr
	1	747	Heidelberg	1988
	2	747	Spirit of L.A.	1996
...	...	...	...	

2.5.1.2.1.3. Umsetzung von n:m-Beziehungen

Für eine n:m-Beziehung wird eine zusätzliche Relation eingeführt, welche die Identifikationsschlüssel als Spalten enthält. Alle diese Spalten haben einen dynamischen Wertebereich.

Beispiel:



Wird abgebildet als:

Flugzeugtyp	FzTyp	Hersteller	Kapazität	Reichweite
	747	Boeing	340	5200
	A300	Airbus	300	4800
...	...	...	...	

Techniker	Te-Nr	Name	Gehalt
	1	Johnson	60.000
	2	Myers	72.000
...	...	...	

kann_warten	FzTyp	Te-Nr
	747	1
	747	2
	A300	2
	...	...

2.5.1.2.2. Umsetzung von Beziehungen zwischen mehr als zwei Entity-Typen

Sind an der Beziehung mehr als zwei Entity-Typen beteiligt, so ist die Aufnahme von Fremdschlüsseln in die Relationen nicht mehr möglich. Es wird in eigene Relation ähnlich wie in Abschnitt 2.1.3 beschrieben angelegt..

2.5.1.2.3. Umsetzung von Attributen von Relationship-Typen

Wird für den Relationship-Typ eine eigene Relation angelegt, werden die Attribute zu Spalten dieser Relation. Wurde bei einer hierarchischen Beziehung der Relationship-Typ durch Einfügen eines Fremdschlüssels abgebildet, können die Attribute in die untergeordnete Relation eingegliedert werden. Dem Umsetzungprozess schließt sich der Normalisierungsprozess für das Relationenschema an.

2.6 Die Datenbanksprache SQL

Eine Datenbanksprache ist eine Sprachen zur Definition, Pflege und Auswertung von Datenbanken auf logischer Ebene. Standard bei relationalen Datenbankverwaltungssystemen ist SQL (Structured Query Language), von IBM entwickelt; seit 1989 international standardisiert. SQL ist eine deskriptive Sprache (beschreibt das WAS, nicht das WIE) Und - **ganz wichtig** - GroSS- unD klEInSCHreiBUNG ist bei SQL völlig egal!

Mit SQL kann ich also dem Rechner, bzw. dem Datenbankverwaltungsprogram, sagen, WAS zu tun ist. Da der Rechner jedoch eine dumme Maschine ist, bzw. mein Program - im Vgl. zum Menschen - ein geistig minderbemitteltes "Etwas" ist; muss ich meine Befehle in einer "Sprache" schreiben, die ein wenig nach ich-rede-mit-einem-Deppen klingt. Trotzdem ist SQL eine Sprache, die für einen Menschen (ohne SQL Kenntnisse) relativ gut lesbar, d.h. verständlich ist. Nachfolgend habe ich - quasi als Einführungsbeispiel - drei SQL-Befehle dargestellt, um ein Gefühl für die Logik von SQL zu vermitteln:

SELECT (Datenabfrage)

Syntax		
SELECT *Spalte* **FROM** *Tabelle* **WHERE** *Bedingung* **ORDER BY** *Spalte* **DESC / ASC**		
Beispiel		
SELECT * FROM *autopreise* **WHERE** *preis < 20000* **ORDER BY** *preis* **ASC** dieser Befehl ließe sich übersetzen mit: **WÄHLE** *alle Einträge* **AUS DER** *Tabelle "autopreise"*, **BEI DENEN** *in der Spalte "preis" ein Wert kleiner 20000 steht*, **UND ORDNE DIESE NACH** *der Spalte "preis"* **IN AUFSTEIGENDER REIHENFOLGE** (d.h. Zeile mit höchstem Eintrag in "preis" steht ganz oben, usw.)		
Befehl	*erforderlich?*	*Erklärung*
SELECT	ja	Definiert welche Spalte(n) der Tabelle gelesen werden sollen, "*" bedeutet: alle Spalten
FROM	ja	Eine Datenbank kann mehrere Tabellen enthalten; auch wenn sie nur eune Tabelle enthält, muss trotzdem hier der Name der Tabelle angegeben werden.
WHERE	nein	Wenn angegeben, werden nur die Zeilen der Tabelle gelesen, in denen die hier angegebene Bedingung wahr ist
ORDER BY	nein	Normalerweise werden die Daten in der Reihenfolge ausgegeben, wie sie in der Datenbank stehen; wenn ORDER BY angegeben ist, kann man die Zeilen nach den Werten einer bestimmten Spalte Ordnen
DESC / ASC	nein	Nur in Verbindung mit ORDER BY: absteigende oder aufsteigende Ordnung

UPDATE (Daten ändern)

Syntax		
UPDATE *Tabelle* **SET** *Feldname='Feldwert'* **WHERE** *Bedingung*		
Beispiel		
UPDATE *autopreise* **SET** *preis = '1000'* **WHERE** *AutoID = 52* dieser Befehl ließe sich übersetzen mit: **ÄNDERUNG** *in Tabelle "autopreise"*; **SETZE DEN WERT** *der Spalte "preis" auf "1000"* **WO** *die Spalte "AutoID" den Wert "52" hat*		
Befehl	*erforderlich?*	*Erklärung*
UPDATE	ja	Definiert in welcher Tabelle ein Feld / Felder geändert werden soll(en)
SET	ja	Der Wert in der Spalte mit dem Namen "Feldname" wird in den neuen Feldwert geändert
WHERE	nein	Wenn angegeben, erfolgt die Änderung nur in den Zeilen, in denen die Bedingung wahr ist

INSERT INTO (Neue Daten in Tabelle einfügen)

Syntax		
INSERT INTO *Tabelle (Spalte1, Spalte2, ...)* **VALUES** *('Wert1', 'Wert2', ...)*		
Beispiel		
INSERT INTO *autopreise (marke, baujahr, preis)* **VALUES** *('VW', '1998', '10000')* dieser Befehl ließe sich übersetzen mit: **FÜGE NEUE ZEILE IN** *Tabelle "autopreise": (in Spalte "marke", "baujahr", "preis")* **SCHREIBE DIE WERTE** *("VW", "1998", "10000")*		
Befehl	*erforderlich?*	*Erklärung*
INSERT INTO	ja	Definiert an welche Tabelle die VALUES angehängt werden sollen
(Spalte1, Spalte2, ...)	nein	Definiert, für welche Spalten die nachstehenden VALUES gedacht sind. Wenn diese Klammer nicht angegeben ist, muss unter VALUES ein Wert für jede Spalte angegeben sein
VALUES	ja	Definiert (in Klammern) die Werte, die in die neue Zeile geschrieben werden sollen

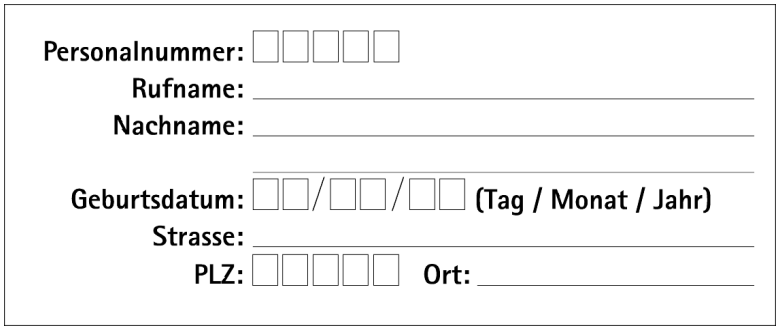
Die SQL-Befehle werden systematisch in zwei Kategorien eingeteilt. Unterschieden wird in **DDL (Data Description Language)** und **DML (Data Manipulation Language)**. [Hinweis: In mancher Literatur wird dem "select"-Befehl aufgrund seiner besonderen Bedeutung eine eigene Kategorie zugestanden.] Es gibt noch eine Reihe weiterer Befehle, die wir nicht behandeln, für deren systematische Einordnung noch weitere Kategorien zur verfügung stehen. (...) Dies sei hier nur der Vollständigkeit halber erwähnt. Aber zurück zu den Klausurrelevanten DDL- und DML-Befehlen:

2.6.1 SQL im Einzelnen: Die DDL-Befehle

DDL steht für "Data Description Language" - es geht also irgendwie um Befehle, die "Daten definieren", eigentlich geht es um Befehle die Tabellen (inkl. der Tabellenstruktur) definieren. DDL-Befehle haben nichts mit dem Einlesen, Auslesen, Löschen oder Updaten von Daten zu tun - insofern ist die Bezeichnung "DDL" irreführend. Für den Hausgebrauch merkt man sich am besten folgende (fränkische)"Übersetzung" für DDL: "Dabellen Definier Language".

2.6.1.1 Eine Tabelle erstellen: "create table"

Was mache ich jetzt mit DDL? Mit DDL lege ich z.B. (**leere**) Tabellen an: Ich **definiere** den Namen der Tabelle, welche Spalten sie hat und was ich in die Tabelle reinschreiben kann. (Beliebige Zeichen oder nur Ziffern oder ein Datum oder ...) Nemen wir wieder den Vergleich mit dem Karteikarten-Archiv: Wenn ich - z.B. für eine Personalabteilung - eine solche "Karteikartnen-Datenbank" anlege, werde ich mir zuerst mal darüber Gedanken machen, wie die Schubladen beschriftet werden sollen (= Namen meiner Tabellen definieren). Dann werde ich Vordrucke der Karteikarten machen lassen, die zum Beispiel. so aussehen:



Hier tue ich im Prinzip genau das gleiche, wie bei einer Tabellendefinition in SQL. Ich habe definiert, welche Attribute in der Karte auszufüllen sind (= Spalten in Tabeele), also WAS ich eintragen kann, und und WIE ich das eintragen kann (soll). Bei "Nachname" habe ich genügend Platz (fast) jeden noch so langen Namen einzutragen, in der entspr. Zeile meiner Datenbanktabelle wären also "bis zu 20 Buchstaben" zugelassen. Anders beim Attribut "Geburtsdatum": Hier ist mir - sinnvollerweise - vorgeschrieben z.B. "01.02.76" zu schreiben, anstelle von "1. Februar 1976" oder ähnliches. In SQL würde ich die entsprechende Tabelle mit folgendem DDL-Befehl anlegen: **create table** personal **(**Personalnummer **integer not-null unique primary key,** Rufname **char(15) not-null,** Nachname **char(20) not-null,** Geburtsdatum **datetime,** Strasse **char(30) not-null**, PLZ **integer not-null,** Ort **char(20) not-null)** Das Ergebnis wäre folgende Tabelle:

personal

Personalnummer	Rufname	Nachname	Geburtsdatum	Strasse	PLZ	Ort
01325	Daniel	Weiland	15.02.58	Merseburger Str 28	10923	Berlin

***Anmerkung**:* Die Personalnummer ist unterstrichen, um deutlisch zu machen, dass es sich hier um den Primärschlüssel handelt. Herrn Daniel Weiland habe ich **als Beispiel** eingetragen. Die mit dem **create table** -Befehl erstellte Tabelle wäre selbstverständlich zunächst **leer**.

