# Datenbanken und SQL

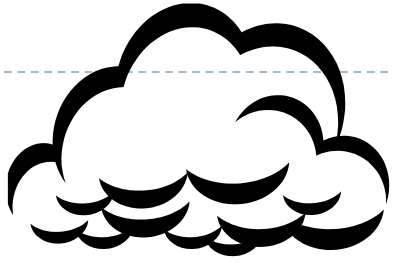
## Kapitel 9

## Moderne Datenbankkonzepte

# Moderne Datenbankkonzepte

- Verteilte Datenbanken
  - Vorteile der verteilten Datenbanken
  - Die 12 Regeln von Date zu verteilten Datenbanken
  - Das CAP-Theorem
  - BASE versus ACID
  - Überblick über moderne verteilte Datenbanken
  - Zwei-Phasen-Commit
- Objektorientierte Datenbanken
  - Definition
  - Objektrelationale Datenbanken
  - Objektrelationale Erweiterungen in Oracle
  - Eingebettete Relationen in Oracle

# Übersicht



- Neue wichtige Begriffe:
  - Cloud-Computing
    - Umfassender Begriff dafür, dass Daten im Netz gehalten werden
    - Der exakte Speicherort ist in der Regel nicht festgelegt und nicht bekannt
    - Beispiele:
      - Drop Box, OneDrive, IMAP (Email) usw.
      - Verteilte Datenbankserver von Amazon, Google, Facebook usw.
  - NoSQL
    - Begriff steht für: Not Only SQL
    - Erweiterung der Sprache SQL für neue Datenbankkonzepte
    - SQL ist sehr weit verbreitet, optimiert für relationale Datenbanken
    - Erweiterungen für nicht relationale Datenbanken sind erforderlich

# Vorteile verteilter Datenbanken

- Schnelle Verfügbarkeit
  - da Daten parallel zugreifbar
  - da Daten eventuell mehrfach gehalten werden
  - da deshalb Daten eventuell lokal direkt verfügbar
- Ausfallsicherheit
  - Wenn ein Knoten ausfällt, sind die anderen noch verfügbar
- Im Extremfall
  - gibt es tausende von Knoten
  - Daten werden redundant gehalten
- Aber: Synchronisierung geänderter Daten ist komplex

# Fundamentales Prinzip

- **Fundamentales Prinzip verteilter Datenbanken:**

  Ein verteiltes System sollte sich dem Anwender gegenüber genauso wie ein zentrales verhalten.

- **Dies heißt:**
  - Der Anwender bemerkt bei seinen Zugriffen keinen Unterschied, ob er zentral auf eine Datenbank oder auf viele verteilte Daten zugreift
  - Dies gilt auch für den Anwendungsprogrammierer!
- **Die folgenden 12 Regeln von Date basieren auf diesem Prinzip**

# Die 12 Regeln von Daten

- J.F. Date stellte 12 Regeln zu verteilten Datenbanken auf:

| | |
|---|---|
| 1. | Lokale Eigenständigkeit jedes Rechners |
| 2. | Keine zentrale Verwaltungsinstanz |
| 3. | Ständige Verfügbarkeit |
| 4. | Lokale Unabhängigkeit |
| 5. | Unabhängigkeit gegenüber Fragmentierung |
| 6. | Unabhängigkeit gegenüber Datenreplikation |
| 7. | Optimierte verteilte Zugriffe |
| 8. | Verteilte Transaktionsverwaltung |
| 9. | Unabhängigkeit von der Hardware |
| 10. | Unabhängigkeit von Betriebssystemen |
| 11. | Unabhängigkeit vom Netz |
| 12. | Unabhängigkeit von den Datenverwaltungssystemen |

# Regeln 1 und 2

- Lokale Eigenständigkeit jedes einzelnen Rechners
  - Jeder einzelne Rechner arbeitet möglichst autonom
  - Dies garantiert eine hohe Ausfallsicherheit
  - Dies erfordert einen hohen internen Kommunikationsaufwand
- Keine zentrale Instanz, die das System verwaltet
  - Fast eine Folgerung aus Regel 1: Wenn jede Instanz eigenständig ist, benötigen wir keine zentrale Instanz
  - Damit müssen sich alle einzelnen Rechner gegenseitig verwalten
  - Dies führt zu einem hohen internen Kommunikationsaufwand

# Regeln 3, 4 und 5

- Ständige Verfügbarkeit
  - Das gesamte System sollte nie abgeschaltet werden
- Lokale Unabhängigkeit
  - Jeder Zugriff ist unabhängig davon, wo sich die gewünschten Daten derzeit befinden, ob lokal oder entfernt
- Unabhängigkeit gegenüber Fragmentierung
  - Fragmentierung: Relationen werden auf mehrere Rechner verteilt (auch sharding genannt)
  - Der Anwender greift unabhängig von der Fragmentierung zu
  - Die Fragmentierung kann dynamisch sein: Daten werden meist dort gespeichert, wo sie häufig zugegriffen werden (Regionalisierung)

