Zusammenfassung

**Lernziele**

* Datenbanksysteme (DBMS) programmieren können (Stored Procedures, Triggers)
* Programmierschnittstellen für die Applikationsentwicklung vertiefter anwenden (JDBC)
* Werkzeuge zur DB-Anbindung an Applikationen einordnen können
* Grundlagen von objekt- relationalen sowie von objekt-orientierten DBMS kennen
* Semistrukturierte DBMS mit XML kennen
* Information Retrieval sowie Data Warehouse/OLAP einordnen können

**Unterlagen / Bücher**

* Elmasri & Navathe (2009): Fundamentals of Database Systems, Verlag AddisonWesley
* Ramakrishnan & Gehrke (2002): Data Base Management Systems, 3rd Ed., ISBN 0-07-115110-9
* Kemper & Eickler, Datenbanksysteme, Verlag Oldenbourg, ISBN 3-486-25706, 7. Auflage mit Übungsbuch
* Heuer & Saake (2000): Datenbanken: Konzepte und Sprachen, mitp-Verlag, 2. Auflage, 3-8266-0619-1

**Lerninhalte**

* Stored Procedures (PL/SQL)
* Cursors, Triggers, Udateable Views
* Backup und Recovery
* Indexe
* Objektrelationale Datenbanksysteme
* Objektorientierte Datenbanksysteme
* Semistrukturierte Daten (XML) in relationalen Datenbanksystemen
* Daten-Replikation und Synchronisierung
* In-Memory-Datenbanksysteme
* Datenstrukturen in Datenbanksystemen
* NoSQL-Datenbanksysteme
* Verteilte und Cloud Datenbanksysteme

Übersicht des Modules

[OR-Mapping (Einführung) 6](#_Toc477278192)

[Ideen und Lösungsansätze 6](#_Toc477278193)

[Semantic Gap in Persistence 6](#_Toc477278194)

[Object-Relation Mapper 6](#_Toc477278195)

[Technologien 6](#_Toc477278196)

[Custom Mappers 6](#_Toc477278197)

[Im Überblick 6](#_Toc477278198)

[Java Persistence API (JPA) 7](#_Toc477278199)

[Layers 7](#_Toc477278200)

[Provider Auswahl 7](#_Toc477278201)

[Grundgedanke 7](#_Toc477278202)

[JPA Kurz-Tour 7](#_Toc477278203)

[Mapping Regeln 8](#_Toc477278204)

[Regeln für Entites 8](#_Toc477278205)

[Table & Column Mapping 9](#_Toc477278206)

[Einfache Mapping Regeln 9](#_Toc477278207)

[Access Types 9](#_Toc477278208)

[Datentypen 10](#_Toc477278209)

[Mapping in XML 10](#_Toc477278210)

[Entity Relations (Beziehungen) 10](#_Toc477278211)

[1:1 Beziehung 11](#_Toc477278212)

[Bidirektionale 1:1 Beziehung 11](#_Toc477278213)

[Inverse 1:1 Beziehung 11](#_Toc477278214)

[N:1 Beziehung 11](#_Toc477278215)

[Bidirektionale N:1 Beziehung 11](#_Toc477278216)

[1:N Beziehung 12](#_Toc477278217)

[N:M Beziehung 12](#_Toc477278218)

[Bidirektionale N:M Beziehung 12](#_Toc477278219)

[Laden der Relations 12](#_Toc477278220)

[Ladestrategien 12](#_Toc477278221)

[«OR Mapping» mit JPA (Forsetzung) 13](#_Toc477278222)

[Wie kommen Entites und Daten zusammen? 13](#_Toc477278223)

[OR-Mapping Varianten 13](#_Toc477278224)

[JPA-Architektur (Begriffe) 13](#_Toc477278225)

[JPA Persistence Unit 13](#_Toc477278226)

[JPA Persistence Context 13](#_Toc477278227)

[Entity State 14](#_Toc477278228)

[Operationen auf den Entities 14](#_Toc477278229)

[Persistencte Context (=Sesssion) 14](#_Toc477278230)

[Object Identity 14](#_Toc477278231)

[Entity-Identität 15](#_Toc477278232)

[Generierungstyp Identity 15](#_Toc477278233)

[Generierungstyp Sequence 15](#_Toc477278234)

[Generierungstyp Table 15](#_Toc477278235)

[Transitive Persistenz 15](#_Toc477278236)

[Explizite (De-)Allokation 15](#_Toc477278237)

[Kaskadierte Persistenz 15](#_Toc477278238)

[Kaskadiertes Löschen 16](#_Toc477278239)

[Änderungen an Relations 16](#_Toc477278240)

[Relations mit Abhängigkeiten 16](#_Toc477278241)

[Änderungen an Relations 16](#_Toc477278242)

[Inkonsistente Bidirektionalität 16](#_Toc477278243)

[Bidirectional Sync 16](#_Toc477278244)

[JPA Transaktionen 17](#_Toc477278245)

[Transaktionsende 17](#_Toc477278246)

[Transaction Isolation 17](#_Toc477278247)

[Explizites Locking 17](#_Toc477278248)

[Optimistic concurrency 17](#_Toc477278249)

[Zusätzliches Locking mit JPA Entity Locking 18](#_Toc477278250)

[Kontrolle über Verhalten und Zustände der Entities 19](#_Toc477278251)

[Merge 19](#_Toc477278252)

[Refresh 19](#_Toc477278253)

[Beispiel 19](#_Toc477278254)

[Vererbung 19](#_Toc477278255)

[Single Table Mapping 20](#_Toc477278256)

[Joined Table Mapping 20](#_Toc477278257)

[Table Per Class Mapping 20](#_Toc477278258)

[Diskussion 20](#_Toc477278259)

[Anfragen in JPA mit JPQL 21](#_Toc477278260)

[Beispiele 21](#_Toc477278261)

[Query Parameters 21](#_Toc477278262)

[Parameter setzen 21](#_Toc477278263)

[Dynamic Queries 21](#_Toc477278264)

[Named Queries 21](#_Toc477278265)

[Criteria API 22](#_Toc477278266)

[Weitere Beispiele 22](#_Toc477278267)

[JPA Diskussion und Implementationen 22](#_Toc477278268)

[Fazit 22](#_Toc477278269)

[Stored Procedures (SP) 23](#_Toc477278270)

[Was sind Stored Procedures? 23](#_Toc477278271)

[Warum Stored Procedures? 23](#_Toc477278272)

[Wann keine Stored Procedures? 23](#_Toc477278273)

[Einführung 23](#_Toc477278274)

[PL/SQL 23](#_Toc477278275)

[Syntax 23](#_Toc477278276)

[Compiler 23](#_Toc477278277)

[PL/pqSQL - die Sprache 24](#_Toc477278278)

[Basis 24](#_Toc477278279)

[Deklarationen 24](#_Toc477278280)

[Block und Kommentare 24](#_Toc477278281)

[Conditoinals und Loops 25](#_Toc477278282)

[Exceptions 25](#_Toc477278283)

[Beispiele 26](#_Toc477278284)

[SELECT…INTO 26](#_Toc477278285)

[INSERT INTO 26](#_Toc477278286)

[UPDATE 26](#_Toc477278287)

[Function that returns void = SP 26](#_Toc477278288)

[Function that returns a scalar 26](#_Toc477278289)

[Function that returns a table 26](#_Toc477278290)

[SET Returning Functions 26](#_Toc477278291)

[Stored Procedures in SQL und Python (PostgeSQL) 27](#_Toc477278292)

[SP mit SQL (PostgreSQL) 27](#_Toc477278293)

[PL/Python (PostgreSQL) 27](#_Toc477278294)

[Beispiel File Exists 27](#_Toc477278295)

[Gemeinsamkeiten Stored Procedures 27](#_Toc477278296)

[Cursor 28](#_Toc477278297)

[Verarbeitung 28](#_Toc477278298)

[Prinzip 28](#_Toc477278299)

[Beispiel 29](#_Toc477278300)

[Cursor mit Parametern 29](#_Toc477278301)

[Cursor for Update 29](#_Toc477278302)

[Vergleich Prozedur / Funktion 29](#_Toc477278303)

[Triggers 30](#_Toc477278304)

[Event Model 30](#_Toc477278305)

[Syntax 30](#_Toc477278306)

[PL/pgSQL 31](#_Toc477278307)

[Trigger-Funktion 31](#_Toc477278308)

[Trigger-Funktionsvariabeln 31](#_Toc477278309)

[Beispiel mit Funktion 31](#_Toc477278310)

[Datenzugriff 31](#_Toc477278311)

[Weitere Beispiele 31](#_Toc477278312)

[Beispiel mit DELETE 31](#_Toc477278313)

[Beispiel 2 32](#_Toc477278314)

[Abgeleitete Attribute 32](#_Toc477278315)

[Auditing mit Triggers 32](#_Toc477278316)

[Ausführung von Triggers 33](#_Toc477278317)

[Diskussion Triggers 33](#_Toc477278318)

[Updatable Views und weitere Views 33](#_Toc477278319)

[Updatable Views 33](#_Toc477278320)

[Beispiele 33](#_Toc477278321)

[Instead-of Triggers 34](#_Toc477278322)

[Materialized Views 34](#_Toc477278323)

[Temporäre Tabellen 35](#_Toc477278324)

[Realisierung von Sicherheit mit Views, Stored Procedures und Triggers 35](#_Toc477278325)

[Zugriffsschutz mit Views 35](#_Toc477278326)

[Row-Level Security 35](#_Toc477278327)

[Zugriffschutz mit Stored Procedures 36](#_Toc477278328)

[Beispiele 36](#_Toc477278329)

[Auditing mit Triggers 36](#_Toc477278330)

[Vorlesung Woche 5 Fehler! Textmarke nicht definiert.](#_Toc477278331)

# OR-Mapping (Einführung)

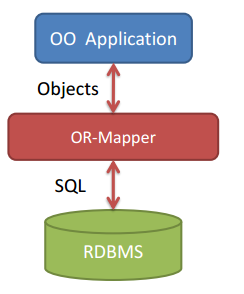
## Ideen und Lösungsansätze

In Software-Programmen möchte man Objekte manipulieren (CRUD), insbesondere speichern (=persistieren) und wieder holen (=abfragen).

Mit reinem JDBC muss ich jede Spalte auf Java Objekte ummappen. Ein OR Mapper soll nun das Objektmodell und das Datenbankmodell verbinden. Dabei möchte ich nichts mit dem Mapping vom Objektmodell zum Datenbankmodell zu tun haben.

## Semantic Gap in Persistence

OR Mapper sollen diesen Semantic Gap zwischen den beiden Modellen schliessen. Die Programme sind meist objekt-orientiert, während die Datenbanken meist relational sind.

Es soll eine «Nahtlose» Abbildung von und zu der darunterliegenden DB sein.

**Funktionen**

* Laden von relationalen Daten als Verbund von Objekten (Joins über mehrere Tabellen)
* Speichern von Änderungen an persistenten Objekten

## Object-Relation Mapper

### Technologien

**Unter Java** Java Persistence API (JPA), Java Data Objects (JDO), Hibernate

**Unter .NET** Entity Framework (EF), NHibternate

### Custom Mappers

In gewissen Fällen sinnvoll

* Bei besonders hoher Abbildungskomplexität (Mappings auf SPs)
* Bei mehreren Datensilos (SQL, NoSQL, Streams, Filesystems)
* Evtl. höhere Performance
* Evtl. besondere Skalierung
* Besondere Transaktionsverwaltung

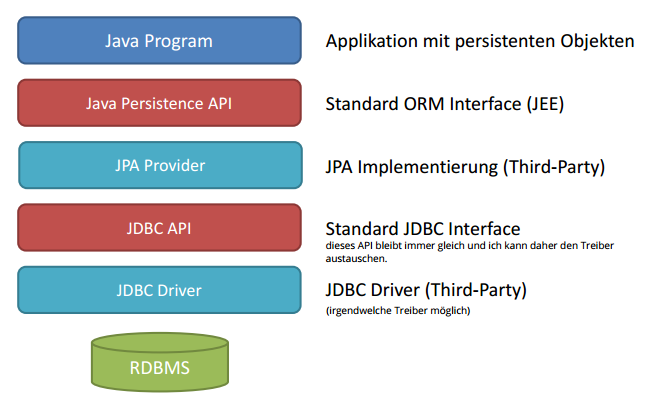
Custom Mappers haben natürlich einen hohen Arbeitsaufwand und die Komplexität ist auch vorhanden.

## Im Überblick

Das **Java Persistence API** (JPA) ist das Standard OR Mapping zu relationalen Datenbanken. Es ist Teil von Enterprise Java Beans, läuft aber auch ohne JEE und ohne JEE Application Server. Es ist nur eine Spezifikation. Diverse Implementierungen vorhanden. Beeinflusst wird es durch Hibernate und Java Data Object.  
**Java Data Object** (sozusagen tod), Standard vor JPA

# Java Persistence API (JPA)

## Layers



## Provider Auswahl

Es gibt verschiedene JPA Implementierungen. Dazu zählen EclipseLink, Hibernate, Apache OpenJPA, DataNucleues,….. In den Beispielen in dieser Vorlesung wird EclipseLink verwendet.

## Grundgedanke

**Entities** als persistente Objekte. Dabei sollen POJO («Plain Old Java Objekte») verwendet werden, also ganz normale Java Objekte. Mit einem ID-Feld für die Abbildung auf den Primarschlüssel in der Datenbank. Zudem ein Parameter losen Konstruktor.

Die **Abbildung** zur Datenbank wird mittels Annotations im Code (nicht ganz POJO) realisiert. Alternativ kann dies auch in separaten Mapping-Files deklariert werden.

Die **Anbindung** an die Datenbank findet im Persistence.XML Konfigurationsfile statt.

## JPA Kurz-Tour

|  |  |
| --- | --- |
| **Entity Klasse** | **Abbildung auf die DB** |
| **JPA Konfiguration** File «persistence.xml » im Folder « META-INF» |  |
| **Laden/Suchen von Entities** | **Explizites Einfügen von Entities** |
| **Update von Entities** | **Explizites Löschen von Entities**  Zum Löschen muss das Objekt zuerst in den Cache geladen werden und kann dann erst gelöscht werden. Mit commit wird dann die Löschoperation erst effektiv ausgeführt wird. |

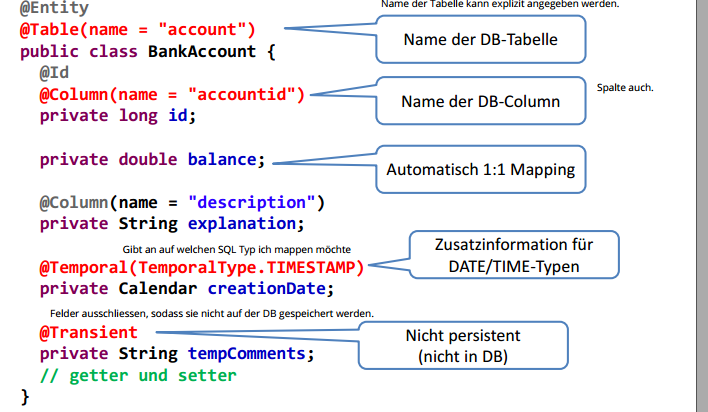
## Mapping Regeln

### Regeln für Entites

Eine Entity ist eine Klasse mit der Annotation @Entity. Diese Klasse kann erben, vererben, kann Interface implementieren oder auch abstrakt sein. Von dieser Seite her gibt es also keine Einschränkungen. Es gibt aber Einschränkungen bezogen auch den Inhalt der Klasse.

* Es muss ein Public oder Protected Konstruktor ohne Argumente vorhanden sein.
* Es ist eine Primary Key Angaben erforderlich (Annotation @id)
* Klasse darf nicht final sein, auch auch keine final Flieds oder Methoden haben
* Die Fileds sollen private oder protected sein und nur über getter/Setter zugreifbar.