Im Folgenden eine **Übersicht** zum **create table** mit allen "definier"-Befehlen, die ich im Skript gefunden, oder die in der Übung verwendet wurden.

Syntax: **create table** |tabellenname| **(** |spaltenname| |**ZeichenArt**| |**ZeichenRestriktion**|, ... **)**

Übersicht: Befehle zur Definition der Art der Zeichen (**ZeichenArt**):

char(n)	Eine Zeichenfolge der Länge n
integer	Eine ganze Zahl (z.B. bei INFORMIX im Wertebereich von -2.147.483.647 bis +2.147.483.647; Wertebereich hängt also vom DBMS ab, aber niemals Kommastellen, weil ja ganze Zahl!
long int	*??ichweisnicht??*
real(n)	Gleitkommazahl, ist (n) angegeben hat die Zahl n Stellen
bit	Boolesche Werte (also "1" oder "0")
datetime	Datums- oder Zeitangabe (genaue Def. hängt vom DBMS ab, also ob 30.11.01 (Tag.Monat.Jahr) oder 11.30.2001 (Monat.Tag.Jahr) usw.)

Übersicht: Befehle zur Definition einer Restriktion der Zeichen (**ZeichenRestriktion**):

null	"Kein-Wert" ausdrücklich zugelassen, d.h. die Zelle der Spalte darf leer bleiben
not-null	"Kein-Wert" nicht zulässig, d.h. es muss ein Wert zugewiesen werden
default 0	Der default-Wert ist der Wert, der Standartmäßig eingetragen wird, wenn nichts angegeben wird. Hier ist dies z.B. die "0" (Vgl. Unterschied zu "null")
unique	(I.d.R. nur in Verbindung mit "primary key") "Einzigartig", d.h. in der ganzen Spalte darf kein Wert zweimal vorkommen.

Zwei **besondere** "Restriktionen" sind:

primary key	Definiert die Spalte als "Primärschlüsselspalte" (I.d.R. in Verbindung mit **unique** und **not-null**)
refers	Stellt quasi eine Art Verknüpfung her, definiert so gesehen einen Fremdschlüssel. Beispiel: Im Befehl "**create table** Bestellung (*[...]* , Kunde **integer refers** KundenListe.Knd_Nr)" weist das **refers** an, dass in der Spalte "Kunde" der Tabelle "Bestellung" ein Wert aus der Spalte "Knd_Nr" der Tabelle "KundenList" eingetragen werden soll. (Einem Assistenten würde ich sagen: "Bei "Bestellung" in die Spalte "Kunde" bitte die "Knd_Nr" eintragen. Die "Knd_Nr" ist der Tabelle "Kundenlist" zu entnehmen.")

Beispiel:

create table Bestellung **(**Artikel **char(10) not-null,** Menge **integer default 1,**
*"**mache Tabelle** Bestellung **(**Spalte Artikel **hat 10 Zeichen & darf nicht "kein Wert" haben,** Spalte Menge **hat ganze Zahl & wenn nichts anderes angegeben dann Wert 1,***

Kunde **integer refers** KundenListe.Knd_Nr**,** Anmerkung **char(27) null)**
*Spalte Kunde **hat ganze Zahl & einen Wert aus** KundenListe.Knd_Nr**,** Spalte Anmerkung **hat 72 Zeichen & darf "Kein Wert" haben)"*

2.6.1.2 Die Tabelle "erweitern": "alter table"

Neben dem "**create table**" gibt es noch weitere DDL-Befehle, zwei davon tauchen im Rihland-Skript auf und werden hier daher auch kurz angesprochen. Es handelt sich um "drop table" und "**alter table**". Mit dem Letzteren kann ich eine **bereits bestehende** Tabelle (Relation) um eine Spalte (Attribut) erweitern. Die Syntax ist - sofern man das "create table" verstanden hat - relativ einfach.

Syntax: **alter table** |tabellenname| **add** |spaltenname| |**ZeichenArt**| |**ZeichenRestriktion**|

Stellen wir uns vor, im Unternehmen X wird beschlossen, jedem Mitarbeiter zum Namenstag zu gratulieren. (Blödes Beispiel,

ich weiß, aber mir fiel nix besseres ein...) Jetzt muss also die arme Sekretärin mit einer Liste aller Vornamen und einem Kalender bewaffnet für jeden Vornamen den Namenstag herausfinden. Und sie muss diese Namenstage ja auch irgendwo eintragen. Dummerweise gibt es aber in der oben erstellen Tabelle keine Spalte "Namenstag". Also muss die bereits bestehende Tabelle "personal" um eine Spalte namens "Namenstag" erweitert werden. Der entsprechende SQL-Befehl lautet demnach:
alter table personal **add** Namenstag **datetime null**
Damit hat die Tabelle "personal" jetzt eine neue Spalte "Namestag", wobei der Namenstag als Datum eingetragen werden muss, und es erlaubt ist auch keinen Wert einzutragen. (Was logisch ist: nicht für jeden Namen gibt es einen Namenstag)

2.6.1.3 Eine Tabelle entfernen: "drop table"

Dieser Befehl ist wohl eindeutig der simpelste aller SQL-Befehle: Will ich die Tabelle "personal löschen, lautet mein Befehl:
drop table personal
Und weg isse.

2.6.2 SQL im Einzelnen: Die DML - Befehle

Gut, wir haben nun unsere **leeren Tabellen** angelegt, nur Tabellen ohne "Inhalt" sind irgendwie recht nutzlos. Als nächstes geht es daher darum, diese Tabellen

a) mit Daten zu "füllen",
b) Daten zu ändern,
b) ggf. Daten zu löschen (Daten, nicht die ganze Tabelle) und
c) bestimmte Daten abzufragen.

All dies tut man mit den sog. DML-Befehlen.

2.6.2.1 Daten einlesen / Tabellen "füllen": "insert into"

Der Befehl ist im Grunde sehr einfach, wenn man sich folgendes Überlegt: Dem DBMS muss gesagt werden, in **welche Tabelle** was für **Werte** in welche **Spalte** geschrieben werden sollen. Das ist auch nicht anders, als wenn ich zu meinem Assistenetn sage: "Bitte trage in die "personal"-Tabelle den neuen Mitarbeiter ein. Seine Personalnummer soll 01325 sein, sein Rufname ist Daniel, sein Nachname Weiland,...(usw.)"

Der enstprechende SQL-Befehl würde dann lauten: **insert into** personal **(**Personalnummer**,** Rufname**,** Nachname**,** Geburtsdatum**,** Strasse, PLZ**,** Ort**) values (**01325, 'Daniel', 'Weiland', '15.02.58', 'Merseburger Str 28', 10923, 'Berlin'**)**
"*Ergebnis:*"
personal

Personalnummer	Rufname	Nachname	Geburtsdatum	Strasse	PLZ	Ort
01325	Daniel	Weiland	15.02.58	Merseburger Str 28	10923	Berlin

Nehmen wir an, dass ich jetzt noch eine neue Mitarbeiterin habe, deren Geburtsdatum ich aber nicht kenne. Was nun? Überhaupt kein Problem, den so wie ich die "Eigenschaften" der Spalten der Tabelle definiert habe (vgl. 2.6.1.1) darf die Spalte "Geburtsdatum" leer bleiben. Ich muss dann nur in der ersten Klammer des Befehls den Spaltennamen einfach weglassen. Der Befehl wäre dann z.B.: **insert into** personal **(**Personalnummer**,** Rufname**,** Nachname**,** Strasse, PLZ**,** Ort**) values (**01025, 'Lisa', 'Maier', Am 'Wildwechel 16', 10924, 'Berlin'**)**

Damit hätte ich die Tabelle "personal" um eine weitere Zeile erweitert:

Personalnummer	Rufname	Nachname	Geburtsdatum	Strasse	PLZ	Ort
01325	Daniel	Weiland	15.02.58	Merseburger Str 28	10923	Berlin
01025	Lisa	Maier	NULL	Am Wildwechsel 16	10924	Berlin

Syntax: **insert into** |tabellenname| **(** |namespalte1|, |namespalte2|, ... **) values (** |Wert1|, |Wert2|, ...**)**
Anmerkung: Wenn ich nun z.B. den Namen der Spalte 2 weglasse, kann ich auch bei "values" den Wert 2 weglassen.

2.6.2.2 Daten änder: "update"

Stellen wir uns vor, Frau Maier ist umgezogen in die "Hübnerstraße 24". Dann muss ich ja mindestens den Eintrag in der Spalte "Strasse" ändern (ich unterstelle hier der Einfachheit halber, dass die PLZ & Ort derselbe geblieben sind). Zum Ändern von Daten steht mit der **update**-Befehl zur Verfügung:

Syntax: **update** |tabellenname| **set (|**spalteÄnder| **= |**wertneu| **) where |**erkennspalte| **= |**wert|

Ich "sage" quasi zum Rechner: "**Verändere** die Tabelle |tabellenname| **bitte verändere (**in der Spalte |spalteÄnder| den Wert auf |wertneu|**) aber nur dann wenn Du in der gleichen Zeile** in der Spalte |erkennspalte| den Wert |wert| findest"

In unserem Beispiel wäre das: **update** personal **set (** Strasse **=** 'Hübnerstraße 24' **) where** Personalnummer **=** 01025

Würde ich hier **kein** " **where** Personalnummer **=** 01025 " angeben, dann würde in **jeder** Zeile "Hübnerstraße" reingeschrieben werden. Und: Es war kein Zufall, dass ich mir hier "Personalnummer" rausgesucht habe: schließlich ist das unser Primär-schlüssel in der Tabelle, und daher nur EINMAL vorhanden. Hätte ich z.B. aus Boshaftigkeit allen "Maier's" in der Tabelle den Rufnamen "Seppl" verpassen wollen, hätte ich log.weise als |erkennspalte| Nachname (mit = 'Maier') gewählt...

2.6.2.3 Daten (Zeilen) löschen: "delete from"

Jetzt kann es natürlich sein, dass ich Frau Maier gefeuert habe (weil die ständig umgezogen ist, und so "rastlose" Mitarbeiter kann ich nicht gebrauchen). Also muss **die Zeile** der personal-Tabelle, **in der** die Personalnummer **von Frau Maier** auftaucht (und die log.weise daher "ihre" Zeile ist) gelöscht werden. (Aber nur die eine Zeile, nicht die ganze Tabelle, daher wäre "drop table" hier keine gute Idee...) Das mach ich mit "delete from", und der Befehl ist sehr einfach:

Syntax: **delete from** |tabellenname| **where |**erkennspalte| **= |**wert|
"***Lösche genau die Zeile(n) aus*** |*tabellenname*| ***in denen Du |****erkennspalte*| **= |***wert*| ***findest***"

Im Beispiel wäre das: **delete from** personal **where** personalnummer **=** 01025 Natürlich könnte ich hier auch mit dem Befehl " **delete from** personal **where** nachname **=** Maier " einfach alle Maier's löschen, wenn ich so wollen würde...