# Regeln 6, 7 und 8

- Unabhängigkeit gegenüber Datenreplikation
  - Die Zugriffe ändern sich nicht, falls Replikate existieren
  - Die Verwaltung der Replikate und die Konsistenz der Daten übernimmt das verteilte System
- Optimierung verteilter Zugriffe
  - Das Suchen der Daten im verteilten System und das Lesen und Schreiben werden intern optimiert
- Verteilte Transaktionsverwaltung
  - Transaktionen werden als atomare Einheiten voll unterstützt
  - Recovery und Concurrency werden voll unterstützt

# Regeln 9 bis 12

- Unabhängigkeit von der verwendeten Hardware
  - PCs, Großrechner, unterschiedliche Netze werden unterstützt
- Unabhängigkeit von den verwendeten Betriebssystemen
  - Windows, MacOS, Unix, usw.
- Unabhängigkeit vom verwendeten Netzwerk
  - Unterstützung aller wichtigen Protokolle, z.B. TCP/IP
- Unabhängigkeit vom verwendeten DBMS
  - Verwendung gemeinsamer Zugriffssprachen wie JSON, PDO
  - Unterstützung von SQL und NoSQL

# Zusammenfassung zu den 12 Regeln

- Regel 9 bis 11 (unabhängig von HW, OS, Netz):
  - Heute weitgehend erfüllt
- Regel 1 bis 6 (eigenständig, verfügbar, fragmentiert, repliziert):
  - Ehrgeizige Ziele, die teilweise schon erfüllt werden
- Regel 7, 8 und 12 (optimiert, Transaktion, DBMS-unabhängig):
  - Nicht widerspruchsfrei
  - Regel 7 und 12 erwarten einheitliche Schnittstellen und einen Transaktionsbetrieb
  - Regel 8 fordert verteilte Transaktionen, mit SQL allein kaum zu erfüllen

# Das CAP-Theorem

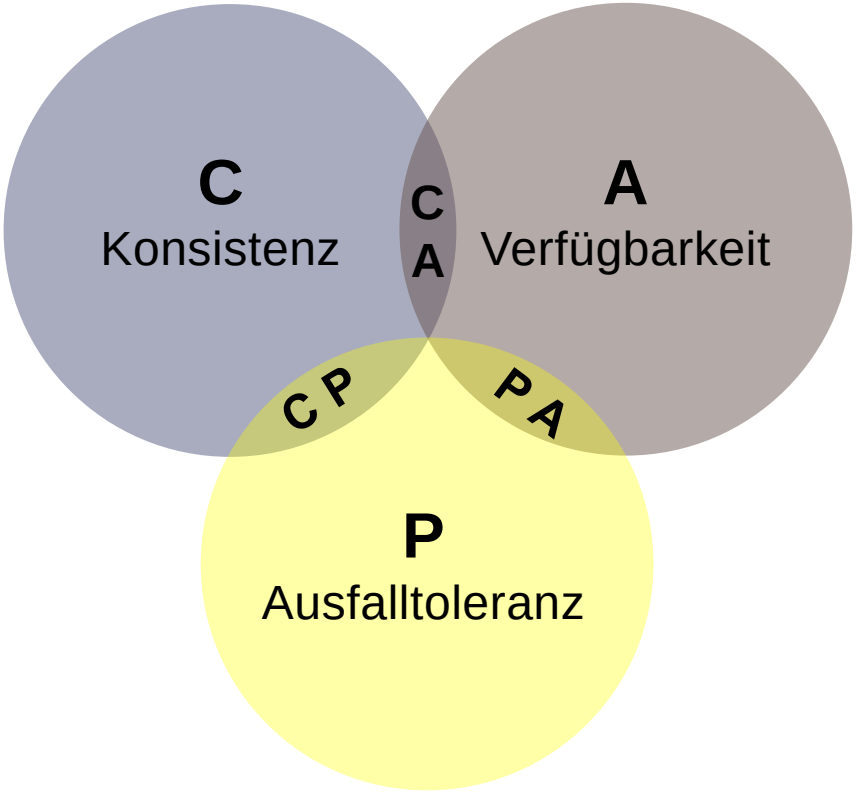
- Konsistenz (**C**onsistence),
  Verfügbarkeit (**A**vailability) und
  Ausfalltoleranz (Tolerance of Network **P**artitions)
  können in verteilten Datenbanken nicht gleichzeitig erfüllt werden.

  - Hier wird sehr gezielt auf das Problem des Widerspruchs der 12 Regeln von Date eingegangen.
  - Nur jeweils 2 der obigen Eigenschaften C, A und P können gleichzeitig vollständig erfüllt werden.