### Table & Column Mapping

Mit @Table kann der Name der Tabelle explizit angegeben werden.

Mit einer Spalte ist dies über @Column möglich.

Mit @Temportal kann ich den entsprechenden SQL-Typ angeben.

Über @Transient kann ich Felder ausschliessen.

### Einfache Mapping Regeln

Standardmässig findet das Mapping mit den gleiche Namen statt. Für die Spalten und die Tabellen kann dies explizit angegeben werden. Es ist Case Insensitiv.

Per Default werden alle Attribute in das Mapping einbezogen. Ausnahmen sind die entsprechenden Annotationen.

Bei Calendar/Date Java-Typ braucht es Zusatzinformationen. Dafür lässt sich die Temporal Annotation verwenden. Eine Abbildung ist auch Date, Time oder Timestamp möglich. Aber aufpassen, die Unterstützung ist nicht immer gleich.

### Access Types

|  |  |
| --- | --- |
| **Field Access** DB-Attribute direkt in Fields abbilden | **Property Access** DB-Attribute über Getter/Setter abbilden |

Entweder macht man Field oder Property Access pro Klasse. Die Attribut-Annotation bestimmt den Access Type. Mindestens @Id Annotation ist pro Klasse nötig.

Beim Propery Access die Attribute Annotation beim Getter angeben. Alle Properties müssen Getter und Setter haben. Statt getXYZ() ist auch isXYZ() möglich (passend für den Boolean)

### Datentypen

Es werden folgende **Typen** beim Mapping **unterstützt**:  
Primitive Typen und deren Wrapper-Typen, String, Enums, Byte und Character Array, Date, Calendar und beliebige Serializierbare Klassen (welche nach BLOB abgebildet werden).

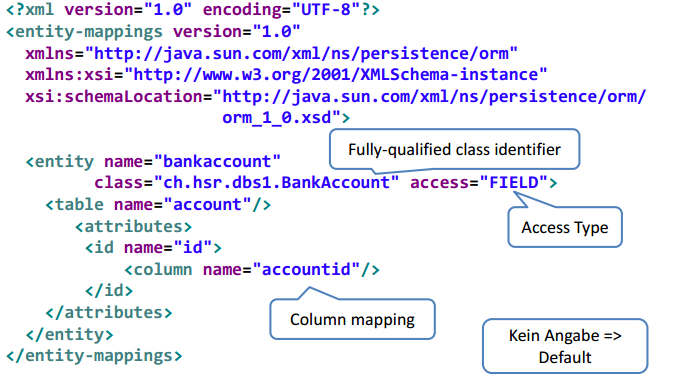
Zu den **unterstützten Relationen** zählen alle Referenzen auf Instanzen mit Entity Klassen. Darunter Collection<>, Set<>,List<> und Maps<> Entities.

Es ist auch eine explizite Detail-Angabe vom DB-Typ möglich. Nützlich für automatische Schema-Erzeugung von JPA.  


### Mapping in XML

Eine Alternative zu den Annotationen in den Java Klassen. Dazu findet die Abbildung in orm.xml im Folder META-INF statt. Die Prioritäten des Mappings sind wie folgt festgelegt.

1. XML-File orm.xml
2. Entity-Annotationen
3. Default-Mapping



## Entity Relations (Beziehungen)

Die Beziehungen können Unidirektional oder bidirektional sein. Es existieren verschiedene Kardinalitäten. In der DB als Tabellen mit FK-Schlüssel Beziehungen. Es ist eine Explizite Annotation oder XML-Mapping nötig.

### 1:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### Bidirektionale 1:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### Inverse 1:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### N:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### Bidirektionale N:1 Beziehung

|  |  |
| --- | --- |
|  |  |

### 1:N Beziehung

|  |  |
| --- | --- |
|  |  |

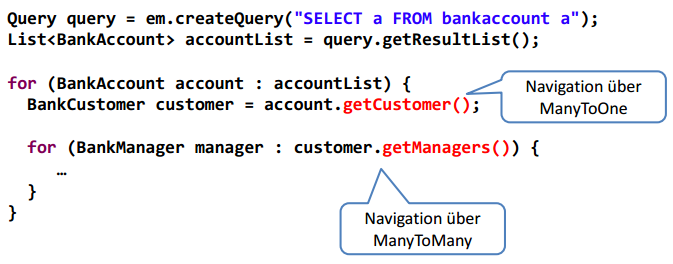
### N:M Beziehung

|  |  |
| --- | --- |
|  |  |

### Bidirektionale N:M Beziehung

|  |  |
| --- | --- |
|  |  |

### Laden der Relations

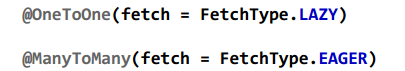
Die Normale Travesierung des Objektgraphen. Objekte laden, welche unter Abhängigkeit von einander stehen.  


### Ladestrategien

**Eager Loading**  
Target Entity wird direkt mit der Beziehung geladen. Default bei @OneToOne und @ManyToOne.

**Lazy Loading**  
Laden der Entity beim ersten Zugriff auf die Beziehung. Default bei @OneToMany und @ManyToMany.

Die Ladestrategien können explizit mit Annotationen gesteuert werden. Es ist ein bisschen doof, da man es bei jeder Beziehung wieder angeben muss. Es gibt andere Entities Frameworks, welche das besser machen.



# «OR Mapping» mit JPA (Forsetzung)

## Wie kommen Entites und Daten zusammen?

**Top down**  Erstelle Business-Modell und erzeuge DB-Schema

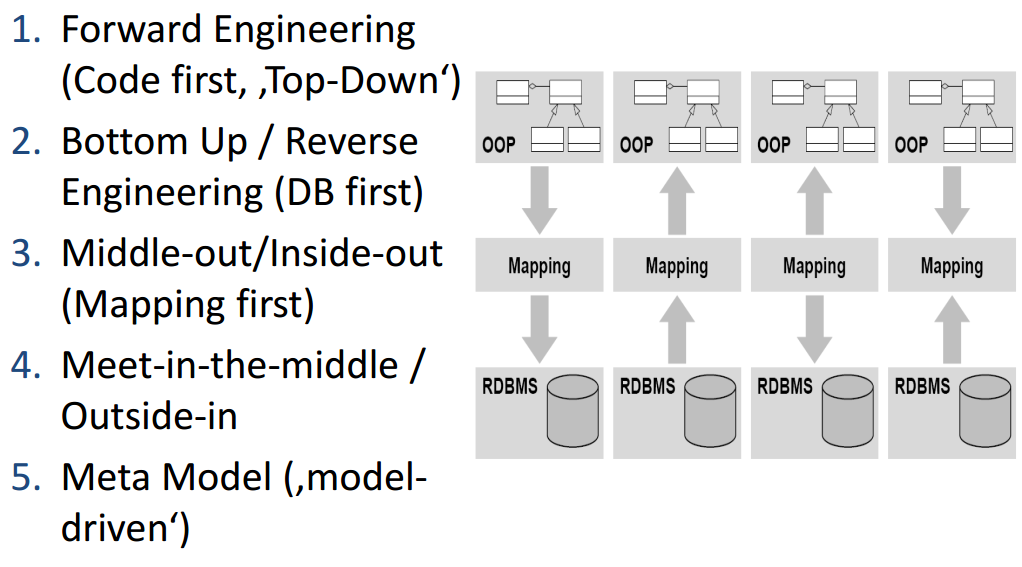
**Bottom up** DB-Schema existiert, erzeuge daraus Business-Modell…

**Middle out** Erstelle Metadaten und generiere Java und DB-Schema

**Meet in the middle** Business-Modell und DB-Schema existieren bereits. Erstelle Metadaten.

**Model Driven** ist für uns nicht wichtig, auch Meta Model genannt.

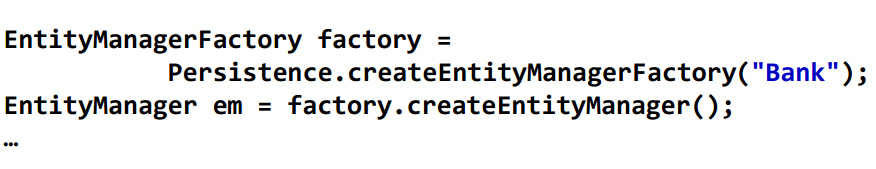
### OR-Mapping Varianten



## JPA-Architektur (Begriffe)

### JPA Persistence Unit

Die Persistence Unit ist die Menge der Entity Klassen und deren Mapping. Bestimmt durch den JPA Provider und die DB-Anbindung. Die EntitiyManagerFactory verwaltet eine PersistenceUnit.



### JPA Persistence Context

Der Persistence Context verwaltet eine Menge von Entity Instanzen zur Laufzeit. Ist eine Entity Instanz managed, dann ist Sie im Persistence Context. Ist eine Entity new oder detached gehört Sie zu keinem Kontext. Der Persistence Context definiert zudem die transaktionelle Session.

Der Enity Manger verwaltet den Persistence Kontext und bietet Lifecycle-Operationen für Entity Instanzen an.

### Entity State

**New** – Entity Instanzen haben keine Persistenz Identity und sind keinem Persistenz Kontext zugeordnet.

**Managed** – Entity Instanzen haben eine Persistenz Identity und sind einem Persistenz Kontext zugeordnet.

**Detached** - Entity Instanzen eine Persistenz Identity und sind aktuell keinem Persistenz Kontext zugeordnet.

**Removed** - Entity Instanzen eine Persistenz Identity und sind einem Persistenz Kontext zugeordnet und sind geschedulted für die Löschung vom Datastore.

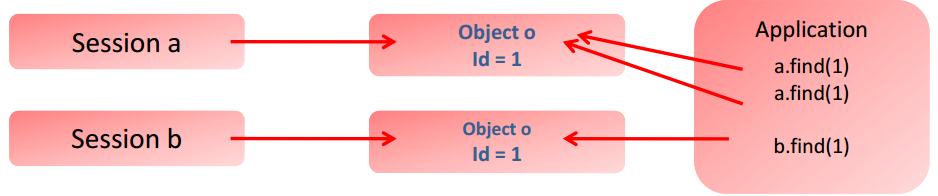
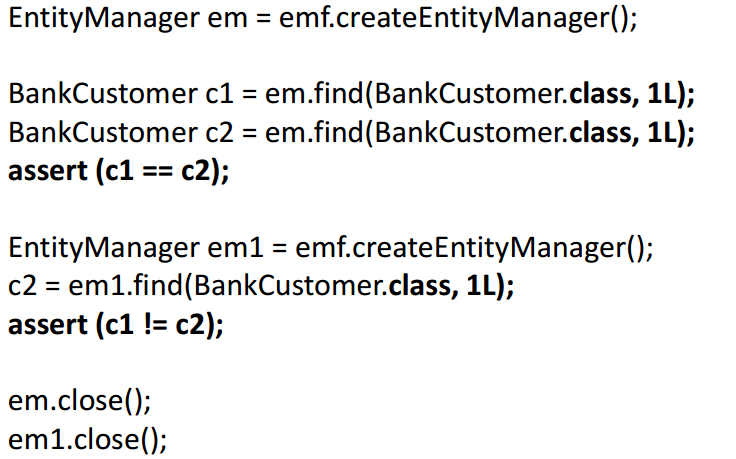
### Operationen auf den Entities

### Persistencte Context (=Sesssion)

Sie werden via dem EntityMangerFactory erzeugt. Der PC kapselt die DB-Session und die Transaktionen und verwaltet einen Cache. Er unterstützt CRUD-Operationen auf Entities. Wenn ein Objekt geladen wird und sich bereits im Cache befindet, dann wird eine Referenz auf den Cache geliefert. Zudem findet Change-Tracking statt. Änderungen auf den Objekten im Cache werden registriert. Geänderte Werte werden beim Commit mit der DB synchronisiert. Beim Close() der EntityManager-Instanz wird der Cache invalidiert.

### Object Identity

Die Session (= Instanz eines EntityMangers = Persistence Context) übersetzt das DB-Konzept des Primärschlüssels in das Konzept von Objekt-Instanzen. Die Session verwaltet daher die Objekt-Identität. Objekte werden über ihre ID im Cache verwaltet. Beim ersten Zugriff auf ein Objekt, wird dessen Zustand von der DB geladen. Bei einem weiteren Zugriff auf das Objekt, wird die Kopie im Cache zurückgegeben. Persistente Objekte werden über ihre Identität identifiziert.

## Entity-Identität

Findet über die Annotation @ID statt. Für jede Entity gibt es ein ID-Property. Es bildet den Primary Key in die DB ab. Die Generierung der Identity findet mit der Annotation @GeneratedValue bei ID-Property statt. Generiert wird sind dann durch die JPA-Instanz beim INSERT neuer ROWS in die DB.

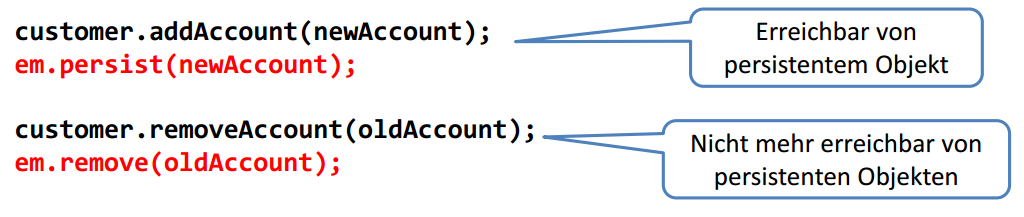
Für die Generierung der ID gibt es verschiedene Strategien. AUTO, IDENTITY, SEQUENCE und TABLE.

|  |  |
| --- | --- |
| Generierungstyp Identity | Generierungstyp Sequence |
| Generierungstyp Table |  |

### Transitive Persistenz

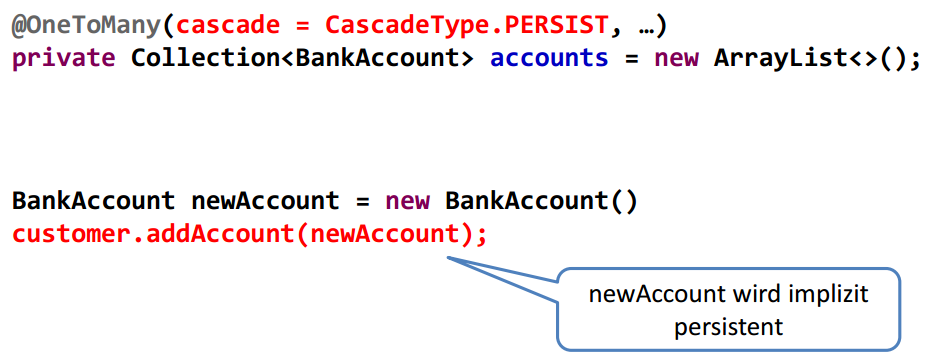
Alle von persistentem Objekt erreichbaren Objekte sollen wiederum persistent sein. Alle anderen Objekte müssen nicht persistent sein. In JPA gibt es keine automatische transitive Persistenz. Entweder explizit in der DB allozieren /deallozieren mit EntityManger.persist(), EntityManger.remove() oder implizit via kaskadierte Relations (Kaskade-Angabe bei Relation Annotation).

### Explizite (De-)Allokation



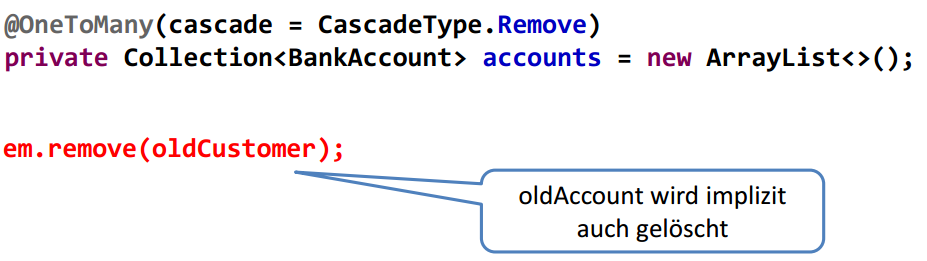
### Kaskadierte Persistenz

Mit der Angabe CascadeType.Persist werden die referenzierten Objekte automatisch persistent.