2.6.2.4 Daten abfragen: Der "select" Befehl

Der Chef sagt zu seinem Assistenten: "Such mir mal aus dem Telefonbuch die Telefonnummer vom Wirtschafzsreferenten raus" Ich schreibe den SQL-Befehl: "**select** telnummer **from** telbuch **where** Beruf = 'Wirtschaftsreferent' " In beiden Fällen wurde mitgeteilt, **was**, **worin** unter welcher **Bedingung** zu suchen war. Wer das kapiert hat, hat das Grundprinzip vom select-Befehl verstanden.

Das heißt die Syntax ist eigentlich sehr logisch, und lässt sich wie folgt darstellen:
Syntax: **select** |spaltenname1|, |Spaltenname2|, ... **from** |tabellenname| **where |**erkennspalte| **= |**wert|

Zwischen **select** und **from** schreibe ich - durch Kommas getrennt - all die Spalten rein, deren Werte mich interessieren., will ich alle Spalten, so schreibe ich **select (*)**. Nach dem **from** kommt logischerweise der Name der Tabelle (damit meine DB auch weiß wo sie suchen soll); und mit der **where**-Klausel definiere ich wieder eine Bedingung, die determiniert welche Zeilen interessant sind. Das Ergenis sind dann logischerweise alle Spaltenwerte aus den Zeilen, in denen meine **where**-Bedingung zugetroffen hat.

2.6.2.5 Tabellen logisch verbinden - die "JOIN"-Befehle

Mit den bisher kennengelernten Befehlen lässt sich schon eine Menge anstellen. Blöd wird es nur, wenn ich auf zwei Tabellen zugreifen müsste, um meine Informationen zubekommen; und dies wird eher die Regel sein. Beispiel: Die zentrale Personalverwaltung eines Unternehmens hat in der Tabelle "mitarbeiter" jew. den Namen, Adresse, Telefonummern, Stellung und die Mitarbeiternummer (m_nummer) aller Beschäftigten - vom Manager bis zum Azubi - gespeichert. Die Abteilung Lohnbuchhaltung hat in einer Tabelle "loehne" alle 'Lohndaten' - also Steuernummer, Was schon Ausbezahlt, Kontoverbindungen etc. Um jetzt aber dem Angestellten X seinen Lohn auf das Konto zu überweisen und ihm einen "Lohn- und Gehaltsrechnung" zukommen zu lassen benötigt die Lohnbuchhaltung Informationen aus **beiden** Tabellen, "mitarbeiter" und "loehne".

Jetzt wäre es natürlich recht praktisch, wenn für die Zeit, in der die Lohnbuchhaltung die Informationen abfragt ein Tabelle existieren würde, in der alle Daten vorhanden sind, und die Attributwerte der Entities auch richtig zusammengesetzt, also die richtigen Zeilen miteinander verbunden wären. Man könnte jetzt natürlich eine solche Tabelle erstellen und speichern. Aber dann hätte man wieder Probleme, weil es **zwei** Tabellen gäbe, in der z.B. die Kontonummer von Herrn X auftaucht (in der "loehne" und in der neuen Tabelle). Und was ist jetzt, wenn Herr X die Bank wechselt? Wer stellt sicher, dass die Änderungen in beiden Tabellen vorgenommen werden? usw... Letztendlich würde man hierbei zu einem haufen neu generierter Tabellen kommen und schließlich bei einer Art dezentralen Datenspeicherung im Sinne von Punkt 2.2 kommen. **Besser** wäre es daher, wenn die neue Tabelle garnicht echt existiert, sondern nur im Moment der Abfrage - quasi als logisches Konstrukt. Und genau so funktionieren select-Abfragen in Verbindung mit dem **join**-Befehl! Hierbei hat uns Herr Ruhland zwei Arten präsentiert: Der **inner join** und der **left outer join** Befehl. Fangen wir mit dem **inner join** an:

select Name, Ort, loehne.Kont_Nr **from** mitarbeiter **inner join** loehne **on** mitarbeiter.PersNr = loehne.PersNr **where** |Bedingung|

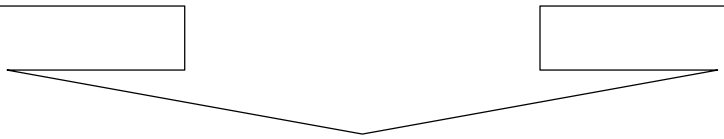
Ich habe bei diesem Befehl einen Teil so "häßlich" umrahmt, das hat einen Sinn: Mann muss sich die **join** befehle nämlich so vorstellen, als würde zunächst eine (temporäre) Tabelle erstellt werden die der Verknüpfung im "Kasten" entspricht und die nach **select** folgenden Zeilen enthält:

mitarbeiter

Name	**Ort**	**PersNr**
Müller	Berlin	001
Maier	Berlin	002
Schmitdt	Muenchen	003

loehne

PersNr	**Kont_Nr**	**Steuer_Nr**
001	456879	25268
003	458963	58694
008	856931	87745



Name	**Ort**	**Mitarbeiter.persNr**	**Loehne.PersNr**	**Kont_Nr**
Müller	Berlin	001	001	456879
Schmitdt	Muenchen	003	003	458963

Alle Zeilen, die nicht mit in die "neue" Tabelle übernommen werden habe ich in den Ursprungs-Tabellen Grau gemacht. Warum werden diese Zeilen nicht mit übernommen? Weil die Verknüpfung (= das was nach dem **on** steht) hier nicht gilt: Es gibt weder in loehne eine Zeile mit der PersNr = 002 noch in der mitarbeiter eine mit PersNr = 008 (warum ist jetzt egal).

In dieser "temporären" Tabelle wird dann die **where**-Bedingung ausgeführt. Falls es keine Bedignung mehr gibt, wäre die obige Tabelle sogar die "AusgabeTabelle". Falls da noch ein **where** Adresse = 'Muenschen' -Teil dranhängen würde, wäre das ergebnis halt nur die Zeile mit Herrn Müller...

Was mach ich jetzt aber, wenn ich z.B. wissen will, **welche** Mitarbeiter **nicht** in der "loehne"-Tabelle auftauchen (weil die dann vielleicht Manager sind, und ich will die Namen aller meiner Manager oder so...). Mit dem "normalen" **inner join** geht das **nicht**: Der zeichnet sich ja gerade dadurch aus, dass er **nur die** Zeilen zusammenlegt, **wo** die **on**-Verknüpfung erfüllt ist... Und mit etwas anderem als mit '**on** mitarbeiter.PersNr = loehne.PersNr' kann ich die Tabellen garnicht verknüpfen, weil das ja die einzigen Spalten sind, die in beiden Tabellen vorkommen (sind übrigens auch Primär- bzw. Fremdschlüssel)

Ich kann mir aber folgendes Überlegen: Wenn ich die Datenbank dazu bringen könnte, die "temporäre"-Tabelle dergestalt zu generieren, dass auf jeden Fall **alle** Zeilen der "mitarbeiter"-Tabelle mit übernommen werden, dann wäre die Datenbank gewissermaßen gezwungen in die Felder, deren Werte eigentlich aus der "loehne" kommen sollten (wo es aber keine Felder gibt, weil die Zeile "fehlt"), den Wert "garnichts" - also NULL reinzuschreiben. Und dann könnte ich in dieser "temporären" Tabelle nach der Bedingung "loehne.PersNr = NULL" suchen lassen und bekäme das Ergenis, was ich haben will. Genau das ist der Trick vom **left outer join** - Befehl:

select Name, Ort, loehne.Kont_Nr **from** mitarbeiter **left outer join** loehne **on** mitarbeiter.PersNr = loehne.PersNr **where** loehne.PersNr = NULL

mitarbeiter

Name	**Ort**	**PersNr**
Müller	Berlin	001
Maier	Berlin	002
Schmitdt	Muenchen	003

loehne

PersNr	**Kont_Nr**	**Steuer_Nr**
001	456879	25268
003	458963	58694
008	856931	87745

Name	**Ort**	**Mitarbeiter.persNr**	**Loehne.PersNr**	**Kont_Nr**
Müller	Berlin	001	001	456879
Maier	Berlin	002	NULL	NULL
Schmitdt	Muenchen	003	003	458963

Und das wär mein Ergebnis:

Name	**Ort**	**Mitarbeiter.persNr**	**Loehne.PersNr**	**Kont_Nr**
Maier	Berlin	002	NULL	NULL

Ich denke die Syntax vom **inner join** und vom **left outer join** ist so offesichtlich, dass ich jetzt nicht näher darauf eingehen will. Nur eine kleine Anmerkung zum **left outer join**: Das Ding heißt ***left*** **outer join**, daher ist es naheliegend, dass die linke Tabelle vollständig (was die Zeilen betrifft) übernommen wird. Aber: "Linke Tabelle" meint natürlich nicht "die Tabelle, die links auf dem Papier hingezeichnet ist", sondern: "Linke Tabelle ist immer die Tabelle, welche *links vom* **left outer join** - Befehl steht!" Das heißt hätte ich "**... from** loehne **left outer join** mitarbeiter **on** loehne.PersNr = mitarbeiter.PersNr ..." geschrieben wäre log.weise aus der "loehne"-Tabelle alle Zeilen mit übernommen worden.

2.6.2.6 Aggregatfunktion und Gruppenbildung: " count" / "group by"

"(Zitat)**Aggregatfunktionen**
In SQL gibt es fünf Aggregatfunktionen: **COUNT,** MIN**,** MAX**, SUM**, AVG, sowie für statistische Auswertungen TOP (höchste *n* Werte), STDDEV (Standardabweichung) und VAR (Varianz).[Anmerkung: die **FETTEN** sollte man kennen]

COUNT (*) zählt die Anzahl der Tupel einer Relation (bzw. des kartesischen Produkts).

Beispiel: *Gib die Anzahl der Artikel aus*

```
SELECT COUNT(*)
FROM Artikel;
```

COUNT (Attributename) zählt die Anzahl der Felder des Attributes, die nicht NULL sind.

Beispiel: *Gib die Anzahl der von NULL verschiedenen Kundennamen aus*

```
SELECT COUNT(Name)
FROM Kunde;
```

Leider verfügt Access-SQL nicht über die Möglichkeit, die Anzahl der verschiedenen Ausprägungen eines Attributs zu zählen, was in Standard-SQL mittels COUNT (DISTINCT Attributename) möglich ist. [?]

MIN bzw. MAX bestimmt den kleinsten bzw. größten Wert eines Attributs.

Beispiel: *Gib den minimalen und den maximalen Preis eines Artikels aus*

```
SELECT Min(Preis) AS Minimalpreis, Max(Preis) AS Maximalpreis
FROM Artikel;
```

SUM bzw. AVG bestimmt Summe bzw. Durchschnitt von numerischen Attributen. Nullwerte bleiben außer Betracht. Enthält eine Attributspalte nur Nullwerte, so ist das Ergebnis NULL.