# C – A – P

- **C**onsistence (Konsistenz)
  - → Transaktionsbetrieb, ACID, Regel 8
  - Erfordert die Atomarität, Konsistenz bei redundanten Daten
- **A**vailability (Verfügbarkeit)
  - → Regel 3 und 7
  - Ein Teilausfall des Systems sollte nicht zum Gesamtausfall führen
- **P**artition-Toleranz (Ausfalltoleranz)
  - → in mehreren Regeln enthalten
  - Ein (vorübergehender) Verlust von Daten muss toleriert werden
  - Ein verspätetes Zustellen ist zu tolerieren

# CAP-Eigenschaften




- CA-Systeme:
  - Klassischer Bereich der relationalen Datenbanken
- CP-Systeme:
  - Daten in einzelnen Knoten können ausfallen. Eventuell Neustart des Systems
- PA-Systeme:
  - Hochverfügbares System auf Kosten der sofortigen Konsistenz

# Das Konsistenzmodell BASE

- **B**asically **A**vailable
  - Die Verfügbarkeit ist wichtiger als die Konsistenz
- **S**oft State
  - Konsistenz wird nach Transaktionsende fließend (soft) erreicht
- **E**ventual Consistency
  - Letztendlich wird die Konsistenz erreicht und garantiert

- Im klassischen Modell steht konträr dazu: **ACID**

# Überblick über moderne DB-Systeme (1)

- Key/Value-Datenbanken
  - Daten werden mit dem Schlüssel abgelegt
  - z.B. Amazon System Dynamo, Riak
  - Leicht zu skalieren, in Cloud-Systemen gerne angewendet
- Dokumentenbasierte Datenbanken
  - Als Dokumente abgelegt ohne fest vorgegebene Strukturen
  - Basiert auf Lotus Notes
  - z.B. MongoDB, CouchDB

# Überblick über moderne DB-Systeme (2)

- Spaltenorientierte Datenbanken
  - Daten werden spaltenweise verwaltet und gespeichert
  - Meist aber Mischformen mit Key/Value,
  - Anwendung in Big Table Konzept von Google
  - z.B. HBase von Microsoft, Cassandra von Facebook
- Graphen-Datenbanken
  - Grundlage ist die Graphentheorie, basiert auf Graphen
  - Sehr gut für rekursive Suche geeignet, für Navis, für Geodaten-banken, für soziale Netzwerke
  - z.B. GraphDB von Sones, Open Source Neo4J

# Zuordnung moderner DB-Systeme

| | CA-System | CP-System | PA-System |
|---|---|---|---|
| **Relationale Datenbanken** | Oracle<br>SQL Server<br>MySQL, … | | |
| **Key-Value Datenbanken** | | BerkeleyDB | Dynamo<br>Riak |
| **Dokumentenbasierte Datenbanken** | | MongoDB | CouchDB |
| **Spaltenorientierte Datenbanken** | | Big Table<br>HBase | Cassandra |

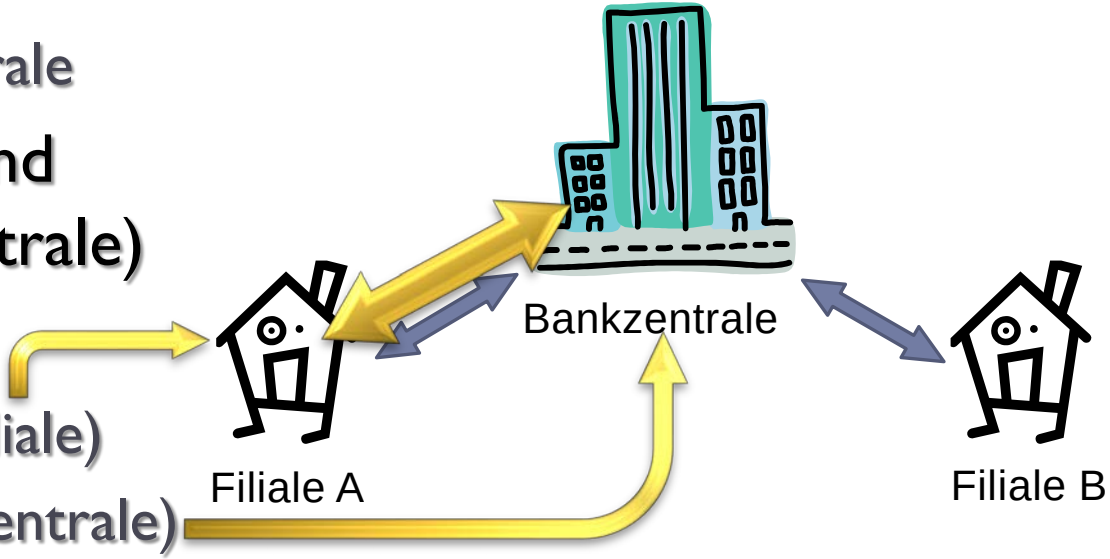
# Zwei-Phasen-Commit

- **Gegeben:**
  - Daten sind auf mindestens 2 Datenbanken verteilt
  - CA-System ist erforderlich
  - z.B. Überweisung von einer Bank auf eine andere

- **Idee:**
  - Jede Datenbank führt lokal ein eigenes Transaktionsprotokoll
    - → Phase 1
  - Übergreifend gibt es ein globales Transaktionsprotokoll
    - → Phase 2