### Kaskadiertes Löschen

Mit CascadeType.Remove wird beim Löschen des Holder-Objekts auch das referenzierte Objekt entfernt.

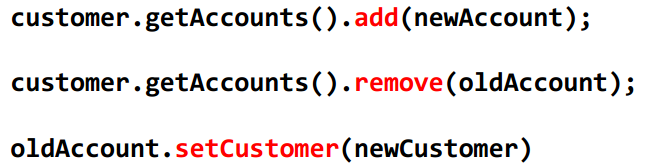


## Änderungen an Relations

### Relations mit Abhängigkeiten

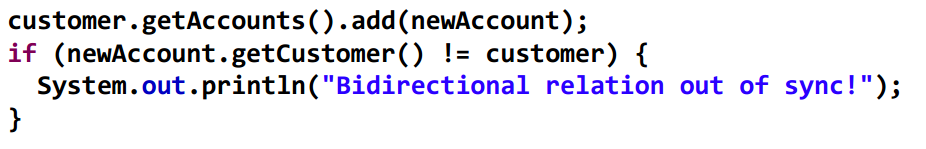
|  |  |
| --- | --- |
| **Aggregation** | **Komposition** |

### Änderungen an Relations

Die Beziehungen lassen sich normal zugreifen. Bei Zugriff findet Eager vs. Lazy Load Verhalten statt. Die Änderungen werden dann bei Commit in die Datenbank gespeichert.

### Inkonsistente Bidirektionalität

Bei den bidirektionalen Relationen muss man aufpassen. Die Änderungen werden durch das JPA nicht synchronisiert und das trotz der Angabe von @OneToMany(mappedBy=»customer»). Es ist nur nach dem Laden konsistent.

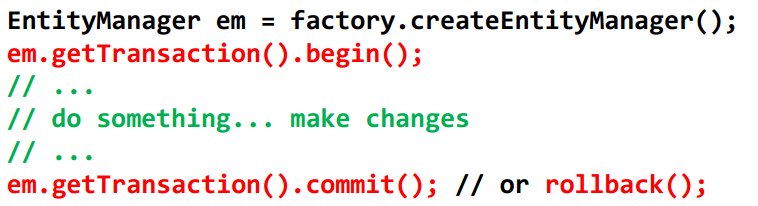


### Bidirectional Sync

|  |  |
| --- | --- |
|  |  |

## JPA Transaktionen

Es sind auch Änderungen in expliziten Transaktionen (gegenseitiger Ausschluss) möglich.



### Transaktionsende

Nach dem **Commit** werden die Änderungen in die DB geschrieben. Die Entity-Zustände werden nicht von der DB refresehed.

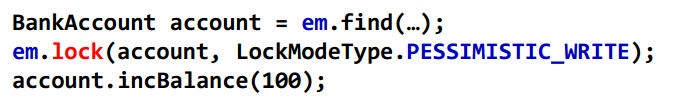
Nach dem **Rollback** werden die Änderungen nicht in die DB gespeichert. Die Entites sind detachated (Sie behalten den Zustand).

### Transaction Isolation

Das JPA basiert auf dem READ COMMITTED Isolation Level. Je nach JPA Provider sind auch andere Levels konfigurierbar. Zum Beispiel bei HIBERNATE im persistence.xml.

### Explizites Locking

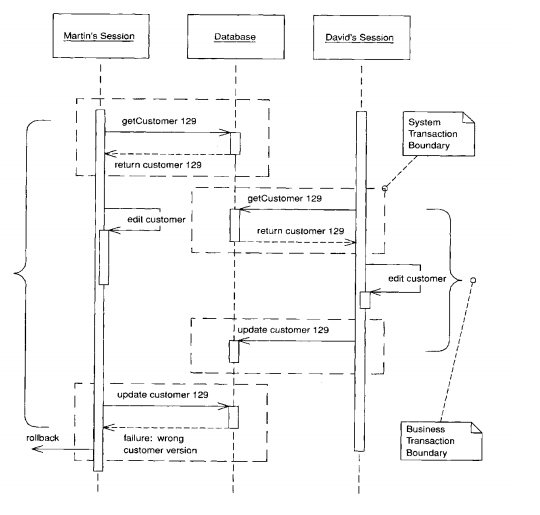
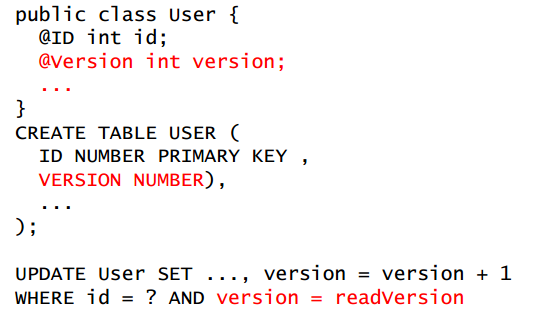
Die Methode lock() beim EntityManager soll Dirty Reads und List Updates ausschliessen.

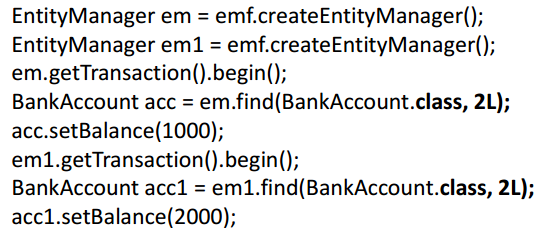
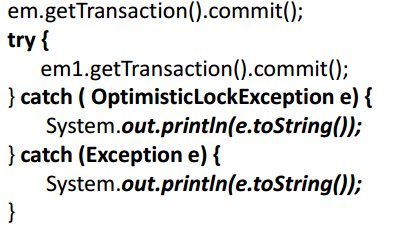


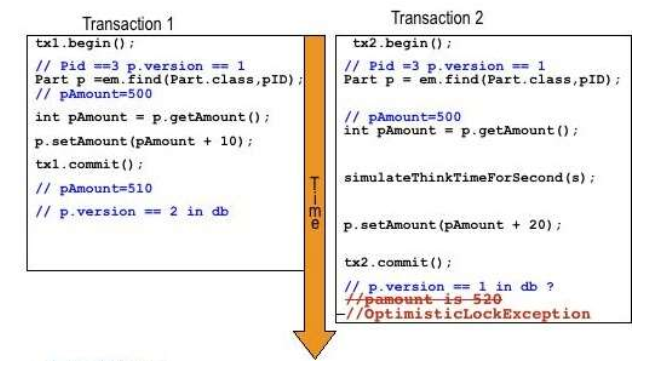
### Optimistic concurrency

« „Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction“.

**Optimistic**: Transaktionen laufen unbehindert an. Beim Abschliessen wird in einer Validierungsphase überprüft, ob Konflikte aufgetreten sind und gegebenenfalls die Transaktion zurückgesetzt.

Demo  
 



### Zusätzliches Locking mit JPA Entity Locking

Mit JPA 2.0 ist es möglich die Entities zusätzlich zu locken. Dies erlaubt es auch an die Art und Weise des Lockings anzugeben. Bei den Art und Weisen wird zwischen OPTIMISTIC\_FORCE\_INCREMENT und OPTIMISTIC unterschieden.

#### OPTIMISTIC

Der typische Anwendungsfall für den OPTIMISTIC-Sperrmodus ist, wo eine Entität eine intrinsische Abhängigkeit von einer oder mehreren Entitäten hat, um Konsistenz zu gewährleisten, zum Beispiel, wenn es eine Beziehung zwischen zwei Entitäten gibt.

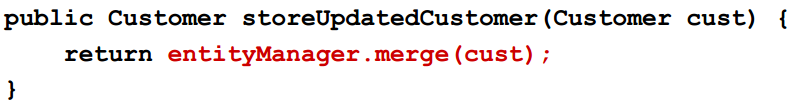
#### OPTIMISTIC\_FORCE\_INCREMENT

OPTIMISTIC\_FORCE\_INCREMENT-Sperrmodus verursacht einen optimistischen Sperrfehler, wenn eine andere Transaktion versucht, die gesperrte Entität zu ändern. Die gemeinsame Verwendung für diese Sperre ist es, die Konsistenz zwischen Entitäten in einer Beziehung zu gewährleisten.

## Kontrolle über Verhalten und Zustände der Entities

### Merge

Verändertes Objekt wird in den Persistenzkontext eingefügt. Zuerst wird überprüft, ob eine Instanz mit derselben Id im Cache ist; falls nicht wird eine Instanz erzeugt und deren Zustand von der DB geladen. Diese Instanz wird mit den Änderungen gemergt. Merge liefert das persistente Objekt zurück.

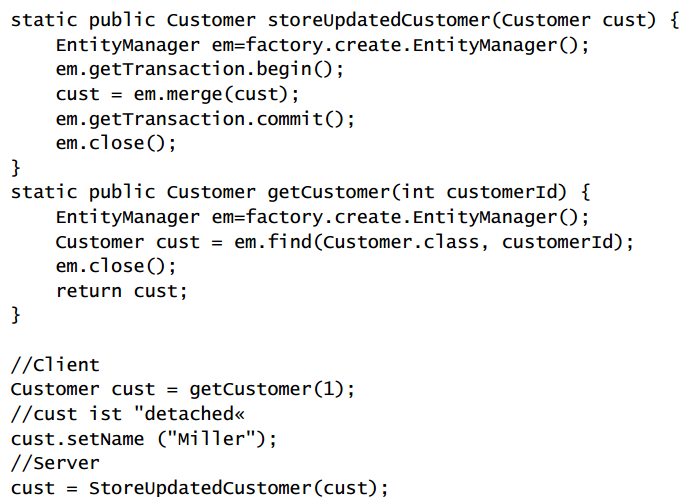


### Refresh

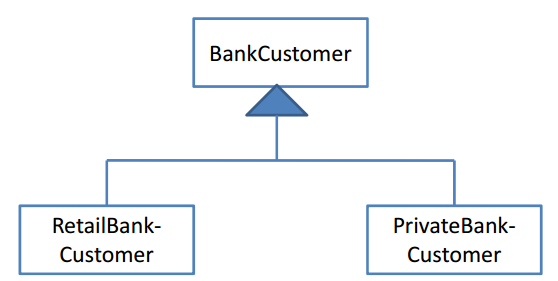
Der Zustand wird neu von der DB geladen.



### Beispiel



## Vererbung

Es gibt drei unterschiedliche Arten eine Vererbung zu implementieren.

Einerseits über Single Table, über Joined Tables oder mit Table Per Class.

|  |  |
| --- | --- |
| Single Table Mapping | |
|  |  |
| Joined Table Mapping | |
|  |  |
| Table Per Class Mapping | |
| **RetailBankCustomer** | **PrivateBankCustomer** |

### Diskussion

Beliebig viele Tabellen können zusammen auf eine Klasse abgebildet werden mit @SecondaryTable. Klassen können zusammen auf eine Tabelle abgebildet werden mit @Embedded. Klassen mit Mapping Informationen der Superklassen können in einer Tabelle abgebildet werden (mit @MappedSuperclass).

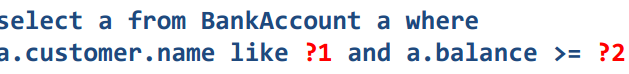
## Anfragen in JPA mit JPQL

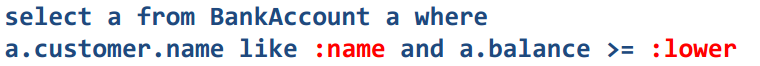
JPQL steht für Java Persistence Querying Language. Es ist eine Anfragesprache in der Analogie zu SQL, es operiert aber auf dem Entity Modell und nicht auf dem DB-Modell. Daher Entities & Fields statt Tables & Columns. Es wird dann vom JPA Provider in SQL übersetzt. Es kennt Path Expressions (=Implizit Joins).



### Beispiele

### Query Parameters

**Positional Parameters  
**

**Named Parameters  
**

### Parameter setzen

Es sind auch Entities als Parameter möglich. Somit wird die SQL-Injection verhindert.

Position   
Named 

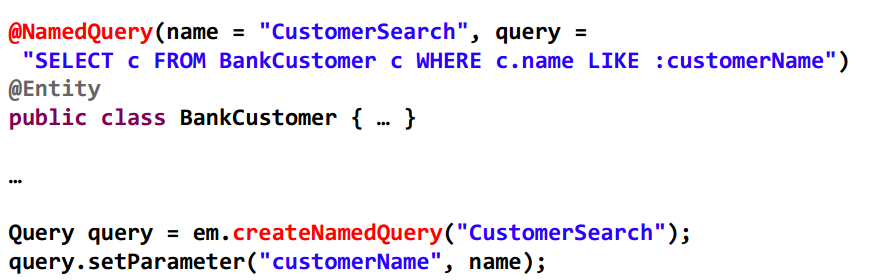
### Dynamic Queries

Eine Query welche zur Laufzeit gebaut und geprüft wird.

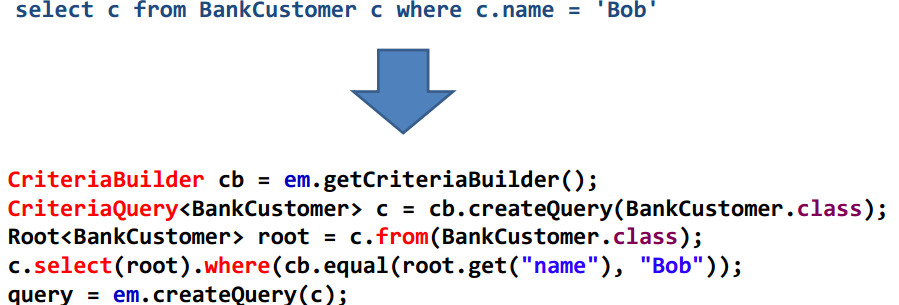


### Named Queries

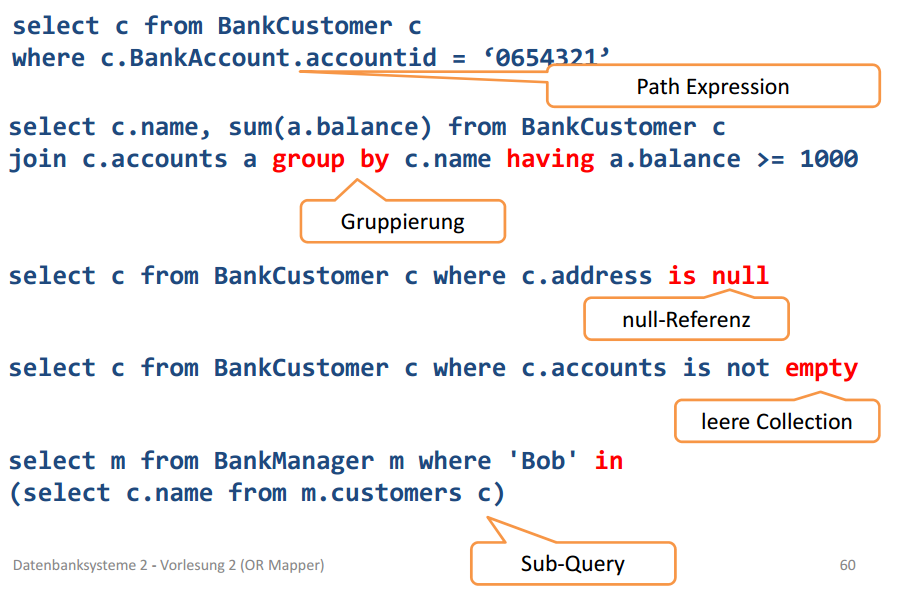
Queries welche statisch bekannt sind. Sie können vom JPA Provider vorgeparst und optimiert werden. Die Annotation @NamedQuery muss bei der Entity-Klasse annotiert werden.