Beispiel: *Gib das durchschnittliche Gewicht eines Artikels aus*

```
SELECT AVG(Gewicht) AS Durchschnittsgewicht
FROM Artikel;
```

Beispiel: *Gib die Summe des Wertes aller aktuellen Bestellungen aus*

```
SELECT SUM(Bestellung.Menge*Artikel.Preis) AS Gesamtwert
FROM Bestellung, Artikel
WHERE Bestellung.Anr=Artikel.Nr;
```

Gruppenbildung

Für manche Zwecke ist es nützlich, eine Relation horizontal in mehrere Teilrelationen zu zerlegen. Dies kann mit einer GROUP BY-Klausel geschehen:

```
SELECT ... FROM ... WHERE ... GROUP BY attributnamen
```

Für jede Wertkombination der Attributliste in der GROUP BY-Klausel wird eine Teilrelation (Gruppe) erzeugt. Eine GROUP BY-Klausel hat den Effekt, daß alle Aggregatfunktionen in der SELECT-Klausel auf jeder Gruppe einzeln operieren.

Achtung: Enthält eine Anfrage eine GROUP BY-Klausel, so dürfen in der SELECT-Klausel nur die Attribute der GROUP BY-Klausel vorkommen. Weitere Attribute sind nur als Operanden in einer Aggregatfunktion erlaubt.

Beispiel: *Gib die aktuelle bestellte Menge und die Anzahl der Bestellungen für jeden Artikel aus*

```
SELECT Name, SUM(Menge) AS Bestellte_Menge,
COUNT(*) AS Anzahl_Bestellungen
FROM Artikel, Bestellung
WHERE Artikel.Nr=Bestellung.ANr
GROUP BY Name;
```

Beispiel: *Gib den Gesamtwert und das Gesamtgewicht der Bestellungen jedes Kunden aus*

```
SELECT Kunde.Name, Adresse, SUM(Menge*Artikel.Preis) AS Gesamtwert,
```

```
SUM(Menge*Artikel.Gewicht/1000) AS Gesamtgew_in_Kilo
FROM Kunde, Artikel, Bestellung
WHERE (Kunde.Nr=KNr) AND (Artikel.Nr=ANr)
GROUP BY Kunde.Name, Adresse;
```
(Zitat Ende)"

2.6.2.7 Der "create view" Befehl

Der **create view** speichert mein Ergebnis einer ganz normalen **select**...-Anfrage als Tabelle. Beispiel: Hätte ich geollt, dass meiner vermeintlichen Manager aus der **select...left outer join**-Anfrage (Siehe Punkt ???) als neue Tabelle gespeichert wird, dann hätte ich einfach ein "**create view** |Name| **as**" dem Befehl vorangestellt:

Create view manager **as**
select Name, Ort, loehne.Kont_Nr **from** mitarbeiter **left outer join** loehne **on** mitarbeiter.PersNr = loehne.PersNr **where** loehne.PersNr = NULL

Das Ergebnis wäre die "view" Namens "manager", die ich wie eine normale Tabelle mit weiteren Befehlen ansprechen kann.

2.6.2.8 Nochma Übersicht (DDL & DML ohne Select-Familie):

DLL-Befehle

```
create table  tabellenname  (  namespalte1 | char      | primary_key  ,  ...  )
                                          | integer   | not-null
                                          | bit       | default0
                                          | datetime  | null
                                          | ...       | ...
```

```
drop table    namedertabelle
```

```
alter table  namedertabelle  add  namespalte | char      | primary_key
                                             | integer   | not-null
                                             | bit       | default0
                                             | datetime  | null
                                             | ...       | ...
```

DML-Befehle

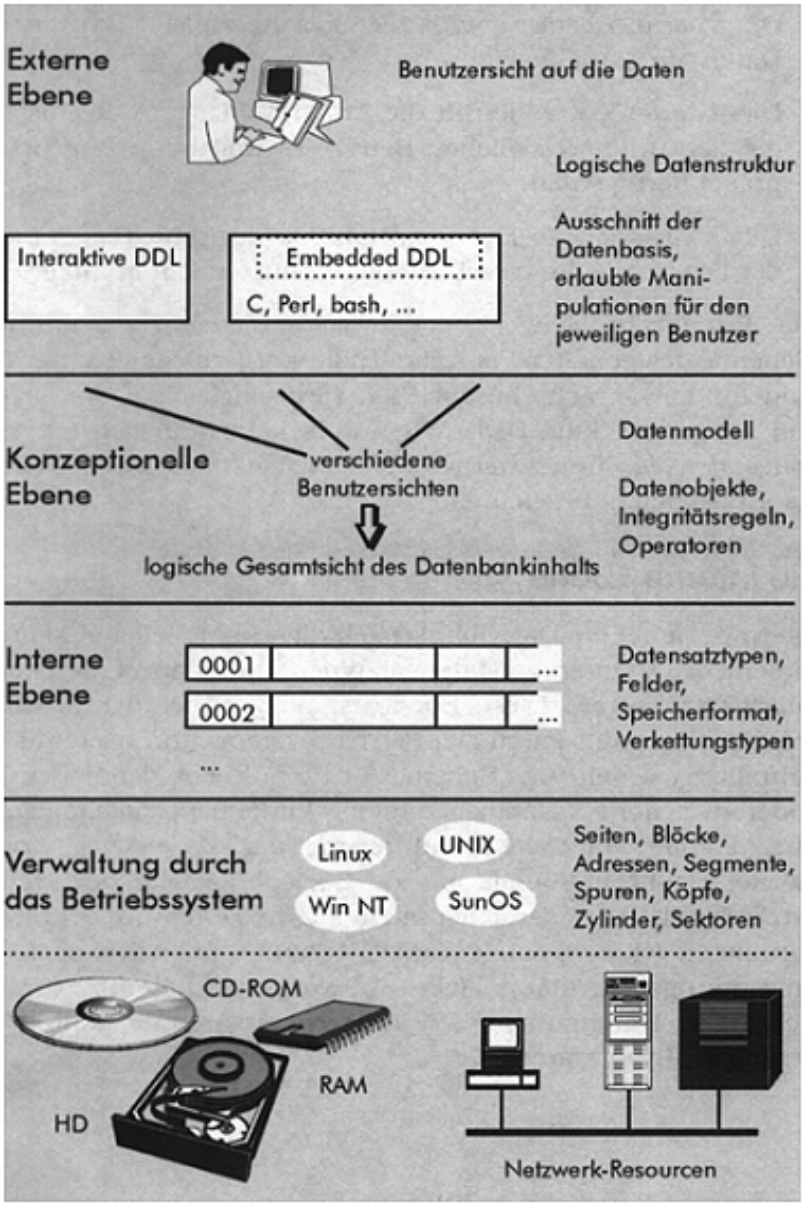
```
insert into namedertabelle  ( namederspalte1,  namederspalte2,  namederspalte3,  ...)
                    values  ( wert1,           wert2,           wert3,           ...)
```

```
delete from   namedertabelle  where  spaltenname = wert
```

```
update        namedertabelle  set  ( namespalteX = wert ) where  spaltenname = wert
```

2.7 Das Drei-Ebenen Schema

"(Zitat) **Das Dreischichtenmodell nach ANSI/SPARC:** Die Idee hinter dem ANSI/SPARC Modell ist die Datenunabhängigkeit der Daten gegenüber Veränderungen der Speicherstrukturen. Das DBMS ist eine Schnittstelle zu den Daten.



Es gibt 3 Betrachtungsebenen der Daten:

- *interne Ebene:* Speicherstruktur der Daten auf physikalischen Medien
- *konzeptionelle Ebene:* enthält die logische Gesamtsicht der Daten
- *externe Ebene:* Art in der die Daten den Benutzern präsentiert werden

Die **interne** Ebene ist direkt über der Resourcenverwaltung des BS angesiedelt. Hier werden interne Datensatztypen, und Verkettungsmechanismen über Pointer, physischer Indizierung der Datensätze, Hashfunktionen und Speicherformate definiert.

Die **externe** Ebene dient dazu, das der User nur die Daten sieht, die er benötigt, und nicht auch die, die er nicht sehen darf, oder die er nicht versteht. Es wird ein spezieller Ausschnitt der Datenmenge dargestellt, und die erlaubten Operationen mit den Daten werden zur Verfügung gestellt.

Dazwischen liegt die **konzeptionelle** Ebene. Hier erscheinen die Daten nicht als Bytes auf dem Speichermedium, und auch nicht als Felder einer Eingabemaske, sondern als Datenobjekte. (Zitat Ende)"

2.8 Normalisierung

"(Zitat) **Normalisierung:** Hat man eine relationale Datenbank so entworfen, ist der nächste Schritt, zu prüfen, ob die Datenbank so strukturiert ist, daß jede Tabelle so aufgebaut ist, daß das Hinzufügen oder Löschen einer Reihe (eines Eintrages in ihr) nicht Informationen verändert oder löscht, die anderwärts noch benötigt werden. Z.B. daß das Löschen eines Titels in der Titelaufnahmedatei (weil z.B. das betreffende Buch verloren ging) nicht die Informationen über den Autor löscht (die man evtl. in Zukunft durchaus noch braucht).

Treten solche Probleme auf, dann spricht man von Modifikations-Anomalien (modification anomalies), d.h. unerwünschten Folgen für die Datenbank, wenn Daten in einer Tabelle hinzugefügt, geändert oder gelöscht werden. Solche Anomalien treten dann auf, wenn eine Tabelle Einträge enthält, die sinnvollerweise auf zwei oder mehr Tabellen verteilt werden sollten. Das Aufspalten einer Tabelle in zwei oder mehrere Tabellen, um solche Anomalien zu beseitigen, nennt man Normalisierung (normalization).

Zwischen 1970 und 1981 betrieb man Forschung, wie man Modifikations-Anomalien vermeiden kann. Je nach der Quelle der

Modifikations-Anomalie, die man auschließt, unterscheidet man seither verschiedene Normalformen (normal forms):

1. Erste Normalform (first normal form) (1NF)
2. Zweite Normalform (second normal form) (2NF)
3. Dritte Normalform (third normal form) (3NF)

Erste Normalform (first normal form) (1NF)

Eine Tabelle hat die erste Normalform, wenn sie folgende Bedingungen erfüllt:

- sie ist zweidimensional mit **Reihen** und **Spalten**
- jede **Reihe** enthält Daten, die zu einem **Objekt** oder einem Teil eines Objektes gehören
- jede Spalte enthält Daten für ein einziges Attribut des Objektes
- jede Datenzelle (Schnittstelle zwischen Reihe und Spalte) enthält einen **einzigen** Eintrag
- Jede Spalte muß einen (in der Tabelle) **einmaligen** Namen tragen
- **keine** zwei Reihen dürfen **identisch** sein
- die **Reihenfolge** der Spalten und Reihen ist **bedeutungslos**

Zweite Normalform (second normal form) (2NF)

In der zweiten Normalform hängen **alle** Spalten (Attribute), die nicht zum Primärschlüssel gehören, vom **Primärschlüssel** als ganzem ab. Alle Tabellen in der ersten Normalform, die einen einfachen (aus nur einer Spalte bestehenden) Primärschlüssel haben, sind so automatisch auch in der zweiten Normalform. Ist aber der Primärschlüssel zusammengesetzt, ist für die zweite Normalform zu prüfen, ob es Attribute gibt, die in Wirklichkeit nur von einer oder einem Teil der Spalten des Primärschlüssels abhängen. Ist dies der Fall, ist mit diesem Teil des Primärschlüssels eine neue Tabelle zu bilden und die ursprüngliche Tabelle in zwei Tabellen aufzuteilen. (Einzelheiten und Beispiele entnehme man den angeführten weiterführenden Ressourcen).