# Die Idee des Zwei-Phasen-Commits

- Überweisung
  - von Filiale A zur Zentrale
- Zwei Datenbanken sind involviert (Filiale, Zentrale)
- Also:
  - Lokale Transaktion (Filiale)
  - Lokale Transaktion (Zentrale)
  - Koordination (Globale Transaktion)



# Das Zwei-Phasen-Commit

1. Jede Datenbank arbeitet getrennt im Transaktionsbetrieb
   - Jede Datenbank macht eigene Recovery (z.B. Logs, Checkpoints)
   - Jede Datenbank beendet mit einem „lokalen“ Commit
2. Eine Datenbank ist zusätzlich der Koordinator
   - Der Koordinator startet eine „globale“ Transaktion
   - Diese überwacht die „lokalen“ Transaktionen
   - Bei Erfolg aller lokalen Transaktionen wird dies an alle zurückgemeldet
   - Die „lokalen“ Commits werden dann in „globale“ umgewandelt

# Algorithmus des 2-Phasen-Commits

| | |
|---|---|
| **Lokales Abarbeiten einer Transaktion** | Jede betroffene Datenbank startet eine lokale Transaktion: Lokale Änderungen werden in der lokalen Logdatei protokolliert. (Beginn der **Phase 1**) |
| **Melden des Transaktionsendes** | Am Ende einer lokalen Transaktion erfolgt eine entsprechende Meldung (Commit bzw. Rollback) an den Koordinator (Ende der **Phase 1**). |
| **Globales Transaktionsende** | Der Koordinator sammelt alle lokalen Meldungen. Liegen nur erfolgreiche Rückmeldungen vor, so wird ein globales Commit, ansonsten ein Rollback eingetragen (**Phase 2**). |
| **Endgültiges lokales Transaktionsende** | Das Ergebnis der globalen Transaktion wird an alle lokalen Rechner geschickt. Jeder lokale Rechner übernimmt das globale Ergebnis (Commit oder Rollback) als endgültiges. Erst jetzt ist die Transaktion abgeschlossen (Ende der **Phase 2**). |

# Zusammenfassung: 2-Phasen-Commit

- Der transaktionsübergreifende Commit wird sichergestellt
- Die Konsistenz wird datenbankübergreifend garantiert
- Der 2-Phasen-Commit wird kommerziell angewendet
- Das Protokoll ist extrem aufwendig
  - da der Koordinator ständig die lokalen Transaktionen überwachen muss
  - da auch Netzausfälle mit einkalkuliert werden müssen
  - da Netzverzögerungen nicht sofort zum Abbruch der globalen Transaktionen führen sollen

# Objektorientierte Datenbanken

- Objektorientierte Datenbanken entstanden
  - in Folge zu den objektorientierten Programmiersprachen
  - ab 1990
  - als rein objektorientierte Datenbanken
    - Diese sind heute praktisch ohne Bedeutung
  - als Erweiterung der relationalen Datenbanken
    - Diese wurden in die SQL 3 und SQL 2003 Norm aufgenommen
- Die wichtigsten objektorientierten Datenbank sind:
  - Oracle
  - PostgreSQL

# Definition (objektorientierte Datenbank)

- Eine Datenbank heißt objektorientiert, wenn sie grundlegende objektorientierte Konzepte wie Objekte, Klassen, Methoden, Kapselung und Vererbung enthält und verwendet.

- Diese Definition ist sehr allgemein gehalten und bezieht damit die objektrelationalen Datenbanken voll mit ein.