### Criteria API

Queries mit Method Chaining und Generics. Es ist von Vorteil, da es statisch ist (zur Compilezeit geprüft). Es ist aber eine unübersichtliche Weise und wird daher nicht immer verwendet.  


### Weitere Beispiele



## JPA Diskussion und Implementationen

**Zur JPA-Spezifikation**  
Die Query-Sprache und die Annotationen «bindet» die Domain an die Persistenzsicht. RDBMS könnten aber noch abstrahiertet sein. Zudem besteht Bedarf an besserer Konfiguration für Property Access oder auch DB-spezifische Typen.

Es sind mehrere Implemtationen vorhanden, was sehr von Vorteil ist. Die Tools sind aber erst am Kommen und damit ihrem Standard voraus. Das heisst auch, dass sie noch nicht ausgereift sind. (zum Beispiel für benutzerdefinierte Datentypen).

### Fazit

Erfahrungen mit DataNucleus  
JPD bietet besser OO-Kompatibilität als das JPA. Es wird aber kein lazy-loading von grossen Recordsets unterstützt. Man kann sagen eine Two-Man Show.

**Erfahrungen mit Hibernate**  
Es ist konform zu JPA und EJB, hat jedoch auch eine eigene Query-Sprache. Ist ok falls RDBMS gesetzt ist und der Standard nicht im Vordergrund ist. Es ist verbreitet, hat aber mehr Dependencies als EclipseLink. Zudem bietet es eine breite Palette von unterstützten DBMS.

# Stored Procedures (SP)

## Was sind Stored Procedures?

“A SP is a subroutine available to applications that access a RDBMS”. Ein SP kann also mit User Definied Funtion, Subroutine oder Methode bezeichnet werden. Grundsätzlich basiert es auf den Procedure Calls. Die Programme sind so nahe oder gar bei den Daten gespeichert. Zudem bilden sie Grundlage für Triggers.

### Warum Stored Procedures?

* Domain-Logik (Funktionen und Datenkapselung)
* Performanz – Funktionen nahe bei den Daten
* Daten-Konsistenz (Triggers Security)
  + Verfeinerter Zugriffschutz
  + Log Auditing von Zugriffen
* Code-Wiederverwendbarkeit (Anstelle von Subqueries)
* Separation of Duties (Trennung der Aufgaben)

### Wann keine Stored Procedures?

Im Fall wenn Views oder ein Object-relationaler Mapper (ORM) genügen. Zudem macht es den Software-Entwicklungsprozess um ein Element komplexer. Die Wartbarkeit sowie Portierbarkeit darf man nicht ausser Acht lassen.

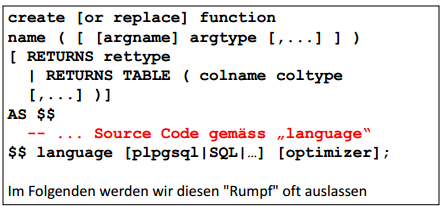
### Einführung

Sie bestehen aus Declarations, Assignments, Loops und sind in der DB gespeichert. Sie können aus anderen SP, Triggern, PL/SQL-Blöcken oder von der Client-Applikationen aufgerufen werden. Es sind Datenbankobjekte (Wie Tabellen). SP laufen innerhalb einer äusseren Transaktion). Zudem haben Sie ein eigenes Recht für die Ausführung.

## PL/SQL

SQL ist eine deklarative Programmiersprache, während PL/SQL eine prozedurale Sprache mit «eingebettetem» SQL ist. Jede Datenbank hat ihre eigene Abwandlung.

### Syntax



### Compiler

Der Code wird beim Aufruf geparsed und als Pseudocode in der DB gespeichert. Erst bei der ersten Ausführung wird die volle Syntax gecheckt. SQL Statements werden vorkompiliert und bei wiederholtem Aufruf wiederverwendet.

|  |  |
| --- | --- |
| **Hello World** | **Beispiel Increment** |

# PL/pqSQL - die Sprache

## Basis

**Deklaration und Block**DECLARE …. sowie BEGIN … END

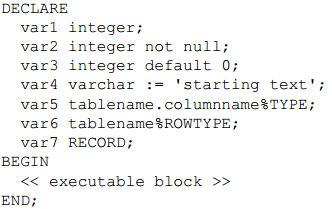
**Assignment**  
variable := expression;

**Rückgabe**  
return value;

**SQL einbetten**  
Direkt mit INSERT, UPDATE, DELETE oder SELECT… INTO…

**SQL ausführen (dynamisches SQL)**  
EXECUTE query\_string;

### Deklarationen



### Block und Kommentare

|  |  |
| --- | --- |
| **Blöcke** | **Kommentare**   * -- Single Line comment * /\* \*/ Multi Line Comments * Hinweis: Jedes Objekt kann auch eine Description haben |

### Conditoinals und Loops

|  |  |
| --- | --- |
| **Conditionals** | **Loops** |

### Exceptions

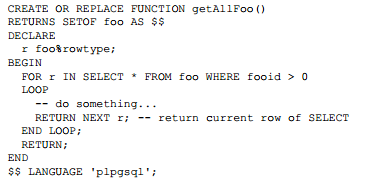
|  |  |
| --- | --- |
| **“Werfen”**   * ***Format***: RAISE level “format” …. * ***Level***: DEBGU, LOG, INFO; NOTICE, WARNING * ***Bsp***. RAISE NOTICE “notice %”, id; | **“Fangen”** |

## Beispiele

|  |  |
| --- | --- |
| SELECT…INTO | INSERT INTO |
| UPDATE | Function that returns void = SP |
| Function that returns a scalar | Function that returns a table |

## SET Returning Functions

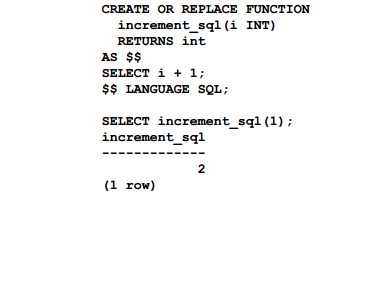
Zu den Datentypen stehen SETOF oder RECORD zur Verfügung. Sowie das Schlüsselwort RETURN NEXT.



# Stored Procedures in SQL und Python (PostgeSQL)

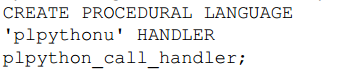
## SP mit SQL (PostgreSQL)

SQL – inline, wie Makros



## PL/Python (PostgreSQL)

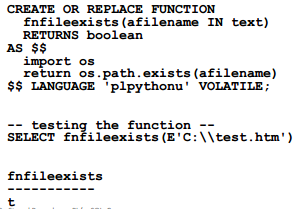
Dabei ist Python in den DB-Server eingebettet. Es nutzt libpq, das C API zu PostgreSQL. Die Python-Binaries müssen installiert werden. Python muss man in PostgreSQL zuerst installieren.



PL/Python ist eine «untrusted» Sprache, darum das «u» in plpythonu». Trusted ist ein «Security Feature».

### Beispiel File Exists

Prüft ob ein File existiert (dazu wird unteranderem “untrusted» benötigt).



## Gemeinsamkeiten Stored Procedures

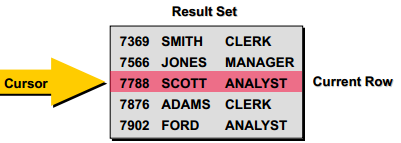
Folgendes ist gemeinsam mit den Store Procedures von PostgreSQL:

* Parameter
  + Namen; IN-Parameter (call by value); OUT(by reference)
* Funtion Overloading
  + HelloWorld(integer), HelloWorld(text)
* Variadic Functions
  + Varargs, «…»
* Set Returing Functions
  + RETURN SET OF….
* Function Polymorphism
  + Anyelement, anyarray

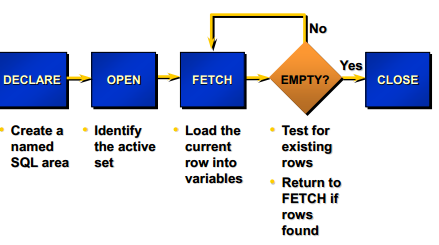
# Cursor

*«handle or name for a “private SQL area” - an area in memory inside DB server in which a parsed statement, procedures or functions are kept”*

Ein Select-Statement liefert in der Regel eine Menge von Tuples (=Result Set). Der Cursor erlaubt den sequentiellen Zugriff auf die einzelnen Tuples des Result Set.



## Verarbeitung

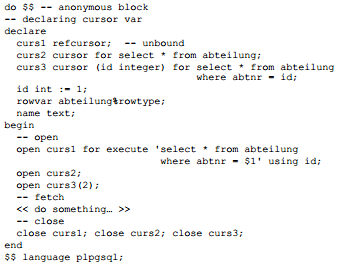


|  |  |
| --- | --- |
|  |  |

## Prinzip

|  |  |  |
| --- | --- | --- |
|  |  |  |

## Beispiel



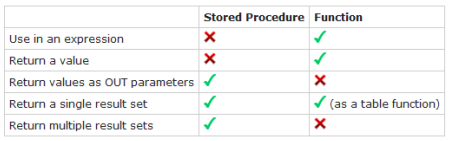
## Cursor mit Parametern

|  |  |
| --- | --- |
|  |  |

## Cursor for Update

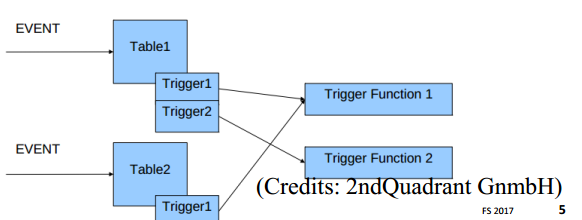
|  |  |
| --- | --- |
|  | **Beispiel** |

## Vergleich Prozedur / Funktion



# Triggers

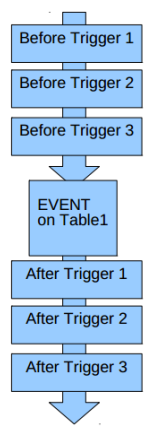
Triggers sind Implementationen von komplexen Konsistenzbedingungen. Es sind Datenbank-Objekte und immer einer Tabelle zugeordnet. Sie werden in Stored Procedures programmiert und haben keine Parameter. Zudem können Sie nicht direkt aufgerufen werden. Sie werden vom DBMS beim Eintreten eines Events aufgerufen. Bei der Ausführung haben Sie die Rechte des Owners.

Triggers können Parameter an eine Funktion weitergeben. FOR EACH [statement | ROW]. Bei den Events steht INSERT, UPDATE, DELETE und TRUNCATE zur Verfügung. Die Funktion wird vor oder nach Änderungen ausgeführt.

**Einsatzgebiete**

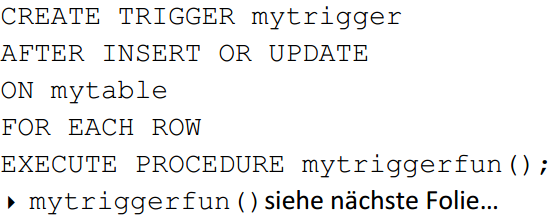
* In der Sicherheit, z.B. das Änderungen an Angestelltendaten nur während der Arbeitszeit an Wochentagen ausgeführt werden können
* Berechnen von abgeleiteten Attributen
* Sammeln von Statistik- und Logdaten
* Für Updatable Views

## Event Model

Die «Before Triggers» können den Inhalt von neuen Zeilen ändern. «After Triggers» können nur auf das reagieren, was passiert ist. Die Return-Werte der «After Triggers» werden ignoriert. Die Triggers werden in alphabetischer Reigenfolge ausgeführt.

## Syntax

**DDL für Triggers  
**

**Syntax Beispiel eines CREATE TRIGGER  
**

## PL/pgSQL

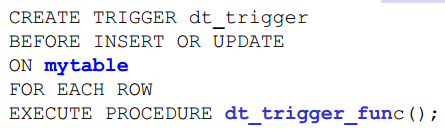
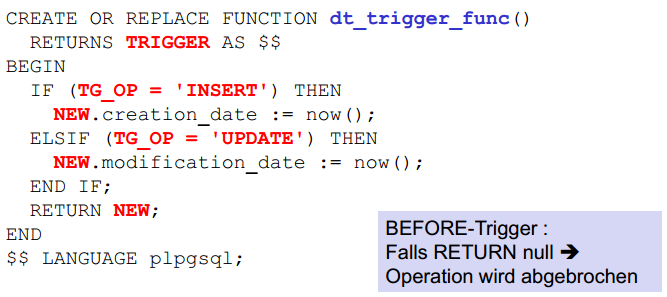
### Trigger-Funktion

Eine Trigger-Funktion hat keine Funktionsparameter. Diese werden über Trigger-Funktions-Variabeln übergeben, unter anderem TG\_NARGS (Anzahl Parameter) und TG\_ARGV[] (Array von Parametern als Text).

### Trigger-Funktionsvariabeln

|  |  |
| --- | --- |
| TG\_NAME | Name des Triggers(TG) |
| TG\_WHEN | BEFORE oder AFTER |
| TG\_LEVEL | ROW oder STATEMENT |
| TG\_OP | INSERT, UPDATE, DELETE (TRUNCATE) |
| TG\_RELID | OID der Tabelle |
| TG\_RELNAME | Name der Tabelle |
| TG\_TABLE\_SCHEMA | Schema der Tabelle |

### Beispiel mit Funktion



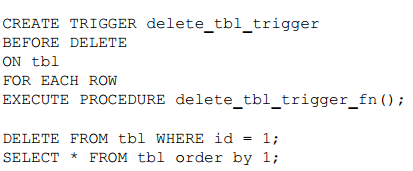
### Datenzugriff

Diese Ergänzungen sind nur für ROW-Level Triggers (nicht STATEMENT) möglich. Dort sind die Trigger-Funktions-Variabeln NEW und OLD vom Typ Record vorhanden.

* NEW (Tupel vor der Änderung)
  + Nur bei INSERT und UPDATE Triggers
  + E.g. NEW.creation\_date := now();
* OLD (Tupel nach der Änderung)
  + Nur bei UPDATE und DELETE Triggers

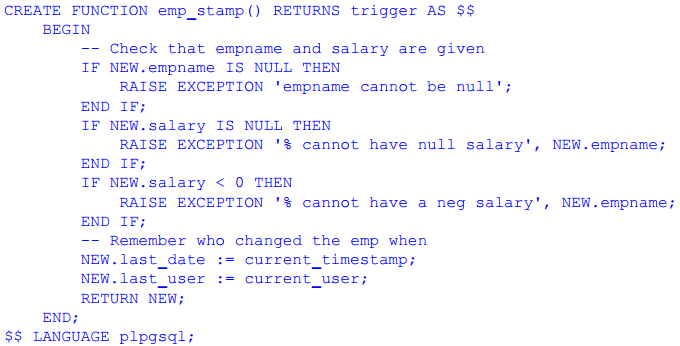
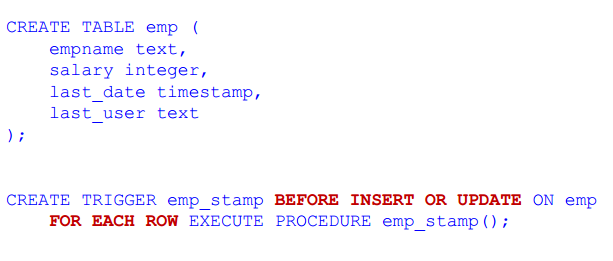
## Weitere Beispiele

### Beispiel mit DELETE



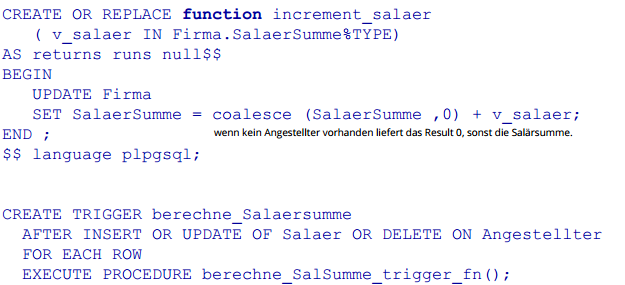
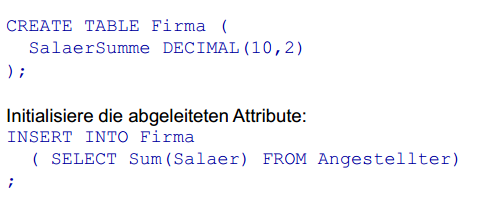
### Beispiel 2

Sicherstellen von Vorbedingungen und das Berechnen von zusätzlichen Attributen.