Dritte Normalform (third normal form) (3NF)

Eine Tabelle in der dritten Normalform enthält **keine** transitiven Abhängigkeiten, d.h. Abhängigkeiten der Form, daß ein Attribut A1 von einem anderen Attribut A2 abhängt, welches wieder von einem anderen Attribut A3 abhängt. Solche Abhängigkeiten können bewirken, daß das Löschen einer Reihe mit einem Eintrag von A3 über A2 zu einem Informationsverlust für A1 führt. (Zitat Ende)"

"(Zitat) **Abhängigkeit und Normalisierung**

Innerhalb des relationalen Modells spielt die Abhängigkeit von Attributen bzw. Attributskombinationen eine bedeutende Rolle. Abhängig ist ein Attribut A von einem Attribut B dann, wenn zu einem bestimmten Wert von A höchsten ein Wert von B möglich ist, d.h. in einem Datensatz vorausgesagt werden kann, daß immer dann wenn A den Wert X hat, B den Wert Y haben muß. Diese **Abhängigkeit** nennt man **funktional**. Eine besondere Stellung hat der **Identifikationsschlüssel**: er ist ein Attribut (oder eine Attributskombination) von dem alle anderen Attribut einer Relation funktional abhängig sind. Besteht der Identifikationsschlüssel aus einer Attributkombination, so darf keines dieser Attribute von einem anderen Attribut des Identifikationsschlüssels abhängig sein. Von **transitiverAbhängigkeit** spricht man dann, wenn ein Attribut C von einem Attribut A derart abhängig ist, daß C funktional von einem Attribut B abhängt, das seinerseits funktional von A abhängt. (...)

Erste Normalform

Eine Relation befindet sich in der 1. Normalform, wenn ihre Attribute **nur einfach Attributwerte** aufweisen. **Mengen** sind damit als Attributwerte **ausgeschlossen**. Auch wenn nach dem ersten Normalisierungsschritt als Werte noch Wertemengen verwendet werden - ein häufiger Fehler bei der Datenbankerstellung - können diese prinzipiell von der Datenverwaltung nicht mehr befriedigend angesprochen werden.

Zweite Normalform

Eine Relation ist in der 2. Normalform, wenn sie in der 1. Normalform ist und **jedes nicht** zum **Identifikationsschlüssel** gehörige Attribut voll **von diesem** abhängig ist.

Dritte Normalform

Eine Relation befindet sich in der 3. Normalform, wenn sie in der 2. Normalform ist und **kein** Attribut, das nicht zum Identitfikationsschlüssel gehört, **transitiv** von diesem abhängig ist.

2.9 Datenbanken und Vernetzung

2.9.1 Das Client-Server Modell

Erstmal zu den Begriffen:

ALLGEMEIN:

Client: Begriff aus dem Netzwerkbereich: ein Client nimmt Dienste in Anspruch, deshalb wird eine an den Server angeschlossene Arbeitsstation als Client bezeichnet. Der Client schickt Anfragen des Benutzers in einem speziellen Protokoll an den Server und stellt dessen Antworten in lesbarer Weise auf dem Bildschirm dar.

Client / Server-Architektur: Modell einer Netzwerkstruktur oder ein Datenbankkonzept, bei der / bei dem eine hierarchische Aufgabenverteilung vorliegt. Der Server ist dabei der Anbieter von Ressourcen,
Dienstleistungen und Daten - die Arbeitsstationen (Clients) nutzen sie.

Server: von "to serve" (dienen, jemanden versorgen) abgeleitet: zentraler Rechner in einem Netzwerk, der den Arbeitsstationen / Clients Daten, Speicher und Ressourcen zur Verfügung stellt. Auf dem Server ist das Netzwerk-Betriebssystem installiert, und vom Server wird das Netzwerk verwaltet. Im WWW sind Server Knotenpunkte des Netzes.

INSBESONDERE AUF DBMS BEZOGEN:

Client/Server: die Clients übernehmen alle Anwendungen, der Server (der Serverteil des DBMS) beschränkt sich auf den unmittelbaren Umgang mit den Daten. Der Server ist also ein echter Database Server.

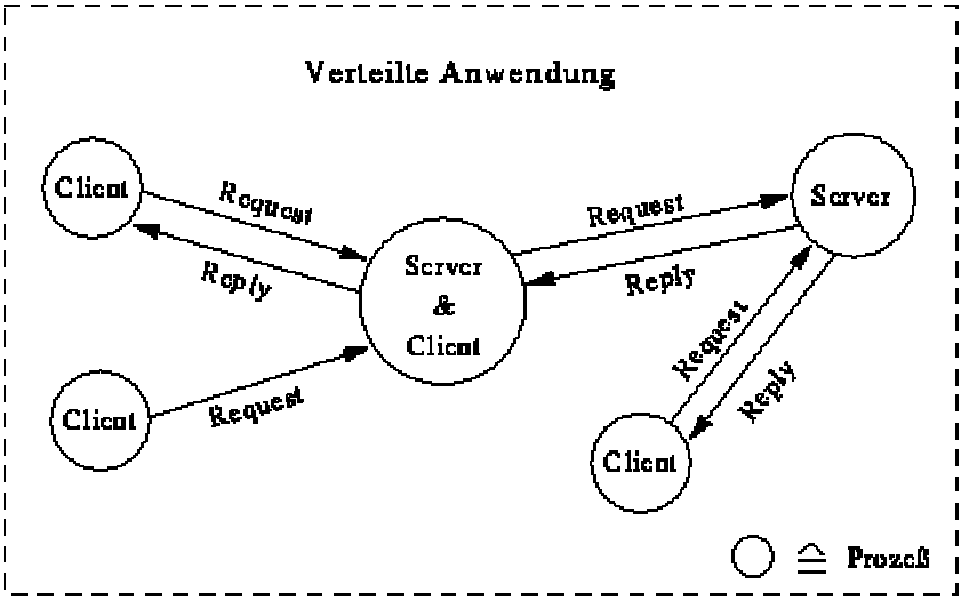
Client/Server-Datenbanken: Client/Server-Datenbanken ermöglichen wie dateiorientierte Datenbanken die Datenspeicherung, verfügen jedoch überzusätzliche Funktionen. Bei Client/Server-Modell wird eine Anwendung in zwei Komponenten untergeteilt: Ein Front End-Client, der dem Benutzer die Informationen anzeigt, und ein Back End-Server, der Daten speichert, abruft und manipuliert und den Grossteil der Verarbeitung für den Clien ausführt. Bei einem Client/Server-System ist der Server gewöhnlich ein leistungsstärkerer Computer als der Client und dient als zentraler Datenspeicher für viele Clien-Computer. Auf diese Weise ist das System leichter zu verwalten
Typische Beispiele: -freigegebene Datenbanken
-Remote-**Datei**-Server
-Remote **Druck**-Server

Vorteile von Client/Server-Datenbanken
-Leistungsstarke Opererationen,weil Dateien von einem einzigen Datenbank-Server verarbeitet werden
-Erhebliche Verbesserung von der Leistung bei betimmten Operationen, insbesondere wenn Low-End-Computer mit langsamen Pozessoren und wenig Arbeitsspeicher (RAM) als Benutzerarbeitsstation verwendet wird. Zum Beispiel kann ein umfangreiche Abfrage auf einem High-End-Rechnerwesntlich schneller als auf einer Client-Arbeitsstation erledigt werden.
-Verringerung der Netzaktivität, weil die Daten auf effiziente Weise übermittelt werden. Nur die tatsächlich von der Anwendung benötigten Daten werden übertragen.
-Kritische Funktionen , wie z.B. Transaktionsprotokolle, komlizierte Backup-Funktionen, redundante Festplatten-Arrays Tools zur Fehlerbehebung werden unterstützt.

"(Zitat) **Das Client-Server Modell**
In diesem Modell unterscheidet man zwei verschiedene Arten miteinander kooperierender Prozesse: die Server, die einen Dienst (Service) bereitstellen und die Klienten, die Nutzer eines solchen Dienstes. Ein Klient sendet eine Anfragenachricht,

in der er einen bestimmten Dienst nachfragt, an einen Server. Dieser erfüllt den Dienst, wie in Abbildung 3 dargestellt, indem er die nachgefragten Daten oder eine Fehlermeldung zurückliefert, die den Grund des Versagens beinhaltet.

Abbildung 3: Das Client-Server Modell



Es ist jedoch auch möglich, daß ein Prozeß gleichzeitig sowohl Klient als auch Server ist. Wenn ein Server für die Bereitstellung eines Dienstes den Dienst eines anderen Servers benötigt, wird er zum Klienten bezüglich des anderen Servers. (Zitat Ende)"

Fat Client

z.B. folgenden Befehl an Server: select from Uniangehöriger where name like 'M_ller' => liefert vielleicht 100 Datensätze, den "Rest" (also das raussuchen, was wirklich relevant ist) macht der client selbst. => Gegenteil von **thin client**, der nur "dummer" Empfänger ist

2.9.2 Verteilte Datenbanken (Distributet Database)

Erstmal zu den Begriffen:

> **Distributed Data Processing:** ein Client-Server-System mit sowohl **mehreren Servern** als auch **mehreren Clients**, wobei verschiedene Server **dieselbe** Datenbank oder **identische** Teile derselben Datenbank enthalten können.
> **Vorteil**: geringer Datenverkehr, da man die Daten dort speichert, wo sie zumeist gebraucht erden.
> **Problem**: die Daten auf allen Servern immer auf dem gleichen Stand halten.