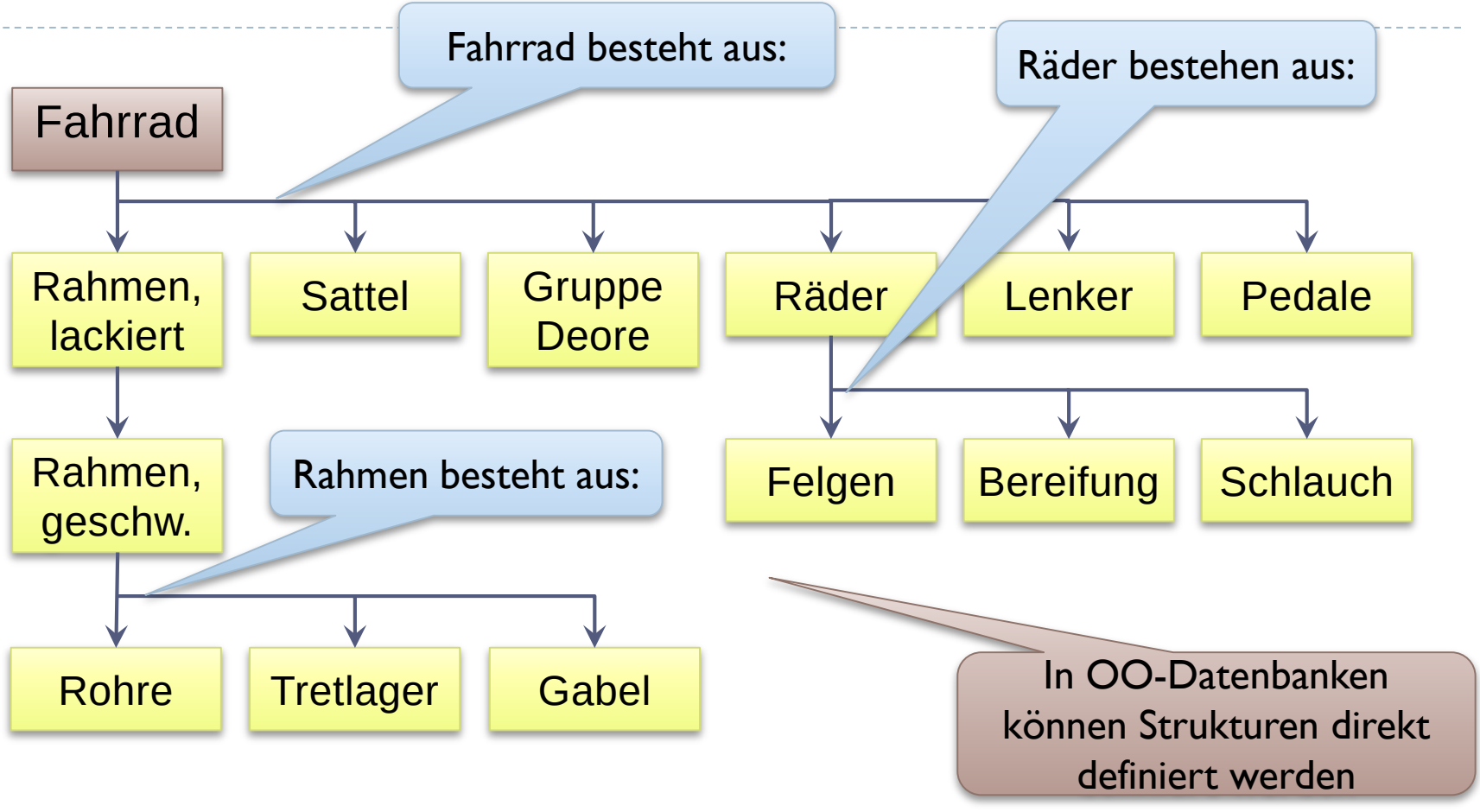
# Objektorientierte Konzepte (Überblick)

- **Objekt:** Einzelne Gegenstände und Entitäten, die sehr komplex sein können, z.B. ein bestimmtes Flugzeug
- **Klasse:** Ein Objekttyp, z.B. ein Airbus A300
- **Eigenschaft einer Klasse:** Eigenschaft eines Objekttyps, z.B. der Preis, das Gewicht
- **Methode einer Klasse:** Routine, die auf Objekte angewendet werden, z.B. StarteFlugzeuge, LandeFlugzeug
- **Kapselung:** Eigenschaften und Methoden sind nur für bestimmte Anwendungen erlaubt.
- **Vererbung:** Spezialisierung von Klassen. Aus einem Flugzeug wird beispielsweise ein Segel- oder Motorflugzeug abgeleitet

# Nachteile relationaler Datenbanken

| Nachteile | Auswirkung |
| --- | --- |
| Flache Struktur | Komplexe Objekte werden „flachgeklopft“; ihr Aufbau ist aus dem Design nicht mehr direkt ersichtlich |
| Keine Rekursion | Der Aufbau komplexer Objekte kann nur schwer nachvollzogen werden (Stücklistenproblem) |
| Viele Relationen | Häufige Joins auf zusammengehörige Relationen verlängern die Laufzeit |

# Vorteil OO-Datenbanken: Strukturen



# Objektrelationale Datenbanken

- Echte Erweiterung der relationalen Datenbanken
- Erweiterung um objektorientierte Konzepte
- Idee
  - Relationale Datenbanken sind weit verbreitet
  - Diese können weiter verwendet werden
  - Zusätzlich wird eine Erweiterung angeboten, die Schritt für Schritt eingesetzt werden kann
- Verbreitung:
  - Oracle, PostgreSQL

# Definition ($NF^2$-Normalform)

- **Eine Relation ist in der $NF^2$-Normalform, falls sie bis auf die Atomarität alle Bedingungen an eine Relation erfüllt.**

- Idee:
  - Ein Attribut kann aus komplexen Strukturen bestehen, z.B.
    - Aufzählung (Liste)
    - Struktur (Objekt)
    - Relation (eingebettete Relation)
  - Damit lassen sich sehr komplexe Strukturen in nicht normalisierten Relationen nachbilden

# Objektrelational in Oracle

- Objektorientierte Datenbanken benötigen Programmier-sprache
  - In Oracle: PL/SQL
- Oracle unterstützt:
  - Variable Felder
  - Objekte
  - Objekt-Sichten
  - Eingebettete Relationen
  - Objekt-Methoden (Prozeduren / Funktionen)