## Abgeleitete Attribute

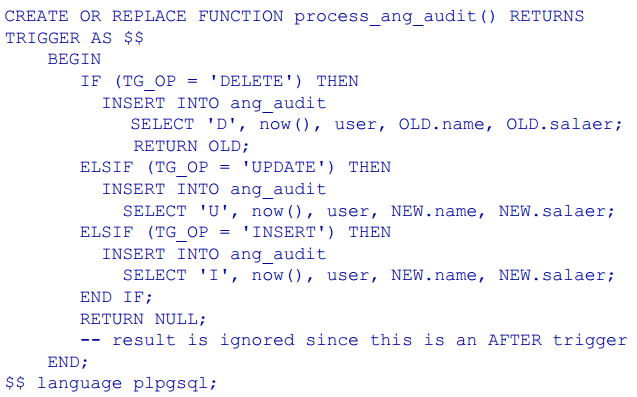
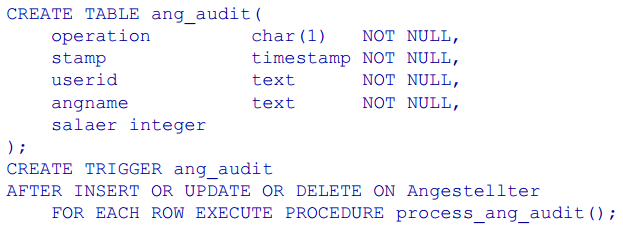
Mit diesen Attributen sollte die Redundanz in einer Datenbank vermieden werden. Die Tabelle enthält berechnete Werte (zum Beispiel die Salärsumme). Dieser abgeleitete Wert muss nach jeder Änderung der Basisdaten (z.B. Salär des Angestellten) neu berechnet werden.



## Auditing mit Triggers

Triggers können auch zum Auditing von Änderungen benutzt werden, um so Änderungen nachweisen zu können. Wichtig: Wir bei einem Rollback die Transaktion rückgängig gemacht so geht auch der Auditingeintrag verloren.

Der Trigger LogSalaerAenderung wird immer aufgerufen, wenn mit update Angestellter SET Salaer = …. ein Salär verändert wird. Der Trigger schreibt einen Auditsatz in die Tabelle ang\_audit.



## Ausführung von Triggers

In der folgenden Reihenfolge werden die Triggers ausgeführt.

* Führe alle BEFORE Statement Triggers aus
* Für jedes betroffene Tupel gilt:
  + Führe alle BEFORE Row Triggers aus (Abbruch, wenn Trigger-Funktion null liefert)
  + Bearbeite Tupel (Locks werden gehalten)
  + Führe alle AFTER Row Triggers aus
* Führe alle AFTER Statement Triggers aus.

## Diskussion Triggers

Triggers machen eine Datenbank tendenziell langsamer und schwerer wartbar (z.B. bei Load, Dump/Restore). Zudem ist es möglich die Triggers bei manchen DB’s (vorübergehend) auszuschalten. Die Implementation der Triggers von PostgreSQL und Oracle sind ähnlich. Es gilt aber bei Casading Triggers aufzupassen

# Updatable Views und weitere Views

## Updatable Views

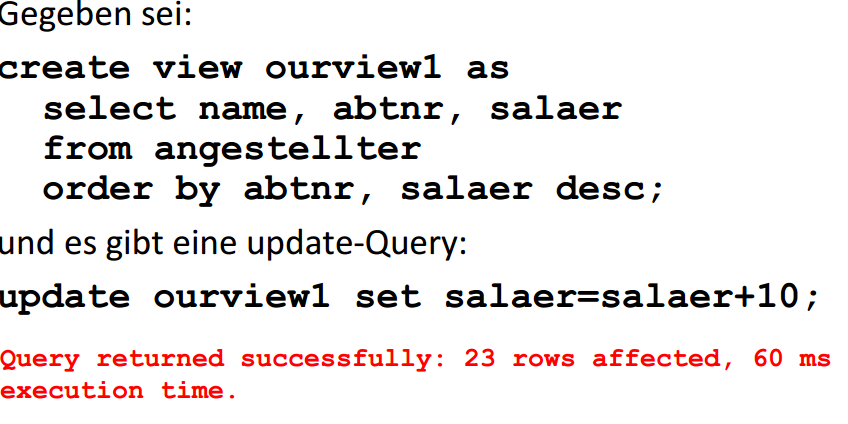
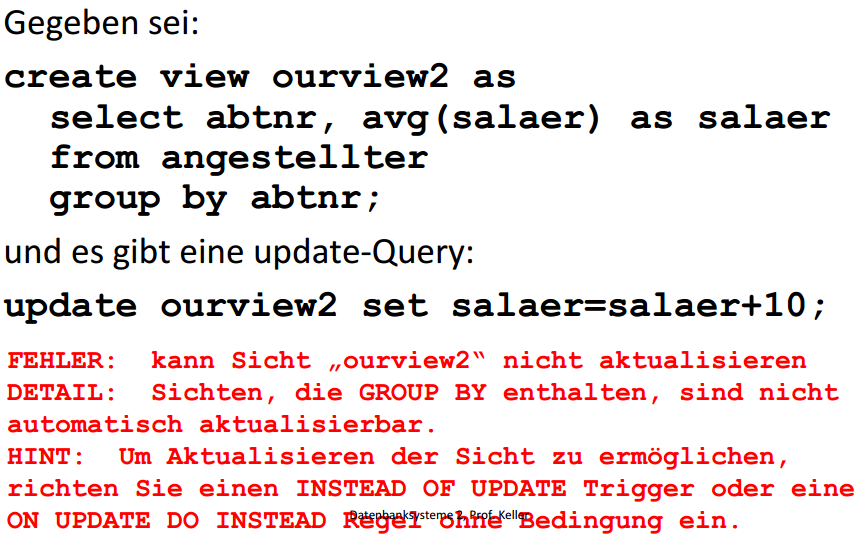
Views sind externe Sichten für die Applikation (Kapeslung) und verringern die Komplexität von Queries. Bei Views sind Modifikationsoperationen (insert, update, delete) möglicherweise nicht mehr eindeutig. Dieses Problem kann durch Updatabele Views oder INSTED-OF Triggers stattfinden. Views sind updatable, wenn die untengenannten Bedingungen erfüllt werden. Man spricht dann von automatisch aktualiserbaren Views. Geht dies nicht, muss mit INSTEAD-OF gearbeitet werden.

Eine View ist automatisch aktualisierbar (updatable), wenn:

* Die View genau einen Eintrag in der FROM-Klausel, der eine Tabelle oder eine andere updatable View sein muss.
* Die View darf keine WITH, DISTINCT, GROUPBY, HAVING; LIMIT oder OFFSET enthalten.
* Die View darf kein UNION, iNTERSECT oder EXCEPT enthalten.
* Die Select-Liste der View darf keine Aggregation, Window-Funktion oder SET-returning Funktion enthalten.

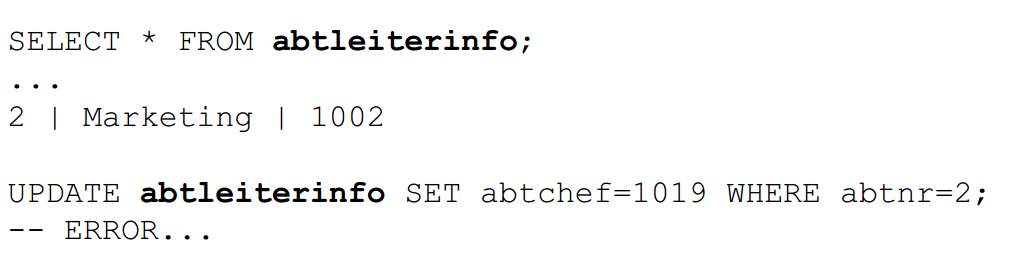
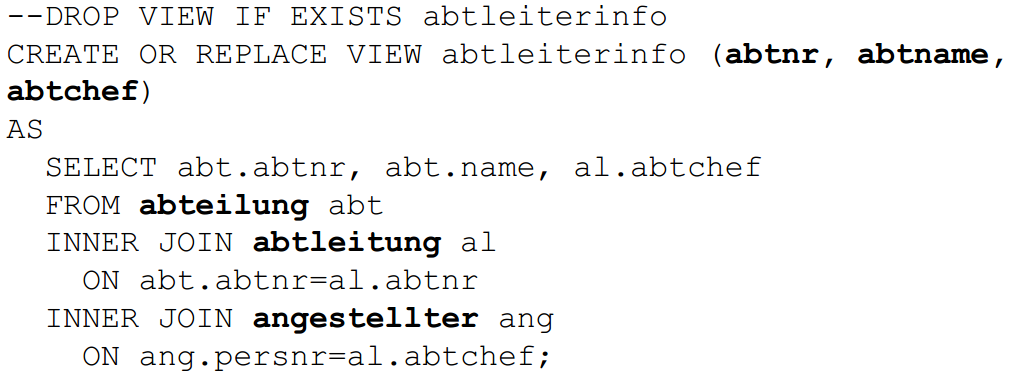
Eine Kolonne ist aktualisierbar, wenn Sie eine einfache Referenz auf eine aktualisierbare Kolonne der darunterliegenden Relation ist. Eine automatisch aktualisierbare View kann ein Mix von aktualisierbaren und nicht aktualisierbaren Kolonnen enthalten.

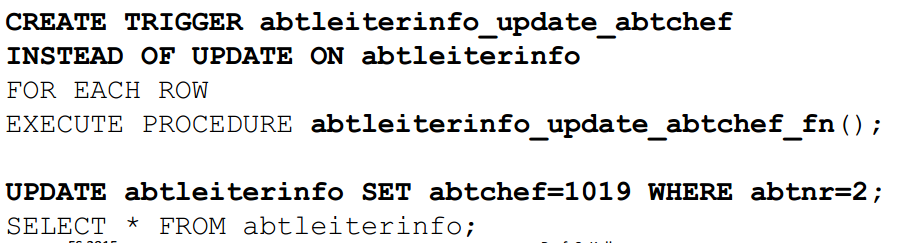
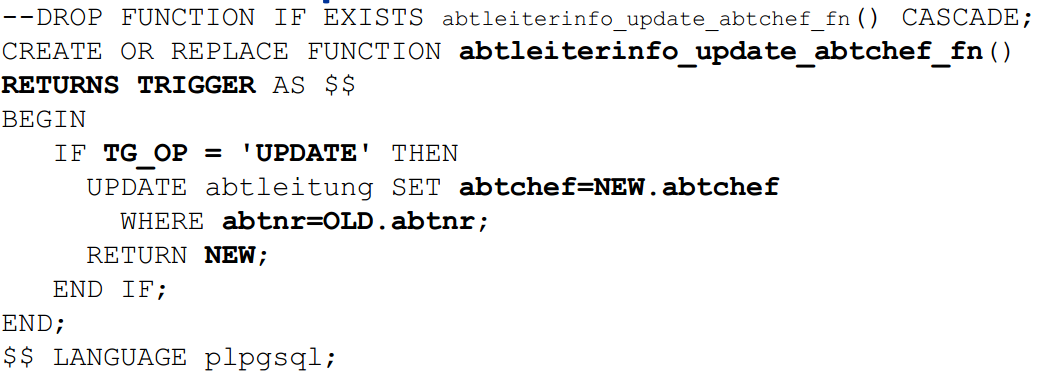
### Beispiele

## Instead-of Triggers

Die Grundlagen der Triggers wurde bereits im vorhergehenden Kapitel behandelt. Instead-of-Triggers werden anstelle der ursprünglichen SQL-Operation ausgeführt. Es können damit Modifikationen auf Tabels und Views definiert werdem. Sie leiten Modifikationen (INSERT, UPDATE; DELETE) auf die Views der darunterliegenden Tabelle weiter.

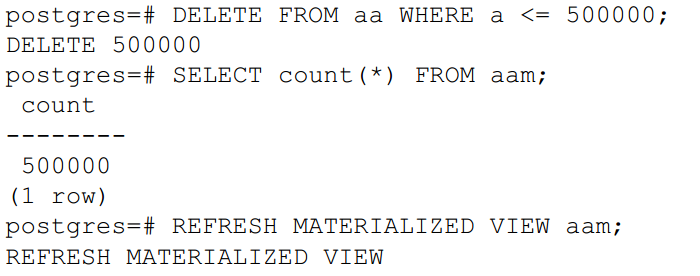
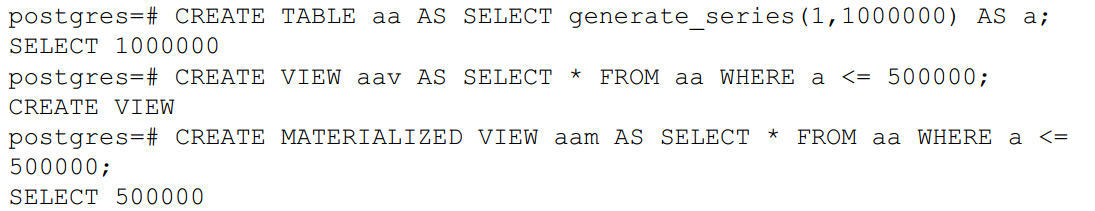
**Beispiel Normal**  


**Beispiel INSTEAD OF INSERT**

## Materialized Views

Gemäss Definition ist eine View eine logische Relation (oder virtuelle Tabelle), welche über eine im DBMS gespeicherte Abfrage definiert wird. Diese kann wie eine normale Tabelle abgefragt und wird bei jedem Aufruf neu berechnet.

Eine «Materialized View» ist gemäss der Definiton eine zwischengespeicherte View. Sie wird nur wenn explizit angeben, neu berechnet.

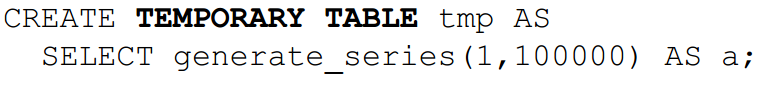
**Beispiel**  


**Indexe auf Views**In PostgreSQL und in Oracle gibt es keine Indexe auf Views. Diese greifen auf die Indexe der darunterliegenden Tabelle zu. Bei MS SQL gibt es Indexe auf die Views. Und Indexe auf Marterialized Vies gibt es in Oracle und in MS SQL Server.

Views kann man (theoretisch) auch zwischenspeichern. Dies ist typisch in einem Data Warehous. Es gibt aber Probleme, ausser mit read-Only Daten. In Oracle und im MS SQL Server ist dies implementiert.