Der **folgende Text** geht zwar etwas über das Thema hinaus (es kommen Intratnet, etc. drin vor), bleibt aber allg. im Bereich der "Ruhland-Winfo-Themen" und ich halte ihn für sehr gut lesbar:

"(Zitat) Eine verteilte Datenbank ist i.d.R. dadurch gekennzeichnet, daß ein inhaltlich, logisch zusammengehöriger Datenbestand physisch auf mehrere Speichermedien verteilt wird. Das Ausmaß der Verteilung reicht von lokalen Netzwerken (LAN - Lokal Area Network) bis hin zu globalen Systemen wie das Internet (WAN - Wide Area Network). Die Verteilung von Datenbanken und Systemen reflektiert die beobachtbare Tendenz zu zunehmender Verteilung traditionell zentralisierter Geschäftsprozesse. Dadurch werden neue Formen der Handelsbeziehungen und Märkte eröffnet. Datenbanken spielen in diesem Zusammenhang eine wesentliche Rolle, da sie durch die globale Vernetzung als Informationsquellen dienen. Vor diesem Hintergrund werden daher die verschiedenen Formen von Datenbanken kurz beschrieben und auf die Vor- und Nachteile speziell der verteilten Datenbanken eingegangen.

a) Verteilte Datenbanken als Weiterentwicklung zentraler Datenbanken

Das Mainframekonzept als Ausgangssituation

Das Konzept zentraler Datenbanken entstand in den 60iger Jahren aufgrund der Notwendigkeit zur effizienter Verwaltung von Daten (insbes. der Buchhaltung). Dieses war nur mit Hilfe von Großrechnern (Mainframes) zu bewerkstelligen, da es noch keine leistungsfähigen Deskto-

prechner gab. Die Datenabfrage erfolgte über ein Terminal, welches weder Prozessor noch Festwertspeicher besaß und mit dem Mainframe verbunden war (thin client). Dieses System, welches ursprünglich für die Buchhaltung konzipiert war, gewann zunehmend an Bedeutung für die übrigen Abteilungen (Beschaffung, Produktion, Logistik) und wurde durch das geänderte Anforderungsprofil zu einer zentralen Datenverarbeitungsanlage für den ganzen Betrieb weiterentwickelt. Der Großrechner mußte die Anfragen aller vernetzter Abteilungen bearbeiten und die Daten entsprechend an die lokalen Terminals übermitteln. Mit steigender Anzahl von Systemanfragen (queries) erwies sich die Leistungsfähigkeit der Großrechner als zunehmend unzureichend, um eine hinreichend zufriedenstellende Abfragegeschwindigkeit sicherzustellen. Im Zuge der Entwicklung von leistungsfähigen Desktoprechnern wurde das Mainframekonzept sukzessive durch Systeme dezentraler Datenhaltung ersetzt.

b) Verschiedene Ansätze zur Gestaltung von verteilten Datenbanken

Horizontale Fragmentierung

Eine Möglichkeit zur Verteilung von Datenbeständen liegt in der horizontalen Fragmentierung. Hierbei werden die Tupel (Datensätze) einer Relation auf verschiedene Speicherorte aufgeteilt. Dies ist dann sinnvoll, wenn Abfragen an die Datenbank von verschiedenen Standorten erfolgen und nur spezielle Datensätze von Interesse sind. Ein Beispiel hierfür ist eine Datenbank, die Kundendaten speichert. Es ist ratsam die Kundendaten in der Filiale zu speichern, die für den Kunden verantwortlich ist.

Vertikale Fragmentierung

Bei vertikaler Fragmentierung werden Attribute einer Relation getrennt gespeichert. Das bedeutet, man verteilt eigentlich zusammengehörige Datensatzbestandteile. Die Marketingabteilung interessiert z.B. nur der Name und die Anschrift eines Kunden, während für die Buchhaltung andere Fakten über den selben Kunden von Wichtigkeit sind. Hier bietet sich an, die jeweiligen Kundenattribute in den einzelnen Abteilungen zu speichern. Diese Form der Verteilung gebietet aber, daß in jeder Relation ein einheitlicher Primärschlüssel (eindeutiger Identifikator eines Tupels) vorliegt.

Gemischte Fragmentierung

Unter gemischter Fragmentierung versteht man die Kombination aus horizontaler und vertikaler Fragmentierung. Das Verteilen von Datenbanken (sowohl horizontal als auch vertikal) kann vollkommen redundanzfrei geschehen. Dieses führt zu niedrigen Abfragegeschwindigkeiten und zu meistens unzureichender Verfügbarkeit der Daten. Allein bei Änderung eines Tupels kann eine komplett redundanzfreie Speicherung Vorteile bieten, denn es kann nicht zu inkonsistenten Datenbeständen kommen und aufwendige Abgleichverfahren fallen weg.

c) Gründe für die Dezentralisierung

Hohes Leistungspotential

Wie am Anfang geschildert, entwickelte sich die Idee verteilter Datenbanken aus der zunehmenden Unzulänglichkeit zentraler Datenbanken die aufkommende "Abfrageflut" zu bewerkstelligen. Das Fragmentieren von Datenbeständen führt zu einem leistungsfähigeren Informationssystem, dessen sich mehrere Nutzer effektiver ohne lange Wartezeiten bedienen können. Redundante Speicherung ist dabei ein wichtiges Kriterium, denn auch bei vollkommener Redundanzfreiheit kann es zu Abfagestaus kommen.

Hohe Datensicherheit (Verfügbarkeit)

Die Abhängigkeit von einer einzigen zentralen Datenbank ist oft unerwünscht, da bei Ausfall der zentralen Rechneneinheit keine Abfragen mehr bearbeitet werden können. Der Ausfall eines Rechners in einem Netzwerk verteilter Datenbanken blockiert nicht automatisch alle anderen Knoten. Der Schaden kann hier auf den Teil der Datenbank, der auf dem ausgefallenen Rechner gespeichert wurde, begrenzt werden.

Hohe Flexibilität

Die Erweiterung eines verteilten Datenbanksystems ist i.d.R. leichter und kostengünstiger als der Ausbau einer zentralen Lösung. Dieses gilt insbesondere für die konzeptuellen Entwurf (Planungsebene), denn bei ungeteilten Systemen muß die Erweiterung an die gesamte Datenbank angepaßt werden.

Niedrigere Hardwarekosten

Teilbereiche von Datenbanken lassen sich heute schon auf schnellen Home PC implementieren, während zentrale Datenbanksysteme teure Großrechner voraussetzen, die vornehmlich von IBM vertrieben werden. In den letzten Jahren stieg die Leistungsfähigkeit der PC exponentiell an; der PowerPC Chip von IBM, Motorola und Apple spielt hierbei eine nennenswerte Rolle.

d) Probleme und Lösungsansätze bei Einsatz verteilter Datenbanken

Redundanzen und damit verbundene mögliche Inkonsistenzen

Die meisten oben genannten Gründe für ein dezentrales Datenbanksystem erfordern eine partielle redundante Speicherung eines Tupels. Dies bedeutet, daß **ein** Faktum auf **mehreren** Servern gleichzeitig abgelegt wird. Wenn dieser Datensatz auf *einem* Speicherorte von dem Nutzer geändert wird, wird er nicht zwangsläufig auch auf dem zweiten Server aktualisiert. Es kommt zu einer Inkonsistenz in der Datenbank, die diese als Informationsquelle unbrauchbar macht. Um aber trotzdem eine redundante Speicherhaltung zu ermöglichen, gibt es verschiedene Arten des Abgleichs von Datenbeständen. Das Zweiphasen-Commit-Verfahren sieht einen sofortigen Abgleich der Daten bei einer Änderung vor. Die gesamte Datenbank wird für die Zeit gesperrt, bis jeder Server eine positive Rückmeldung abgegeben hat. Andernfalls wird die Transaktion auf allen Rechnern rückgängig gemacht (roll back) und somit die Datenbank in den ursprünglichen Zustand versetzt. Das Replikationsverfahren stellt die zweite Möglichkeit zum Abgleich dar. Hierbei übernimmt ein spezieller Rechner (replication server) die Aufgabe, Änderungen über einen festgelegten Zeitraum zwischenzuspeichern und zeitversetzt alle Teile des Netzwerks zu aktualisieren.

Erhöhter Update- und Wartungsaufwand

Die Dezentralisierung von Datenbeständen macht es zwingend notwendig, das Datenbankprogramm auf die verschiedenen Rechner des Netzwerkes zu installieren. Bei einem anstehenden Versionswechsel des Datenbanksystems muß jeder Knoten aktualisiert werden (update), was mit erhöhten Kosten im Falle starker Fragmentierung verbunden ist. Die Software muß mit mehrfachen Lizenzen gekauft werden und die Implementierung auf allen Knoten dauert erheblich länger als bei einem zentralen Datenbanksystem, welches zu höheren Lohnkosten führt. Dieses gilt auch in bezug auf Wartung und Systempflege.

Systemerweiterungen durch unautorisierte Personen ("Wildwuchs")

Ein in der Praxis häufig beobachtbares Phänomen verteilter Datenbanken ist der sogenannte Wildwuchs, bei dem das dezentrale System von Datenbankanwendern (Sekretariat, Aussendienstmitarbeiter) unsystematisch für ihre Aufgaben erweitert werden. Das kann zu Lücken in der betriebsweiten Informationsversorgung führen, die nur unter erheblichen Kosten wieder zu beheben sind. Diese Problematik kann mit strenger Kompetenzverteilung und Zugangssperren, die nur autorisierten Personen Änderungen gestatten, vermieden werden.

e) Integration verteilter Datenbanken in verteilte Informationssysteme

Das Intranet als unternehmensinternes Kommunikationsmedium

Das Intranet stellt eine gängige Art der firmeninternen Kommunikation dar. Hierbei wird die Methode des Internets für geschlossene Firmennetze verwendet. Der Informationsaustausch erfolgt auf Basis des TCP/IP (Transmisson Control Protocol/Internet Protocol) Protokolls. Mit Hilfe des Konzepts des WWW (World Wide Web) und des Protokolls HTTP (Hyper Transport Text Protocol) sowie eines hypertextbasierten WWW-Browser wird eine einheitliche Oberfläche gestaltet, die es ermöglicht, auf einfachem Wege Informationen auszutauschen. Zu den wichtigsten Diensten, die aus dieser Art der Kommunikation resultieren, gehören das WWW, FTP (File Transfer Protocol) sowohl E-Mail (elektronische Briefe).

Formatierte und unformatierte Datenbanken innerhalb eines Intranets

Das Bereitstellen von Informationen bzw. Daten in Form von HTML (Hypertext Markup Language) weist den Nachteil auf, daß sie in der Regel nicht formatiert vorliegen. Das bedeutet, es gibt nicht unbedingt eine einheitliche "Maske" für die vorliegenden Daten. Eine Abfrage bzw. Suche von Datensätzen kann nur über eine Stringsuche geschehen, nicht wie bei formatierten Datenbanken über einen vorgegebenen Formalismus. Der Nachteil einer Stringsuche liegt in ihrer Ungenauigkeit bezüglich des Ergebnisses, was zu erhöhter Suchaufwand führt. Um die Vorteile einer formatierten Datenbank auch innerhalb eines Intranets nutzen zu können, haben Softwarehersteller solcher Datenbanken Einbindungsmöglichkeiten ihrer Produkte entwickelt. Es ist somit möglich über die einfache Bedienung durch das WWW-Konzept auf formatierte Datenbanken zuzugeifen. Abfragesprachen wie etwa SQL (Structured Query Language) werden somit nicht mehr benötigt. Kosten für Aus- und Weiterbildung von Mitarbeitern werden durch diese Symbiose aus Intranet und formatierter Datenbank gesenkt sowie die Effizienz durch präzisere Informationen gesteigert.