# Variable Felder in Oracle (1)

CREATE TYPE Typname AS
{ VARRAY | VARRYING ARRAY } ( Anzahl ) OF Datentyp

- Ein Feld wird erzeugt
- Die maximale Größe des Feldes ist anzugeben (Anzahl)
- Beispiel:
  - Feld TProdukt aus max. 50 Zeichenketten mit 30 Zeichen:

```
CREATE TYPE TProdukt AS VARRAY ( 50 ) OF CHAR ( 30 ) ;
```

# Nicht atomare Relation VerkaeuferProdukt

| VerkNr | VerkName | PLZ | VerkAdresse | Produktname | Umsatz |
|---|---|---|---|---|---|
| V1 | Meier | 80331 | München | Waschmaschine, Herd, Kühlschrank | 17000 |
| V2 | Schneider | 70173 | Stuttgart | Herd, Kühlschrank | 7000 |
| V3 | Müller | 50667 | Köln | Staubsauger | 1000 |

- **Realisierung mit Oracle:**

```
CREATE TABLE VerkaeuferProdukt
(     VerkNr       CHAR(4)        PRIMARY KEY ,
      VerkName     CHAR(20)       NOT NULL ,
      PLZ          CHAR(5) ,
      Adresse      CHAR(60) ,
      Produktname  TProdukt ,
      Umsatz       NUMERIC (10, 2)        ) ;
```

# Einfügen der ersten Zeile:

```
INSERT INTO VerkaeuferProdukt VALUES
(   'V1', 'Meier', '80331', 'München',
    TProdukt( 'Waschmaschine', 'Herd', 'Kühlschrank' ), 17000 );
```

- Das Feld kann nicht direkt eingegeben werden
- Es muss mit dem Objekttyp spezifiziert werden
- Die weiteren Zeilen werden entsprechend eingefügt
- Aufruf mittels:

```
SELECT * FROM VerkaeuferProdukt;
```

# Objekte in Oracle

```
CREATE [ OR REPLACE ] TYPE Typname AS OBJECT
(  Spalte   Datentyp  [ , … ] ,
   [ { MEMBER { Prozedurname | Funktionsname } }  [ , … ] ]  )
```

- Es werden zuerst Attribute (Spalten) definiert
- Dann folgt die Deklaration von Memberfunktionen und –prozeduren (Methoden)

# Objekttyp: TAdresse

- **Beispiel:**
  - Als Objekt wird eine Adresse mit PLZ, Ort und Straße definiert. Zusätzlich enthält das Objekt eine Methode Anschrift

```
CREATE OR REPLACE TYPE TAdresse AS OBJECT
(   Strasse      VARCHAR2 ( 30 ) ,
    PLZ          VARCHAR2 ( 5 ) ,
    Ort          VARCHAR2 ( 20 ) ,
    MEMBER FUNCTION Anschrift RETURN VARCHAR2    ) ;
/
```

3 Attribute: Strasse, PLZ, Ort

1 Methode (Funktion): Anschrift

In Oracle wichtig, wenn weitere Zeilen folgen

# Anwendung von TAdresse

- In Relation Lieferant (analog in Relation Kunde, Personal):

```
CREATE TABLE LieferantNeu
(   Nr          INTEGER             PRIMARY KEY ,
    Name        VARCHAR ( 30 )      NOT NULL ,
    Adresse     TAdresse ,
    Sperre      CHAR                ) ;
```

Neuer Datentyp

- Ausgabe aller Mitarbeiter mit Wohnort:

```
SELECT  Name, L.Adresse.Ort
FROM    LieferantNeu L;
```

Objekt Adresse mit Attribut Ort

Oracle benötigt einen Aliasnamen!

# Einfügen in LieferantNeu

- **Existiert die Relation Lieferant, so können alle Daten direkt übernommen werden:**

```
INSERT INTO LieferantNeu
SELECT Nr, Name, TAdresse( Strasse, PLZ, Substr(Ort, 1, 20) ),
       Sperre
FROM   Lieferant ;
```

… mit 3 Attributen

Objekt TAdresse …

Achtung! Ort ist CHAR(20), in Lieferant jedoch CHAR(25)!

- **Die Funktion Substr erzeugt ein passendes Attribut!**

# Objektsichten

- In rein relationalen Datenbanken können alternativ auch Objektsichten verwendet werden.
- Beispiel
  - Relation SLieferant als Objektsicht

```
CREATE VIEW SLieferant ( Nr, Name, Adresse, Sperre )  AS
    SELECT   Nr, Name, TAdresse( Strasse, PLZ, Ort ), Sperre
    FROM     Lieferant ;
```

Sicht verwendet Objekttyp!

- Die Zugriffe sind analog wie in LieferantNeu!
- Die Sicht ist änderbar!