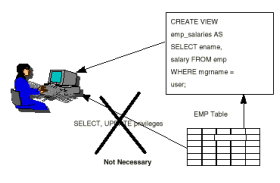
## Temporäre Tabellen

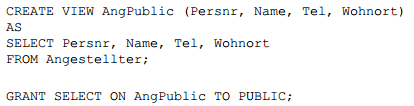
Erzeugt eine Tabelle, die am Ende der Transaktion oder der Session automatisch gelöscht wird. Anwendung findet es bei komplexeren Auswertungen, die Zwischendaten halten müssen. Es wird von fast allen DBMS unterstützt.



# Realisierung von Sicherheit mit Views, Stored Procedures und Triggers

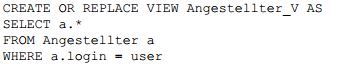
## Zugriffsschutz mit Views

 Der Benutzer greift in diesem Fall über die View auf die Daten zu. Die Views erlauben horizontale und vertikale Filterung der Daten. Der Benutzer benötigt keine Privilegien für den Zugriff auf die darunterliegende Tabelle. Mit Views können vertrauliche Daten geschützt werden: Zuerst wird ein Sicht mit den öffentlichen Attributen definiert (vertikale Filterung), dann wird diese mit der Grant-Anweisung allen Benutzern zugänglich gemacht. Der Zugriff auf die unterliegende Tabelle wird verwehrt. Vertrauliche Daten, die in der Sicht nicht enthalten sind, bleiben so geschützt.



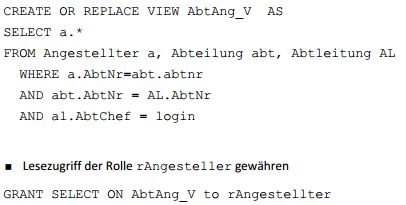
## Row-Level Security

Verschiedene Benutzer sehen unterschiedliche (horizontale) Ausschnitte der Tabellendaten. Diverse DBMS bieten direkte Unterstützung an. Mit Views kann der Zugriff auf einzelne Tupels einer Tabelle eingeschränkt werden (horizontale Filterung).

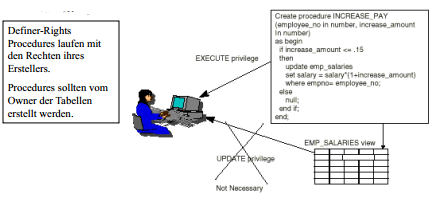
Beispiel Ein Angestellter soll vollen Lese-Zugriff auf seine Daten haben  


Die Funktion «user» liefert den Login-Namen des angemeldeten Benutzers. Dazu wird in der Tabelle «Angestellter» der Login-Name abgelegt. Die View Angestellter\_V liefert nur noch die Daten des angemeldeten Benutzers.

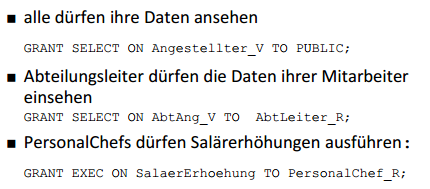
Falls der angemeldete Benutzer Abteilungsleiter ist, darf der die Daten seiner Mitarbeiter einsehen.



## Zugriffschutz mit Stored Procedures

Der Benutzer (Applikation) modifiziert über den Aufruf von Stored Procedures die Daten. Stored Procedures kapseln die Daten und können Teile der Geschäftslogik implementieren. Der Benutzer benötigt keine Privilegien für die Modifikation der darunterliegenden Tabelle. Modifikationsopertationen mit Stored Procedures und Lesezugriff über Views.

### Beispiele



## Auditing mit Triggers

Wie im Kapitel Triggers beschrieben, kann ein Audit Log mit Triggers erstellt werden, um nach zu verfolgen, wer was gemacht hat.

# Datenstrukturen

## Was ist eine Collection?

Gruppierung einer variablen Anzahl von Datenelementen (...) allgemein (...) des gleichen Typs oder (...), die von einem gemeinsamen Datentyp abgeleitet sind. Eine Collection ist eine abstrakte Datenstruktur und schreibt keine spezifische Implementierung vor, obwohl es oft eine konventionelle Auswahl gibt.

**Systematik**  
Es wird dabei zwischen den linearen Collections und den assoziativen Collections unterschieden. Zudem gibt es noch Graphen und Trees. Zu linearen Collections zählen Listen, Array, Priority Queues und Heaps. Zu den assoziativen Collections zählen Sets, Mutlisets, und assoziativen Arrays.

## Was ist ein Datentyp?

Was braucht es für einen (neuen) Datentyp (in Datenbanken)?

|  |  |
| --- | --- |
| **Datentyp** | **Beispiele** |
| Name/Deklaration | Easteregg |
| Datenstruktur | Outer: text; inner: text |
| Konstruktoren | Easteregg\_from\_text(‚x’,’yy’) |
| Accessor-Funktionen | … |
| Update-Funktionen | … |
| Hilfs-Funktionen | Easteregg\_astext() |
| Operatoren | + |
| Indexe | … |

## Arrays

Arrays ist eine Art „Collection“ mit einer fixen oder variablen Länge. Arrays sind aber keine Sets. Denn Sets modellieret man als spearate Tabelle (besonders bei vielen Updates). Bei PostgreSQL hat der Basistyp eine variable Länge.

Es ist daher eignet für wenige Updates / Inserts, ist aber ein Datentyp für andere Software-Bibliotheken. Zudem ist es das Basiselement zur Repräsentation von Bäumen und Pfaden. Es ist „nicht-relational“ aber genormt.

### Szenario 3D-Array

Die Ergebnisse einer Simulation sollen als 3D-Arrays mit Float-Werten verwaltet werden. Dessen Dimensionen sind in allen drei Richtungen variabel, bewegen sich aber im Bereich 200x200x10 (~10MB) und sind pro Simulation fix.

**Typische Zugriffsmuster auf das 3D-Array sind**:  
a. 2D-Subarray (200x200 Elemente) auslesen;  
b. 1D-Subarray (200 Elemente) auslesen, meist eine Spalte oder eine Zeile eines 2D-Subarrays;  
c. Auf eine Untermenge von a) oder b) selektieren.

**Lösungsansätze**

1. Array über nxm – Beziehungen zu Tabellen normalisieren
2. Array-Typ verwenden
3. PostGIS 2.0 Raster (Multiband mit 10 Bändern) verwenden-

### Konstruktoren und Accessoren

|  |  |
| --- | --- |
|  |  |
|  |  |

Funktionen sind auch möglich. Dazu schaut man aber am besten in der Dokumentation des jeweiligen Herstellers nach.

### Operatoren

Equality „=“  


„Is contained by“ <@  
 

Overlap &&  


## Dictionaries

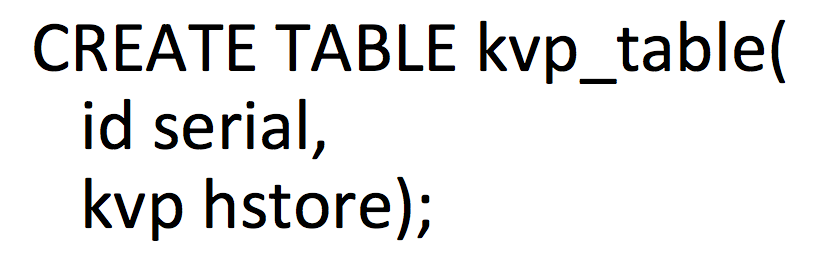
Ist eine abstrakte Datenstruktur und ein Datentyp zur gleichen Zeit. Alternative Bezeichnungen für Dictionary sind „Associative Array“, „Map Collection“, Key-Value Pair oder Hash.

Die meisten modernen Computersprachen unterstützen Dicitionaries als (primären Basis-)Datentyp.

### Key Value-Paare

Key Value Pairs (KVP), auch "Entity Attribute Value" (EAV) Modell oder Objekt-Attribut-Wert-Modell in der Datenbank-Technologie genannt. Kennen Sie die Ausnahmen! "RDBMS sind auf Konsistenz und Effizienz ausgelegt. Durch die Verwendung von KVP-Schemas werfen Sie die Abruf-Fähigkeit in den Abfluss! "

**Beispiel**

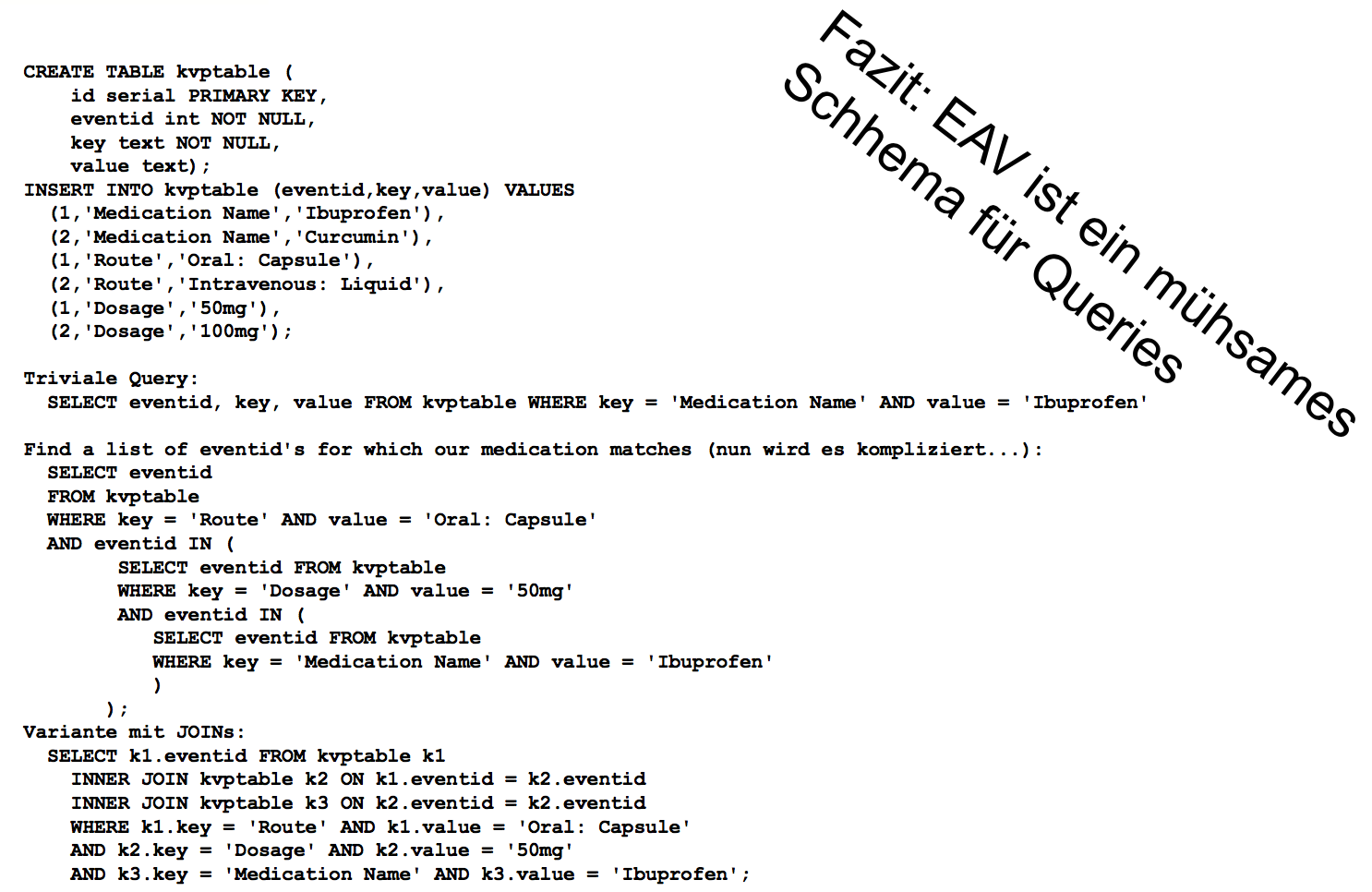
**Evtl. geeignet für**

* Einfache Datenspeicherung und Erfassung
* Zeilen mit vielen Attributen, die selten verwendet werden.
* Semi-strukturierte Daten

**Tipps**

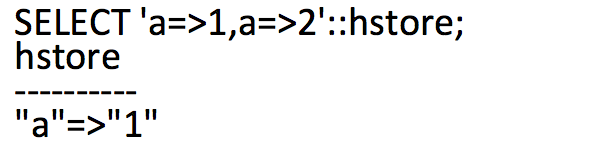
* Versuchen Sie, die Berichterstattung gegen große Heaps von Dictionaries zu vermeiden
* Betrachten Sie zusätzliche Lookup-Schlüssel.
* Betrachten Sie das Ausführen von Triggern oder eines posteriori Prozesses, um KVPs in ein Entity-basiertes Schema zu verwandeln.

### Beispiel mit EAV



### Hstore in PostgreSQL

HSTORE implementiert Dictionaries als Datentyp. Keys und Values sind vom Typ Text. Er wird unterstützt vom GIST/GIN Index.

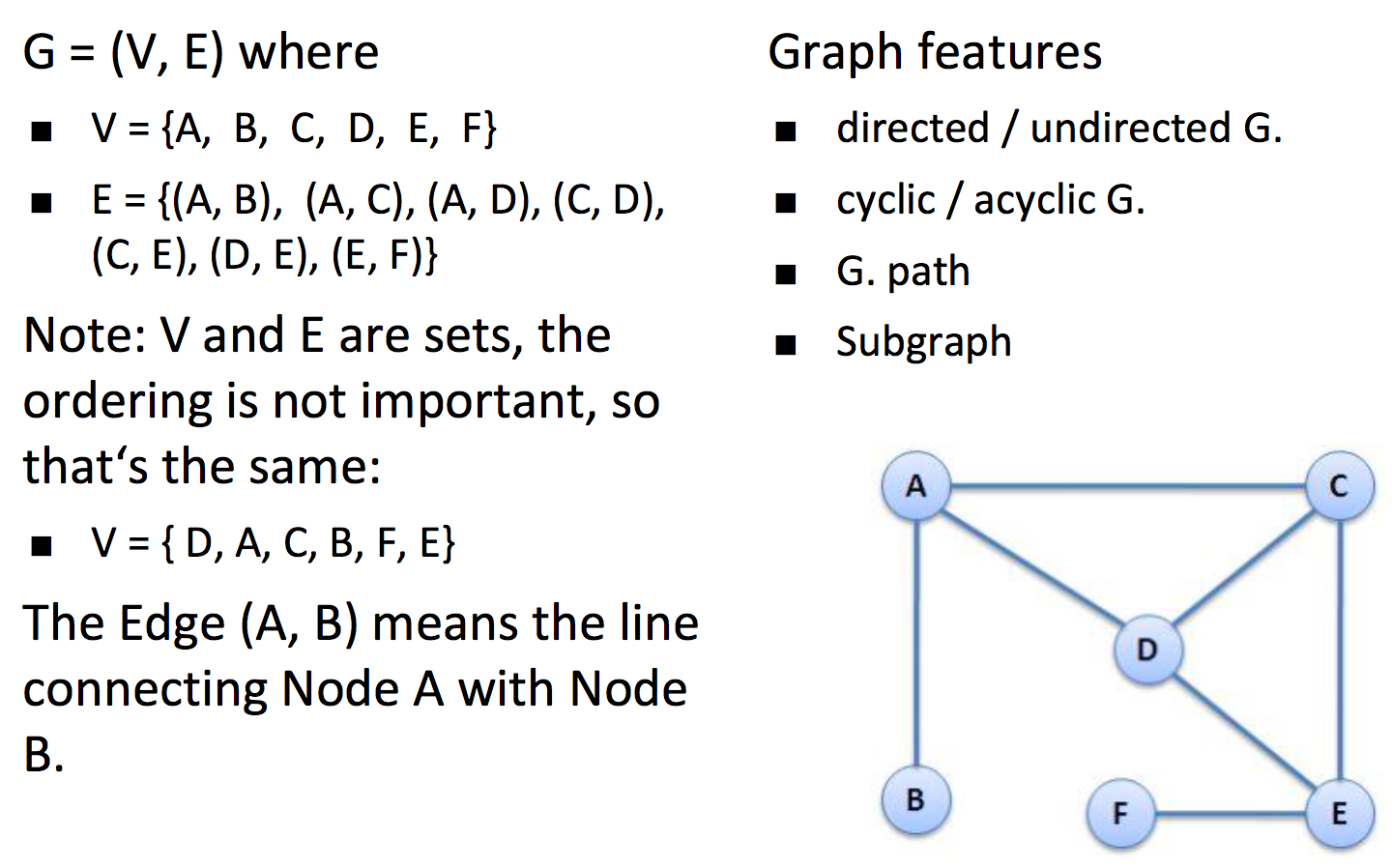
Bei der Syntax müssen die Keys eindeutig sein, doppelte werden ignoriert.  