Unternehmensübergreifende Netzwerke: das Konzept des Extranet

Im Rahmen eines Intranets werden vorwiegend innerhalb einer Unternehmung elektronische Informationsstrukturen aufgebaut. Erweitert man das Netz auch auf Lieferanten und Kunden, spricht man von einem Extranet. Hierbei können Handelspartner auf Teile der Datenbank zugreifen und im Idealfall auch umgekehrt. Ein Kunde ist beispielsweise somit in der Lage, die Produktdatenbank eines Anbieters auf Preis und Verfügbarkeit zu durchsuchen. Bei Nichtverfügbarkeit des Artikels ist es vorstellbar, daß automatisch die Datenbank des Lieferanten auf mögliche Liefertermine abgefragt und dem Kunden weitergegeben wird. Aus welcher Datenbank die Information entnommen wurde, bleibt dem Kunden normalerweise unbekannt. So werden einzelne, verteilte Datenbanken zu einem unternehmensübergreifenden Informationssystem. Voraussetzung dafür ist eine strikte Standardisierung der verwendeten Systeme. Allgemein führt dieses Konzept der Informationsverteilung zu einer Erhöhung der Markttransparenz und zur Senkung von Transaktionskosten; Handelspartner werden schneller und global gefunden. (Zitat Ende)"

2.9.3 Internetdatenbanken

Läuft halt über's Internet... (siehe auch Text unter 2.9.2 und Kapitel 4 & 5)

2.10 XML

Grundprinzip von XML: Ich definiere im Dokument selber, wie meine "Felder" heißen sollen. Und zwar nach der Syntax <nameFeld> hier schreib ich rein was ich lustig bin </name feld>. Ähnlich wie bei html werden auch hier mit tags (=Befehl in Eckigen Klammern) irgendwas definiert. Und damit der Rechner weis, wo z.B. das Feld "Name" beginnt, un wo es aufhört, unterscheidet man zwischen dem einleitenden tag <Name> und dem abschließenden tag</name>. Und das benutzt man halt auch für Datenbank-Abfragen via Internet. Mehr kann ich jetzt so in kurz-Form nicht dazu sagen. Wer Lust hat, kann sich den folgenden Text mal durchlesen und/oder die Seite http://members.aol.com/xmldoku/ besuchen.

Was ist XML?

XML (Extensible Markup Language)

XML ist eine vereinfachte Form von SGML. So wie HTML mit SGML definiert ist, so kann man mit XML eigene Markup-Sprachen oder auch eigene Erweiterungen von HTML bzw. XHTML mit eigenen Tags für bestimmte Elemente mit bestimmten logischen Bedeutungen definieren.

Die mit XML definierten Markup-Sprachen werden als XML-Anwendungen bezeichnet. Die Syntax, Struktur und Bedeutung der Tags wird für jede XML-Anwendung mit einer DTD oder einem Schema definiert. Die Verarbeitung kann mit XML-Parsern mit DOM oder SAX erfolgen. Wie die Elemente sichtbar dargestellt werden sollen, kann mit XSL oder CSS definiert werden. XML-Dokumente können auch Hypertext-Links enthalten, entweder wie in HTML oder in der Form von XLink oder XPointer.

XML-Anwendungen eignen sich einerseits für die Darstellung in Web-Browsern - also als Ersatz oder Ergänzung von HTML - und andererseits für die Verarbeitung mit EDV-Programmen (z.B. in der Textverarbeitung, Tabellenkalkulation, Datenbanken, kommerziellen Anwendungen u.a.) und als Austauschformat zwischen solchen Programmen - also als Ersatz für RTF, CSV und EDI.

Warum XML?

XML ist eine Metasprache zur Definition von Markup-Sprachen. So wie HTML mit SGML definiert ist, so kann man mit XML eigene Markup-Sprachen definieren, und künftige Versionen von HTML werden ebenfalls mit XML definiert werden (XHTML).

So wie HTML festgelegt und normiert ist und daher für den weltweiten Austausch und die Übertragung und Verwendung von Web-Pages zwischen vielen verschiedenen Web-Servern und Web-Browsern geeignet ist, so kann man mit XML eigene Datei-Strukturen für verschiedene Zwecke definieren und normieren, die dann ebenfalls von vielen Personen mit vielen verschiedenen Programmen und auf vielen verschiedenen Rechnern verwendet werden können. Mit der Hilfe von Style-Sheets können XML-Dokumente außerdem ebenfalls, so wie HTML-Files, von Web-Browsern dargestellt und ausgedruckt werden.

Wofür kann man nun solche mit XML definierte Markup-Sprachen, sogenannte "XML-Applikationen", verwenden? Welchen Zweck kann es haben, mit XML solche Sprachen festzulegen und zu normieren? Welche Vorteile haben solche XML-Anwendungen gegenüber HTML oder anderen Datei-Formaten?

- Mit XML kann man die logische Bedeutung von Daten, Informationen und Texten definieren - ähnlich wie die Tabellen- und Spalten-Bezeichnungen in Datenbanken und Tabellenkalkulationen.
- XML ermöglicht im Gegensatz zu HTML die Definition eigener oder zusätzlicher "Befehle" (Tags) - ähnlich wie bei der Definition von Macros in der Textverarbeitung
- XML-Applikationen eignen sich als Plattform- und Software-unabhängiges Austausch-Format für Daten zwischen verschiedenen Programmen und Rechnern - ähnlich wie RTF für Texte, CVS für Tabellen, EDI für kommerzielle Anwendungen - aber in einem einheitlichen, allgemein verwendbaren, Hersteller-unabhängigen Format.

Außerdem ist die Syntax von XML so streng festgelegt, daß XML-Anwendungen wesentlich einfacher, bequemer und effizienter von Programmen weiter verarbeitet werden können als HTML-Files.

Markup und Darstellung

Die HTML-Befehle beschreiben eigentlich nur, in welcher Art und Weise die Textteile strukturiert werden sollen (Überschrift, Absatz, Liste, Tabelle, Normalschrift, Fettschrift usw.).

Beispiel:

Der folgende HTML-Code

```
<p Hubert Hans <bPartl</b
<br Muthgasse 18
<br A-1190 Wien
<br geb. 8. M&auml;rz 1949
</p
```

bewirkt eine Darstellung wie

Hubert Hans **Partl**
Muthgasse 18
A-1190 Wien
geb. 8. März 1949

- - -

Mit XML kann man Tags definieren, die die Bedeutung der Informationen angeben. Mit einer geeigneten DTD kann die obige Information also in einem XML-Dokument in der folgenden Form enthalten sein:

```
<person id="p4681"
  <vornameHubert</vorname
  <vornameHans</vorname
  <zunamePartl</zuname
  <titelDr.</titel
  <adresseMuthgasse 18</adresse
  <plzA-1190</plz
  <ortWien</ort
  <geburtstag
    <tag8</tag
    <monatMärz</monat
    <jahr1949</jahr
  </geburtstag
</person
```

- - -

Mit einem geeigneten Style-Sheet bewirkt das XML-Dokument genau die selbe Darstellung wie das obige HTML-Dokument:

Hubert Hans **Partl**
Muthgasse 18
A-1190 Wien

geb. 8. März 1949

- - -

Mit einem anderen Style-Sheet kann dasselbe XML-Dokument aber auch so dargestellt werden (als Visitenkarte):

Hubert Hans Partl

Muthgasse 18

A-1190 Wien

- - -

oder so (in einer Liste):

Partl, Dr. Hubert Hans (1949)

- - -

Verarbeitung in Programmen

XML-Dokumente eignen sich nicht nur für die Darstellung in Web-Browsern und das Ausdrucken auf Papier, sondern auch für die weitere Verarbeitung in Programmen, in denen die logische Bedeutung der Informationen eine Rolle spielt.

Beispiel: Eine Suche nach dem Wort "März" innerhalb von Absätzen <p in HTML-Files der obigen Struktur würde nicht nur die Personen liefern, die im März geboren sind, sondern auch den Rektor mit dem Namen Leopold März sowie alle Personen, die in der Märzstraße wohnen. Eine Suche nach dem Wort "März" innerhalb der in den XML-Files mit <geburtstag bezeichneten Elemente würde hingegen wirklich nur die Personen liefern, die in diesem Monat Geburtstag haben. Außerdem kann man die in den XML-Files beschriebenen Personen leicht nach den einzelnen Datenfeldern sortieren, also z.B. nach der Postleitzahl oder nach dem Geburtsjahr, oder das Durchschnittsalter berechnen oder andere Verarbeitungen der Daten vornehmen.

XML-Syntax

Start- und End-Tags

Die meisten Befehle (Tags) in SGML- und XML-Anwendungen - wie auch in HTML - treten paarweise als Start- und End-Tags auf und geben an, welche Bedeutung der dazwischen liegende (eventuell durch weitere Tags unterteilte) Text hat:

<xxx ... Text ... </xxx

oder

<xxx yyy="zzz" ... ... Text ... </xxx

In HTML ist der End-Tag in vielen Fällen optional, d.h. er darf weggelassen werden. Der Web-Browser kann dann auf Grund der in der HTML-Norm festgelegten Bedeutung der HTML-Tags "erraten", an welcher Stelle er sich den nicht angegebenen End-Tag "dazudenken" muß.

Beispiel: Der HTML-Tag <p beginnt einen neuen Absatz. Der End-Tag </p muß nicht angegeben werden, denn immer wenn ein neuer Absatz mit <p oder eine Überschrift mit <h1 bis <h6 oder eine Liste mit <ul oder <ol oder <dl beginnt, bedeutet das automatisch das Ende des vorherigen Absatzes.

Bei XML-Anwendungen müssen (im Gegensatz zu HTML) die End-Tags *immer* angegeben werden und dürfen niemals weggelassen werden:

Beispiel:

Richtig ist

<p ... Text ... </p<p ... Text ... </p

Nicht richtig wäre

<p ... falsch ... <p ... falsch ...

Diese "strengere" Regel hat unter anderem die folgenden Gründe: Im Gegensatz zur festgelegten HTML-Norm soll der Anwender bei XML-Applikationen die Möglichkeit haben, nachträglich zusätzliche Tags in der DTD oder im Schema zu definieren, und daher können die Verarbeitungsprogramme nicht so wie der HTML-Browser immer nach denselben Regeln die fehlenden End-Tags selbst korrigieren. Außerdem sind die verarbeitenden Programme viel einfacher zu schreiben und können viel effizienter ablaufen, wenn sie sich darauf verlassen können, daß das XML-File syntaktisch richtig ist, und keine automatische Fehlerkorrektur programmiert werden muß.