# Probleme mit Objekten

- **Ausgabe in SQL Developer:**

  SELECT * FROM LieferantNeu;

| NR | NAME | ADRESSE | SPERRE |
|---|---|---|---|
| 1 | Firma Gerti Schmidtner | [BIKEOO.TADRESSE] | 0 |
| 2 | Rauch GmbH | [BIKEOO.TADRESSE] | 0 |
| 3 | Shimano GmbH | [BIKEOO.TADRESSE] | 0 |
| 4 | Suntour LTD | [BIKEOO.TADRESSE] | 0 |
| 5 | MSM GmbH | [BIKEOO.TADRESSE] | 0 |

- **Alternative 1:**

```
SELECT   Nr, Name, L.Adresse.Strasse, L.Adresse.PLZ,
         L.Adresse.Ort, Sperre
FROM     LieferantNeu L ;
```

Sehr umständlich

# Objektmethode Anschrift

- **Alternative 2:**
  - Objektmethode Anschrift verwenden
- **Definition der Methode in CREATE TYPE BODY:**

```
CREATE OR REPLACE TYPE BODY TAdresse AS
  MEMBER FUNCTION Anschrift RETURN VARCHAR2 IS
  Ausgabe  VARCHAR2(60);
  BEGIN
     Ausgabe := TRIM(Strasse) || ', ' || TRIM(PLZ) || '  ' || TRIM(Ort);
     RETURN Ausgabe;
  END;
END;
/
```

Ergebnistyp ist VARCHAR2

Lokale Variable Ausgabe

Konkatenieren von Strasse, PLZ und Ort

Ergebnisrückgabe

# Ausgabe von LieferantNeu

- **Elegante Ausgabe mittels der Methode Anschrift()**

Arbeitsblatt | Query Builder

```
SELECT Nr, Name, L.Adresse.Anschrift(), Sperre
FROM LieferantNeu L ;
```

Skriptausgabe | Abfrageergebnis

SQL | Alle Zeilen abgerufen:5 in 0 Sekunden

| | NR | NAME | L.ADRESSE.ANSCHRIFT() | SPERRE |
|---|---|---|---|---|
| 1 | 1 | Firma Gerti Schmidtner ... | Dr. Gesslerstr. 59, 93051 Regensburg | 0 |
| 2 | 2 | Rauch GmbH ... | Burgallee 23, 90403 Nürnberg | 0 |
| 3 | 3 | Shimano GmbH ... | Rosengasse 122, 51143 Köln | 0 |
| 4 | 4 | Suntour LTD ... | Meltonstreet 65, London | 0 |
| 5 | 5 | MSM GmbH ... | St-Rotteneckstr. 13, 93047 Regensburg | 0 |

# Eingebettete Relationen

- Bisher:
  - Ein Attribut kann eine Liste oder auch ein Objekt enthalten
  - Ein Attribut kann eine Liste aus Objekten enthalten!
  - Aber: Die maximale Größe der Liste ist beschränkt
- Neu:
  - Ein Attribut kann eine Relation enthalten
  - Eine Beschränkung der Größe der Relation existiert nicht
  - Diese Relation heißt:

**Eingebettete Relation**

# Eingebettete Relation am Beispiel

- Die Relation Auftrag enthält Auftragspositionen
- Die Positionen werden in die Relation Auftrag eingebettet
- Definition eines geeigneten Objekts TEinzelposten:

```
CREATE OR REPLACE TYPE TEinzelposten AS OBJECT
(   Artnr          INTEGER,
    Anzahl         INTEGER,
    Gesamtpreis    NUMERIC(10,2)     ) ;
```

Objekt TEinzelposten

- Definition der eingebetteten Relation:

ER_Einzelposten ist Relation vom Typ TEinzelposten

```
CREATE  TYPE  ER_Einzelposten  AS TABLE OF  TEinzelposten ;
```

# Erzeugen der Relation AuftragNeu

```
CREATE TABLE  AuftragNeu
(   AuftrNr        INTEGER   PRIMARY KEY,
    Datum          DATE,
    Einzelposten   ER_Einzelposten,
    Kundnr         INTEGER   REFERENCES Kunde,
    Persnr         INTEGER   REFERENCES Personal
                             ON DELETE SET NULL
)   NESTED TABLE Einzelposten
                   STORE AS  ER_Einzelposten_TABLE ;
```

Eingebettete Relation ER_Einzelposten

Angabe der Tabelle, in der die Daten der eingebetteten Relation gespeichert werden

# Einfügen von Auftrag 2 in AuftragNeu

- Auftrag 2 besitzt zwei Auftragspositionen
- Wir müssen wieder Cast-Operatoren verwenden
- ER_Einzelposten besteht aus Objekten (TEinzelposten):

```
INSERT INTO AuftragNeu VALUES
(   2,  DATE '2013-01-06',
    ER_Einzelposten (   TEinzelposten (100002, 3, 1950),
                        TEinzelposten (200001, 1, 400)      ),
    3,  5
) ;
```