|  |  |
| --- | --- |
|  |  |
|  |  |

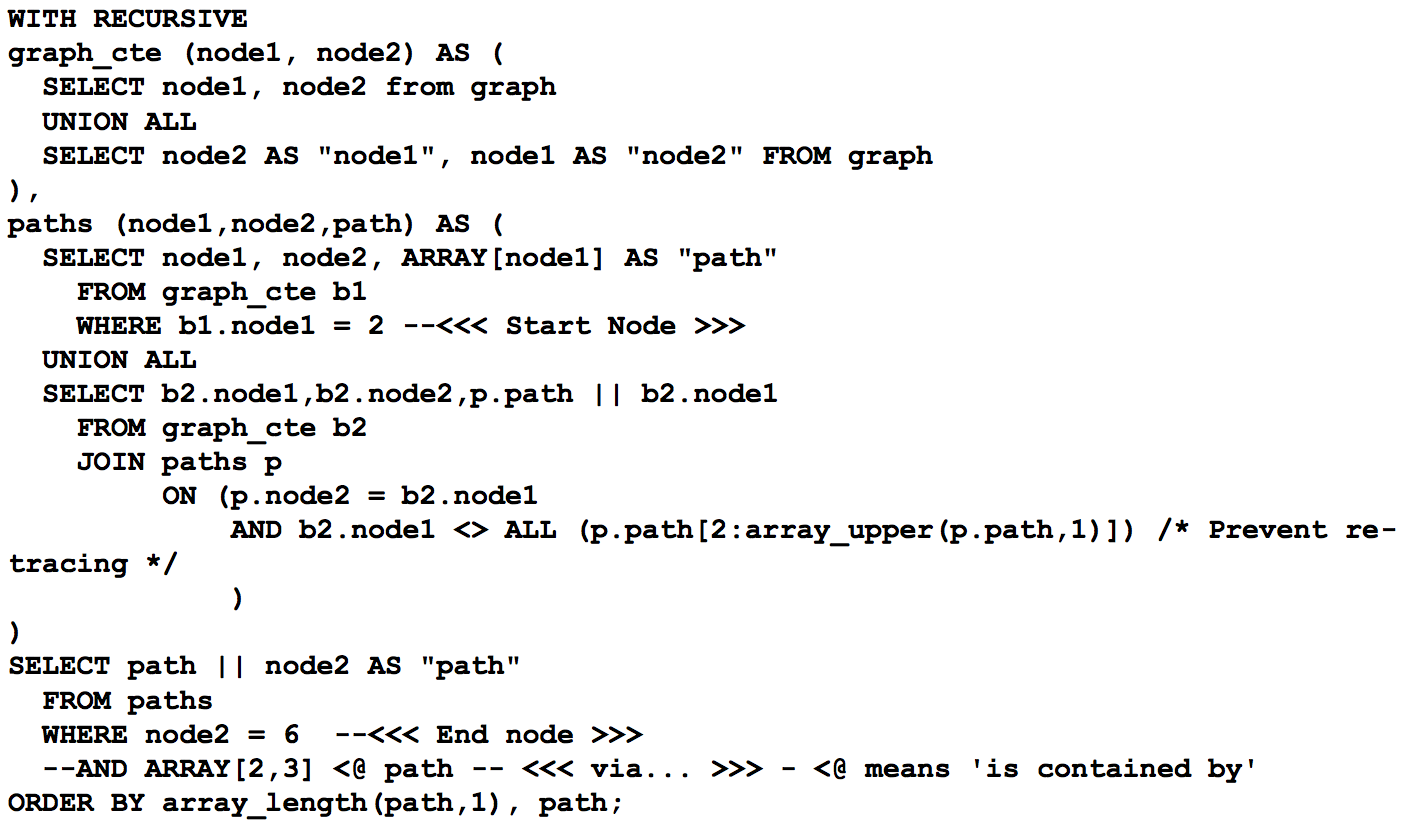
## Graphen

Ein Graph G ist eine Struktur aus einem Set von Nodes oder Vertizes (V = {V1, V2...Vn}) und einem Set von Edges (E = {e1,e2,e3,...en}). Ein Edge verbindet zwei Vertizes. Es repräsentiert ein Pair von Vertizes. (e1 = Vi,Vj).

### Beispiel



### Abfragen eines Graphen (Trick mit CTE...)

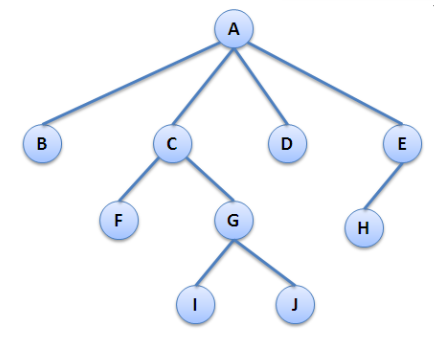
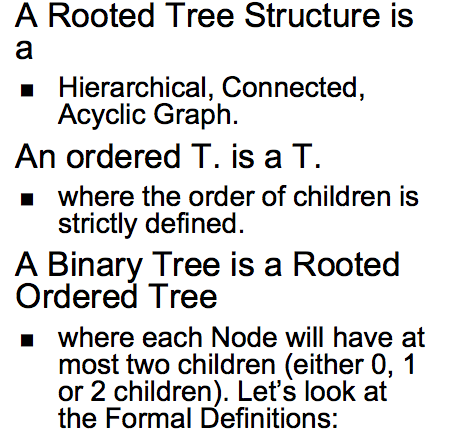


### Diskussion

Rekursive Abfragen sind ein erster Weg, um mit Graphen in SQL zu arbeiten. Grapgen werden im Transport, Engergie und Netzwerkumfeld verwendet. Im GIS wird es für die kürzesten Wege und eine Netzwerkanalyse genutzt. Die Bäume sind Subtypen von Graphen. (siehe später).

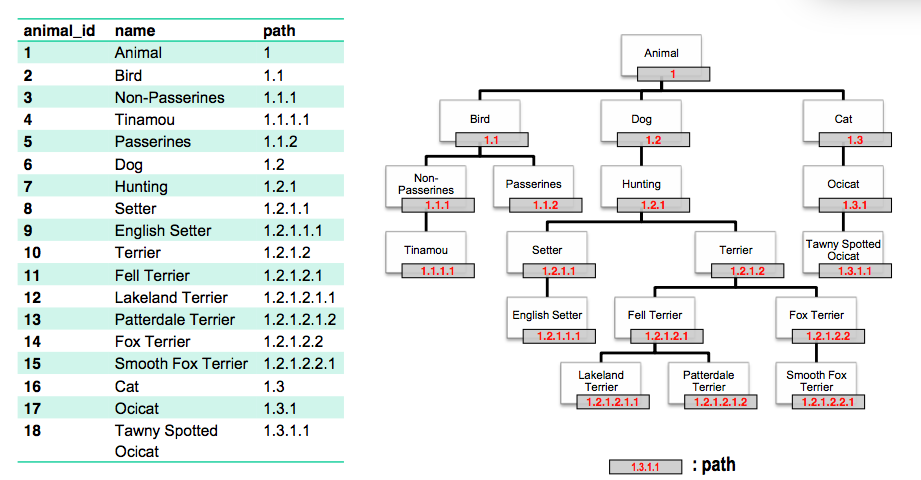
## Trees

Wieso Baumstrukturen? Dafür gibt es viele Beispiele: Organisationen (Hierarchie), Stücklisten, Kataloge, oder Tags. Allgemein kann man sagen für hierarchische Aufzähltypen und Dokumente. Die Datencodierung findet in Listen, Tabellen, XML und JSON statt. Typische Abfragen sind sämliche Nachfolger eines Knotens, sämtliche Vorgänger eines Knotens oder alle Kinder eines Knotens.

Ein Baum ist ein azyklisch verbundener Graph.   


**Tree Features**  
Vorgänger, Kind und Geschwisterknoten. Der Rootknoten ist der einzige Knoten ohne Vorgänger. Ein Leaf-Knoten ist einer ohne Kindknoten. Interne Knoten sind alle, welche kein root oder kein leaf Knoten sind.

|  |  |
| --- | --- |
| **Adjazenzliste (Relationale Implementation)** Diese arbeitet mit den Spalten ID und ParentID. Die Art und Weise ist einfach und ist schnell für Moves, Inserts und Deletes. Es ist aber aufwändig für das Finden von Level, Ancestry und Descendants sowie dem Pfad. Es werde CTE für Abfragen (Traversierung) verwendet. | **Nested Set-Modell (Rel. Implementation)**  Ist eine Art „Modified Preorder Tree Traversal“. Dabei gibt es die Spalten links und rechts. Es ist einfach für Level, Ancestry und Descendants. Aber aufwändiger bei Moves, Inserts und Deletes im Vergleich zur Adjazenzliste. Zudem verlangt er eine bestimmte Ordnugn. Falls andere verlangt wird, ist der Rechenaufwand gross. |

**Materialisierter Pfad  
**

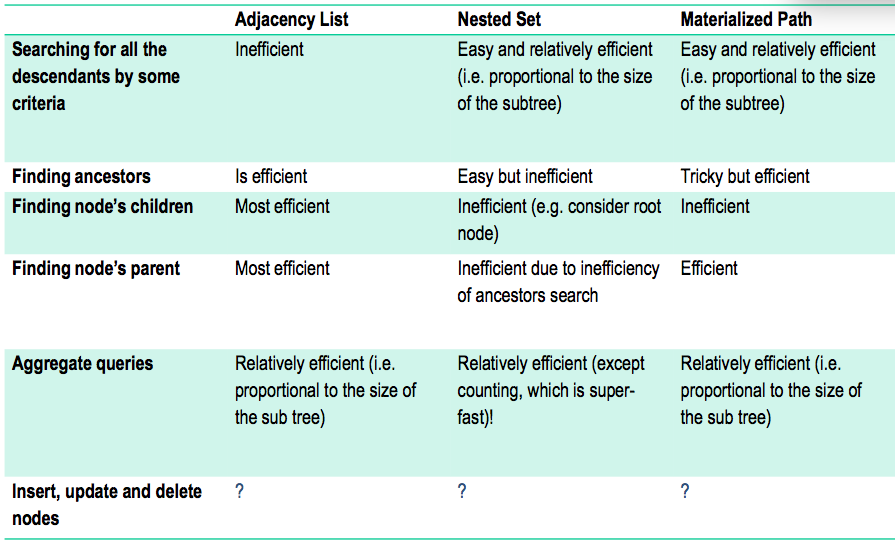
### Tree-Type „Ltree“ in PostgreSQL

Label Tree ist ein ProgreSQL Modul (Extensions). Es lässt hierarchisch strukturierte Beschriftungen/Kennzeichnungen (labels) zu. Indexierbar ist es mit btree und GGIST. Suchfunktion wie bei der Volltextsuche mit tserach. Zum Teil ähnlich wie Arrarys und hstore. Das Label darf nun alphanummerische Zeichen enthalten. Der Label Path besteht aus einer Sequenz von Labels separiert mit einem „.“.

**Beispiel**  
Die Extensions muss einmalig pro DB installiert werden mit CREATE EXTENSION ltree.

|  |  |
| --- | --- |
| **Operatoren (Auszug)** | **Funktionen (Auszug)** |

### Vergleichstabelle



### Zusammenfassung

Dictionaries, Graphen und Bäume kennen nur wenigen DBMS. In PostgreSQL gibt es Arrays, hstore und JSON. Mit dem Index gut unterstützt und können sogar bei Benchmarks mithalten.

## Semistrukturiere Daten

### XML in PostgreSQL

|  |  |
| --- | --- |
|  |  |

### JSON in PostgreSQL

Bei JSON spricht man vom Speichern von Text als Javascript Object Notation.

In PostgreSQL gibt es zwei verschiedene JSON-Datentypen.

* Json 🡺 Speichert die exakte Kopie des Textes
* Jsonb 🡺 Speichert geparstes Binärformat. Der Input ist dabei langsamer, die Verarbeitung aber schneller da kein Reparsing. Zudem sind die Indices schneller.

#### Tour

|  |  |
| --- | --- |
| **JSON-Tabelle aus relationaler Tabelle erstellen** | **JSON-Daten abfragen** |
| **JSON-Tabelle mit externen JSON-Daten erstellen** | **Aktualisieren und Löschen von JSON-Daten** |

# Interne Ebene - Optimierungen

## Indexstrukturen

Ein Index ist eine Datenstruktur für den effizienten Zugriff auf die Daten. DBMS speichern grosse Datenmengen. Um auf diese schnell zugreifen zu können, müssen die Daten speziell (um)strukturiert werden. Indizes dienen nicht der Speicherung, sondern der Beschleunigung der Suche. Das heisst auch, ein Index bedeutet immer einen gewissen Overhead an Daten. Klassischer Denkfehler: Braucht es einen Index, um die Salärsumme aller Mitarbeiter zu berechnen?

### Allgemeines

Die Effizienz der Verarbeitung der Daten hängt von gewählter Datei-Organisation und vorliegenden Indexstrukturen ab. Relevante Operationen sind hier folgende:

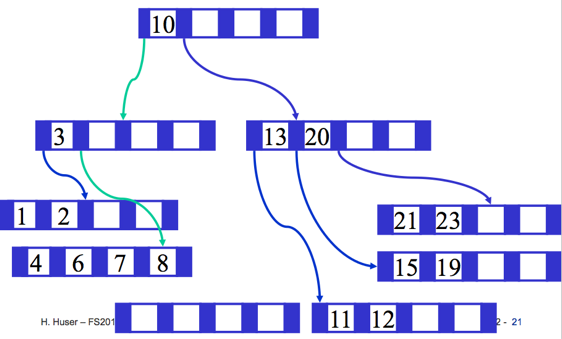
* Durchsuchen (**Scan**) – Alle Seiten, die Sätze enthalten, werden gelesen
* Suchen in Abhängigkeit von einem Gleichheitstext (**Equality Search**) – Alle Seiten, die die Sätze enthalten, die die Gleichtheitsbedingung erfüllen, werden gelesen.
* Suchen in Abhängigkeit von einem Bereichstext (**Range Serach**)
* Einfügungen (**Insert**) – Lesen und Schreiben einer oder mehrerer Seiten erforderlich.
* Löschungen (**Delete**) – analog Insert.

### B-Bäume

Balancierte Mehrwegs-Suchbäume. Dabei steht das „B“ für Bayer oder Balanced (aber nicht binär). Geeignet für Sekundärspeicher und Range- und Equal-Serach.

**Varianten**  
B+-Baum B-Baum mit „Daten“ separat nur in Blättern (analog ISAM)  
R-Baum mehrdimensionaler (räumlicher) balancierter B-Baum

Normale „Binär-Bäume“ (wie AVL) sind gut geeignet für den Hauptspeicher. Bei Datenbanken benötigt man aber ein Verfahren, das auf die Seitengrösse des Hintergrundspeichers abgestimmt ist. Die Lösung dafür sind eben die B-Bäume.