Einzelne Elemente ohne End-Tag

In HTML gibt es Tags, zu denen es keinen End-Tag gibt, weil sie nicht die Eigenschaften eines Textbereiches definieren sondern einzelne, selbständige Elemente darstellen. Beispiele sind <br für einen Zeilenwechsel, <hr für eine Trennlinie oder <img für ein Bild.

Im XML müssen alle Tags der Form <xxx immer einen End-Tag der Form </xxx haben. Die XML-Tags, zu denen es keinen End-Tag gibt, müssen zur Unterscheidung davon in der Form

<xxx /

oder

<xxx yyy="zzz" ... /

gechrieben werden.

Nicht erlaubt ist also

<p ... <br ... falsch ... <br ... falsch ... </p

sondern man muß stattdessen richtig

<p ... <br / ... <br / ... </p

schreiben oder eventuell auch

<p ... <br</br ... <br</br ... </p

Der Grund für diese "strengere" Regel liegt ebenfalls in einer Vereinfachung der Programmierung: Das Verarbeitungsprogramm muß nicht darauf "warten", welche End-Tags eventuell noch kommen werden, sondern weiß immer sofort, ob es sich um ein fertiges Einzel-Element oder um den Start-Tag eines längeren Elementes handelt.

Groß- und Kleinschreibung

Im Gegensatz zu HTML ist bei XML die Groß- und Kleinschreibung *nicht* egal: <xxx und <Xxx und <XXX sind voneinander verschiedene Befehle.

Wenn ein Befehl als <xxx definiert ist, dann darf man nicht stattdessen <XXX schreiben, und auch eine Kombination wie

<XXX ... falsch ... </xxx

wäre nicht erlaubt.

Parameter (Attribute)

Auch die Regeln dafür, wann die Werte von "Attributen" (also Parametern innerhalb von Befehlen) zwischen Quotes-Zeichen eingeschlossen werden müssen, sind bei XML strenger als das, was manche Web-Browser in HTML zulassen. *Alle* Attribut-Werte sollen *immer* in Quotes-Zeichen eingeschlossen werden. Dabei dürfen entweder das ASCII-Zeichen Double-Quotes " oder das ASCII-Zeichen Apostroph ' verwendet werden (natürlich nur paarweise, nicht vermischt), aber nicht die ähnlich aussehenden typographischen Anführungszeichen oder Akzentzeichen.

Richtig sind also nur die folgenden beiden Varianten:

<xxx yyy="zzz" ...

<xxx yyy='zzz' ...

Entities

Wie in SGML (und daher auch in HTML) kann man auch in XML Entities definieren, bei denen einem Namen ein bestimmter Text zugeordnet wird. Diese Entities kann man dann überall im Text und auch in Parametern von Befehlen in der Form

&name;

verwenden.

Typische Beispiele sind:

< für das Kleiner-Zeichen <

> für das Größer-Zeichen

& für das Und-Zeichen &

Entities können aber nicht nur einzelne Zeichen sondern auch längere Textteile enthalten, wie die "Abkürzungen" in der Textverarbeitung.

Schachtelung von Tags

In XML müsssen Start- und End-Tags immer richtig geschachtelt werden.

Beispiel:

Richtig sind

<person <vorname ... </vorname ... </person

<vorname ... </vorname <zuname ... </zuname

Nicht erlaubt wären

<person ... <vorname ... </person ... falsch ... </vorname

<vorname ... <zuname ... falsch ... </vorname ... </zuname

Anmerkung: Diese Regeln gelten an sich auch für HTML genauso wie bei XML. Bei HTML können die Web-Browser aber auf Grund der festgelegten Bedeutung der HTML-Tags versuchen, zu erraten, was der Autor vermutlich gemeint hat, wenn er gegen diese Regel verstoßen hat.

Beispiel: Bei einer (nicht erlaubten) Kombination wie

<p ... <em ... </p <p ... </em ... </p

kann der Web-Browser möglicherweise erraten, welches Layout der Autor damit erreichen wollte. Und wenn er es nicht richtig errät, dann erscheint der Text zwar nicht in der richtigen Schriftart, aber zumindest immer noch vollständig lesbar und in der richtigen Absatz-Struktur.

Bei XML darf die Software *nicht* versuchen, solche Fehler durch Rateversuche zu korrigieren, sondern *muß* die Verarbeitung von fehlerhaften XML-Dokumenten immer mit einer Fehlermeldung abbrechen.

Ein Grund für diese "strengere" Regel ist, daß die XML-Befehle nicht nur das Layout des Textes (wie in HTML), sondern die Bedeutung der Textteile definieren, und falsche Interpretationen der Befehle daher nicht bloß zu unschönen, sondern zu logisch und inhaltlich falschen Ergebnissen führen würden.

Beispiele: Wenn ein Text-Abschnitt irrtümlich in kursiver statt fetter Schrift dargestellt wird, ist das für das Verstehen des Textes im allgemeinen nicht so tragisch wie wenn der Name mit dem Geburtsort oder der Einkaufspreis mit dem Verkaufspreis verwechselt wird.

Die XML-Tags dienen nicht nur für die Interpretation der sichtbar angezeigten Texte durch einen kritisch denkenden Menschen, sondern auch für die Weiterverarbeitung der Informationen durch automatisch laufende Computer-Programme. Deshalb müssen die Syntax-Regeln von XML strenger eingehalten werden als die von HTML.

XML-Anwendungen

Unter XML-Anwendung oder XML-Applikation versteht man die Festlegung (Normierung) von XML-Befehlen für eine Klasse von XML-Dokumenten gleicher Struktur, also für einen bestimmten Zweck.

Das Format und die Struktur der XML-Files sowie die Eigenschaften und die Schachtelung der darin vorkommenden Elemente (XML-Befehle, Tags, Entities) werden für eine XML-Anwendung mit einer DTD oder einem Schema definiert - so wie man bei EDI mit UNSM und MIG das Format und die Struktur der Nachrichten und die Bedeutung der darin enthaltenen Daten für einen bestimmten Nachrichtentyp definieren kann.

Beispiele für XML-Anwendungen

- DocBook (für gedruckte Texte und Bücher),
- WML (für Online-Informationen auf kleinen Displays wie z.B. Handys),
- XHTML (für Online-Informationen auf großen Displays wie z.B. PCs und Fernsehschirmen),
- MathML (für mathematische Formeln),
- CML (für chemische Formeln),
- SVG (für Vektor-Graphiken),
- u.v.a.

Definition von XML-Anwendungen

DTD (Document Type Definition)

Eine DTD beschreibt die Struktur einer Klasse von SGML- oder XML-Dokumenten, also einer SGML- oder XML-Applikation, mit Hilfe eines Text-Files, das alle Syntax-Regeln in einem von SGML vorgeschriebenen Format enthält. Beispielsweise ist jede HTML-Version durch eine DTD definiert. Eine Alternative dazu ist die Definition mit Hilfe eines Schemas.

Schema (Mehrzahl: Schemata)

Ein Schema beschreibt die Struktur einer Klasse von XML-Dokumenten, also einer XML-Applikation, ähnlich wie eine DTD, jedoch nicht in der DTD-Syntax sondern in einer eigenen XML-Syntax.

Verarbeitung von XML-Anwendungen

XML-Parser

Ein XML-Parser ist ein Programm, das ein XML-File liest und den Inhalt in der Form von DOM oder SAX liefert. Ein validierender Parser überprüft zusätzlich die Richtigkeit der Daten an Hand der DTD oder des Schemas.

DOM (Document Object Model)

DOM ist ein Objektmodell, es beschreibt die in einem Dokument einer bestimmten XML-Anwendung enthaltenen Elemente als Objekte, für die Verarbeitung mit einer objekt-orientierten Programmiersprache wie z.B. Java. DOM liefert eine komplette Baumstruktur aller Objekte eines XML-Dokuments und eignet sich daher nicht für extrem große XML-Files.

SAX (Simple API for XML)

SAX ist eine Programm-Schnittstelle (Application Programmers Interface API) für die Verarbeitung einer Klasse von XML-Dokumenten, also einer XML-Applikation, mit Hilfe einer objekt-orientierten Programmiersprache wie z.B. Java. SAX liefert ein XML-Element nach dem anderen in einem Eingabestrom und eignet sich daher auch für sehr große XML-Files.

Style-Sheets

CSS (Casading Style Sheets)

CSS ist ein vom W3-Consortium definiertes, einfaches Format für Style-Sheets für die Darstellung von HTML- und XML-Dokumenten.

DSSSL (Document Style Semantics Specification Language)

DSSL ist eine sehr mächtige und daher auch sehr komplexe Sprache für die Spezifikation der Darstellung von SGML-Dokumenten.

XSL (Extensible Style Language)

Mit XSL wird ein Style-Sheet definiert, das angibt, wie der in einem XML-Dokument definierte Inhalt vom Web-Browser oder von anderen Programmen dargestellt werden soll.

XSL ist mächtiger als CSS und DHTML:

- Mit **XSLT** (Transformation) kann man aus einem XML-File ein anderes XML-File machen, also z.B. bestimmte Elemente weglassen, die Elemente in anderen Reihenfolgen anordnen und zusätzliche Elemente hinzufügen,
- und mit **XSL-FO** (Formatierung) kann man das Layout der Darstellung für die Elemente festlegen.

Darstellung in Web-Browsern

Künftige Web-Browser werden XML-Files direkt am Bildschirm darstellen und am Drucker ausdrucken können, wenn mit einem Style-Sheet definiert ist, wie die einzelnen XML-Elemente dargestellt werden sollen. Erste Ansätze dafür gibt es im MS Internet Explorer Version 5 (mit einer Vor-Version von XSL Style-Sheets oder mit CSS) sowie in Netscape Version 5 und Mozilla (mit CSS Style-Sheets).

Zu diesem Zweck werden am Web-Server sowohl das XML-File mit dem Inhalt der Information als auch das XSL- oder CSS-File mit den Layout-Angaben abgespeichert - so ähnlich wie im Textsatzsystem LaTeX, wo der Inhalt im TEX-File und das Layout im STY-File definiert werden, und so wie dort hat man auch hier die Möglichkeit, den selben Inhalt wahlweise in verschiedenen Layouts darzustellen, z.B. für große und kleine Bildschirme und für Schwarz-weiß- und Farb-Drucker.

Wenn man die Informationen für alle Benutzer verfügbar machen will, also auch für diejenigen, die noch ältere Web-Browser verwenden, muß man sie am Web-Server (zusätzlich) im normalen HTML-Format zur Verfügung stellen - am besten mit einem Umwandlungsprogramm, das die XML-Files mit der Hilfe der Style-Files automatisch in Standard-HTML umwandelt. Und wenn die Informationen auch mit Handy-Telefonen und dergleichen erreichbar sein sollen, dann gleich auch noch im WML-Format.

Es gibt auch Umwandlungsprogramme, die "lockere" HTML-Files in "strenge" XHTML-Files umwandeln, damit diese mit XML-Software weiter verarbeitet oder auch durch zusätzliche Tags zu speziellen XML-Anwendungen erweitert werden können.