Auftrnr, Datum

ER_Einzelposten, bestehend aus TEinzelposten

Artnr, Anzahl, Gesamtpreis als Objekt TEinzelposten

Kundnr, Persnr

# Vorteile objektrelationaler Datenbanken

- Die Relation AuftragNeu muss nicht künstlich in die Relationen Auftrag und Auftragsposten zerlegt werden
- Ein Fremdschlüssel (Auftrnr in Auftragsposten) entfällt
- Die interne Struktur des Auftrags bleibt erhalten
- Es ist kein Join erforderlich, um die Daten zu lesen
- Hohe Performance (da kein Join erforderlich)

# Nachteile objektrelationaler Datenbanken

- Der Fremdschlüssel Artnr (auf Relation Auftrag) kann nicht angegeben werden!
- Die Zugriffsbefehle sind sehr komplex
- Beispiel 1:

```
SELECT * FROM AuftragNeu ;
```

- Beispiel 2:

```
SELECT   Einzelposten
 FROM    AuftragNeu
WHERE    AuftrNr = 2 ;
```

**Einzelposten(Artnr, Anzahl, Gesamtpreis)**

ET_Einzelposten(TEinzelposten(100002, 3, 1950), TEinzelposten(200001, 1, 400))

# Der Operator THE

- Der Operator THE wandelt eine eingebettete Relation in eine „normale“ Relation um
- Beispiel:

Eingebettete Relation

```
SELECT  *
 FROM   THE (   SELECT  Einzelposten
                FROM    AuftragNeu
                WHERE AuftrNr = 2    ) ;
```

Ergebnis: Relation

| Artnr | Anzahl | Gesamtpreis |
|---|---|---|
| 100002 | 3 | 1950 |
| 200001 | 1 | 400 |

# Beispiele mit Operator THE

- **Einfügen eines weiteren Auftragspostens (2 Tretlager):**

```
INSERT INTO THE (  SELECT Einzelposten FROM AuftragNeu
                   WHERE AuftrNr = 2 )
VALUES ( 500013, 2, 60 ) ;
```

„Normale“ Relation dank THE-Operator

- **Ausgabe aller Positionen zu Auftrag 2, die mehr als 100 Euro kosten:**

„Normale“ Relation dank THE-Operator

```
SELECT  *
 FROM   THE (   SELECT  Einzelposten
                FROM    AuftragNeu
                WHERE   AuftrNr = 2   )
 WHERE  Gesamtpreis > 100 ;
```

# Cast-Operator MULTISET

- Der Operator THE wandelt eine eingebettete Relation in eine Relation um
- Um alle Auftragspositionen, die mehr als 100 Euro Umfang haben, auszugeben, sind viele THE-Operatoren erforderlich
- Zur gemeinsamen Ausgabe müssen diese wieder in eine Objektrelation zurückverwandelt werden.
- Dies leistet der Cast-Operator Multiset:

Relation

Ergebnis: eingebettete Relation

CAST ( MULTISET ( Unterabfrage ) AS Objekttyp )

# Beispiel zu MULTICAST

- Ausgabe der Auftragsnummern mit den dazugehörigen Auftragspositionen über 100 Euro:

```
SELECT A.AuftrNr,
       CAST(MULTISET(
            SELECT *
            FROM THE (  SELECT Einzelposten
                        FROM   AuftragNeu
                        WHERE AuftrNr = A.AuftrNr )
            WHERE Gesamtpreis > 100
                  )   AS ER_Einzelposten )
 FROM AuftragNeu A;
```

Liefert je A.Auftrnr alle Auftragspositionen als Relation

davon nur die Positionen mit Gesamtpreis > 100

Rückumwandeln in eingebettete Relation, um diese Positionen mit der Auftrnr anzuzeigen

# Einfügen mittels MULTISET

- Übernahme aller Daten aus Auftrag und Auftragsposten:

```
INSERT INTO AuftragNeu (  Auftrnr, Datum, Einzelposten,
                          Kundnr, Persnr)
 SELECT    AuftrNr, Datum, NULL, Kundnr, Persnr
 FROM      Auftrag;

UPDATE Auftragneu
SET Einzelposten = CAST(MULTISET(
                   SELECT  Teilenr, Anzahl, Gesamtpreis
                   FROM    Auftragsposten
                   WHERE  AuftrNr = Auftragneu.AuftrNr
                                  )  AS ER_Einzelposten ) ;
```

Alle Daten außer Einzelposten aus Relation Auftrag übernehmen

Alle Positionen werden je Auftrag in Attribut Einzelposten übernommen

# Zusammenfassung

- Verteilte Datenbanken
  - Verteilte Datenbanken erhöhen die Zugriffsgeschwindigkeit und die Verfügbarkeit auf Kosten der sofortigen Konsistenz
  - ACID wird durch BASE ersetzt!
  - Zwei-Phasen-Commit garantiert sofortige Konsistenz, aber sehr aufwendig
- Objektrelationale Datenbanken
  - Beliebig komplexe Objekte sind direkt abbildbar
  - Dies erfordert jedoch eine sehr aufwendige Programmierung