Die Idee ist, dass die Knotengrösse der Seitengrösse entspricht. Der Verzweigungsgrad hängt davon ab, wie viele Einträge auf einer Seite Platz haben. Durch entsprechende Algorithmen für das Einfügen und Löschen von Daten garantiert, dass der Baum immer ausbalanciert ist 🡺 Logarithmische Zugriffszeiten.

|  |  |
| --- | --- |
| **Einfügen** | **Löschen** |

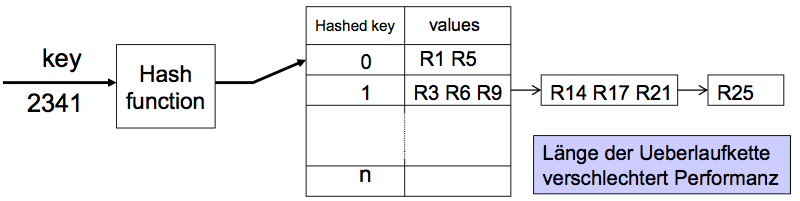
### B+-Baum

Da sind die Datenelemente nur in den Blättern. Innere Knoten enthalten nur einen Zeiger. Zugriff immer bis zum Blattkonten. Die Blattknoten ist verketten. Dies führt zu einem höheren „Fan-out“, da die Datenmenge in den Knoten kleiner ist und zudem zu einer kleiner Tiefe, daher weniger I/O. B+ ist vorherrschend in DBS.

### Clustered und unclustered Indexes

Dieser Teil kann des Vorlesungsfolien entnommen werden.

### Hash



Eine Hash-Funktion (h) ordnet Keys zu Records von 0 bis N-1 und macht dabei Blöcke mit Schüsseln. NULL-Werte sind nicht zugelassen. Das Ganze funktioniert nach der Key-Value Semantik. Es kann da aber zu Overflows kommen, was mit Overflow Chains gelöst werden kann.

Gut für Equal-Serach oder für optimierte Memory-Strukturen.

#### Performance

Das Beste für die Beantwortung von Punktabfragen (Point Queries), vorausgesetzt, es gibt keine Überlaufketten. Gut für Multipoint Queries, aber nutzlos für Range, Prefix oder External Queries.

Muss reorganisiert werden (Drop / Add oder Reorganisation verwenden), wenn es eine erhebliche Überlaufverkettung gibt. Die Vermeidung von Überlauf kann dazu führen, dass der Hashraum nicht gut genutzt wird. Die Größe der Hash-Struktur hängt nicht mit der Größe eines Schlüssels zusammen, da Hash-Funktionen Schlüssel zu Orten oder Seitenkennungen zurückgeben. Hash-Funktionen dauern länger, um auf einem langen Schlüssel auszuführen.

**Postgres**: Hash-Index-Nutzung wird wegen fehlender Unterstützung für WAL-Protokollierung entmutigt

### Bitmap

Ordnet Attributwerte als Bitmuster.

**Eigenschaften**  
Geeignet für Attribute mit wenigen diskreten Werten, d.h. Wertebereiche mit geringer Kardinalität (ca. 1 Prozent der Datenmenge). Schnelle Anfragen, aber langsam bei Modifikationen.

Einsatz vor allem im Data-Warehouse.

|  |  |  |
| --- | --- | --- |
|  |  |  |

### Indexe wählen

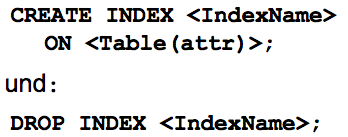
**Kriterien**

* Welche Indexe werden vom DBMS angeboten?
* Wie ist die Tabellengrösse?
* Werden Daten oft gelöscht oder neu eingetragen?
* Welche Attribute werden in Queries (Selektion/Join) benötigt?
* Was sind die Wertebereiche der Attribute?

Index-Kandidaten sind Attribute, die Schlüssel sind, oder an JOINs teilnehmen, oder in Gleichheits- oder Bereichsbedingungen vorkommen oder in Sortierungs und Group By Bedingungen vorkommen.

### Indexe erstellen

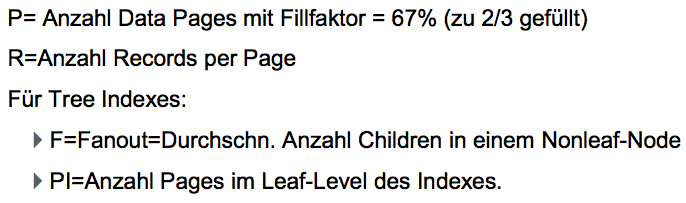
Da gibt es grosse Unterschiede in den heutigen DBMS-Produkten. Es ist keine Spezifikation in SQL vorhanden, wie man Index anlegen oder löschen soll.

Folgende Syntax hat sich eingebürgert. Der Index erhält also einen Namen, über den er auch wieder gelöscht werden kann.

**Unique Indexe** werden automatisch erstellt bei einem PK-Attribut oder bei einem Attribut mit UNIQUE-Constraint. Für einen alternativen Schlüssel muss man es manuell machen. **Non unique Indexes** müssen manuell erstellt werden.

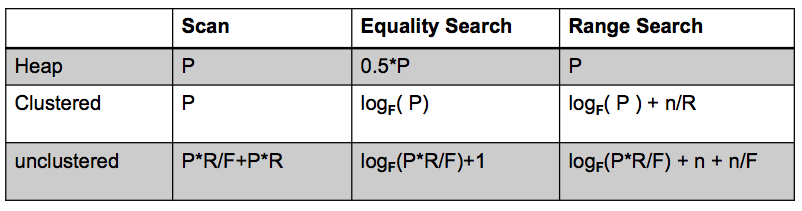
### Kostenmodell

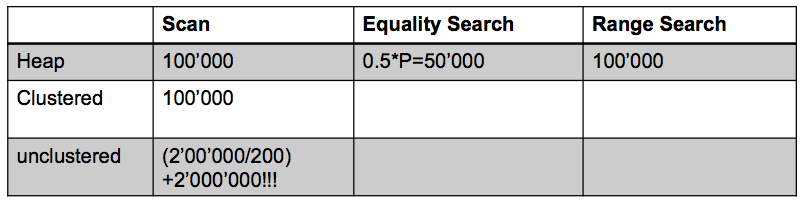
Ziel ist das Abschätzen der Anzahl I/O Operationen für eine Anfrage.

**Definition  
**

**Vergleich der Anzahl I/O’s**

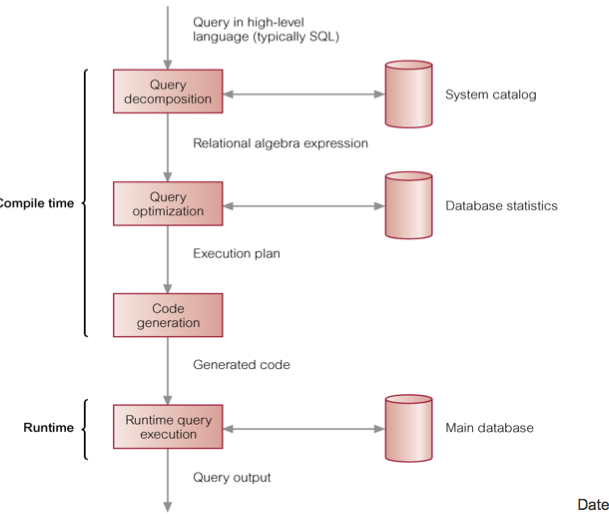
P = Anzahl Pages, R = Anzahl Records per Heap-Page, F = Index Fan Out, N = Anzahl Tupels, die eine Suchbedingung erfüllen.

****

****

## Query Processing

### Einführung

Die Query Engine parst SQL-Anweisung und wandelt diese in einen Query Tree um.

Der Query Tree wird allenfalls transformiert (Query Rewriting) in äquivalenten Query Tree.

Der Optimizier generiert Ausführungspläne (Excecuiton Plan) und wählt den optimalsten (Cost Based Optimizier) basierend auf Statistiken über die Datenverteilung.

Zur Laufzeit werden die kompilierten Ausführungspläne mit den aktuellen Variablen gebunden und ausgeführt.

**Beispiele von Ausführungsplänen**

|  |  |
| --- | --- |
|  |  |

### Queries auf einer Tabelle

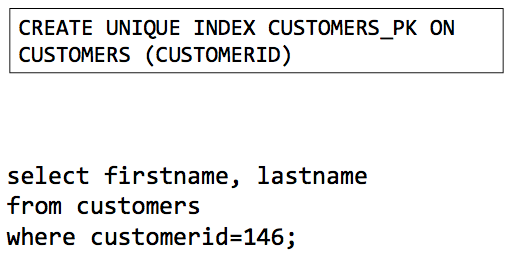
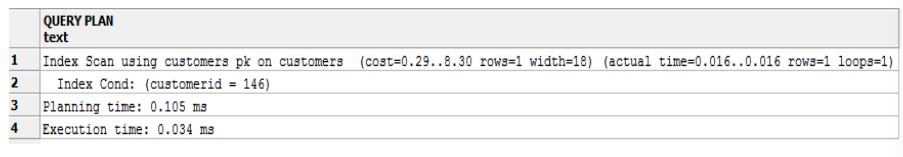
Jede Daten führt bei der Ausführung einer Query verschiedene Operationen aus. Diese sind folgended aufgestellt.

|  |  |
| --- | --- |
| **Oracle**   * Table Scan * Index Range Scan * Index Unique Scan * Table Access by RowID * Sort (in-memory) * Hash (In-Memory Hasing eines Tuples) * Filter * View | **PostgreSQL**   * Seq Scan * Index Scan * Index Only Scan * Bitmal Index Scan / Bitmap Heap Scan / Recheck Cond * Sort * HashAggregate * GroupAggregate |

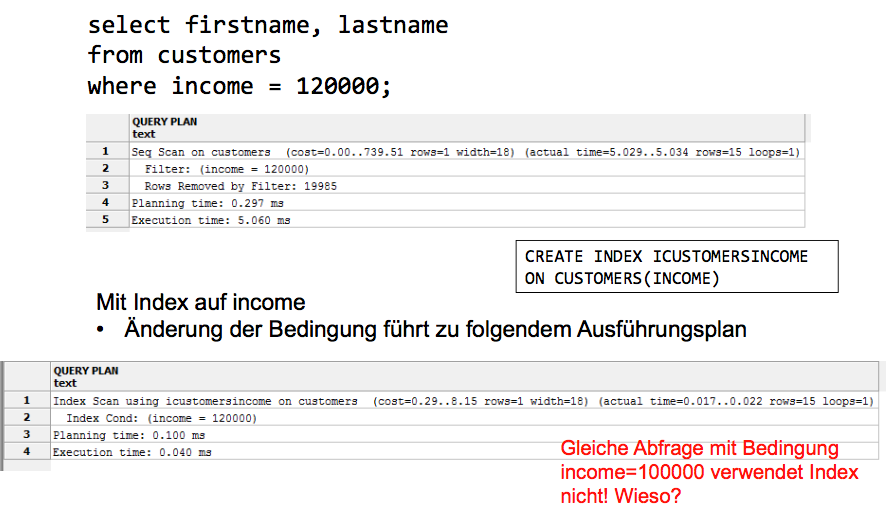
#### Arten von Queries

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

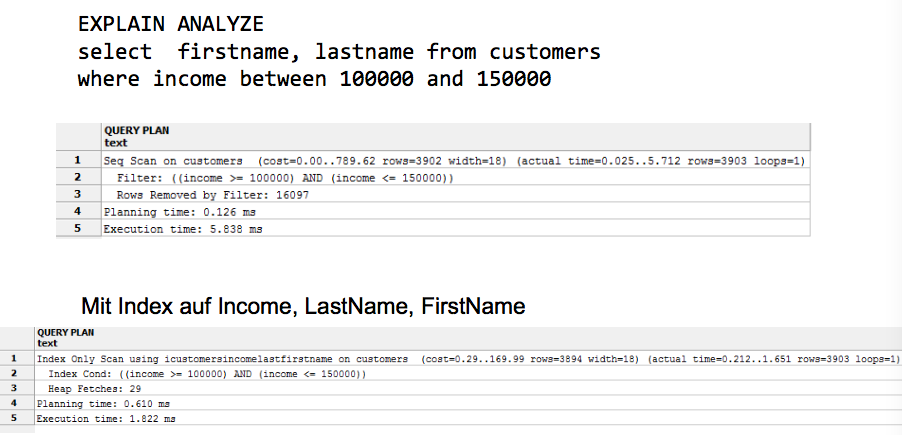
##### Point Query

##### Multi-Point Query



##### Range Query



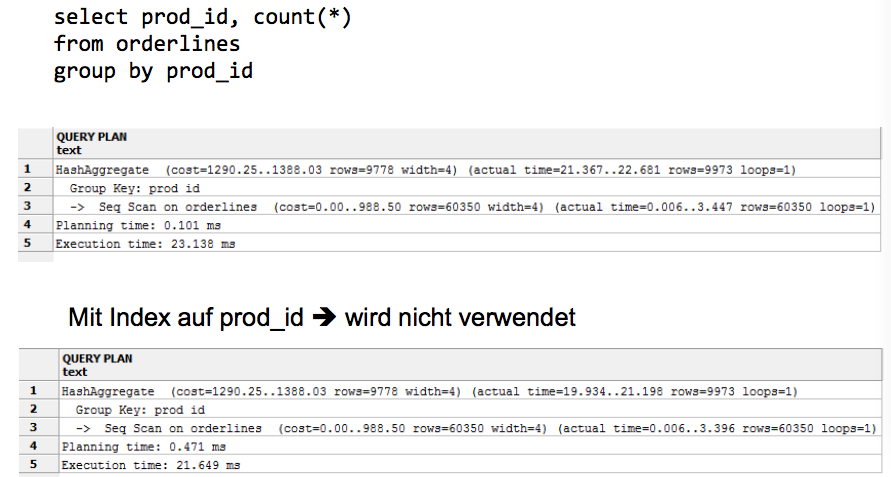
##### Order Query



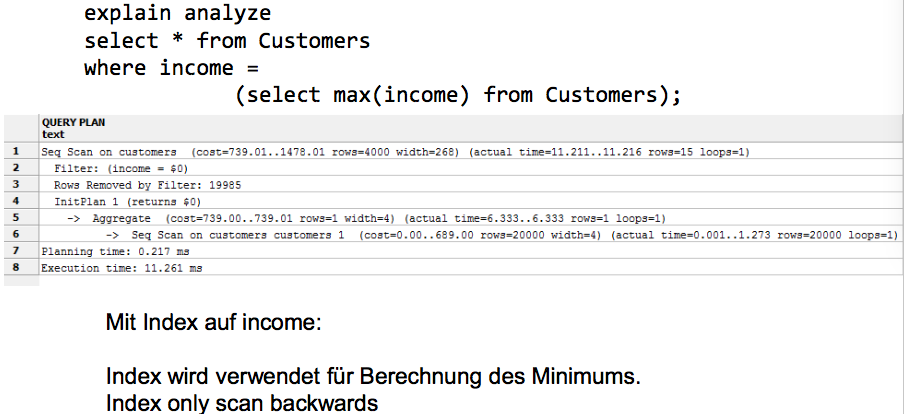
##### Distinct Query



##### Group by Query



##### External Query



#### Clusterd Index in PostgreSQL

In PG werden keine clustered Indexes unterstützt. PG unterstützt cluster-Befehl um Daten sortiert abzulegen, dazu werden die bestehenden Daten kopiert. Leider weiss der PG-Optimizier nichts von den clustering-Anweisungen und kann daher keine optimierten Ausführungspläne generieren.

### Joins (Teil 2)

# Verteilte Datenbanksysteme

## Einführung

Ein VDBMS (Verteiltes DBMS) besteht aus kooperierenden DBMS, welche verteilt auf verschiedenen Computer (Knoten, Site) eines Netzwerks laufen. Jeder Knoten hat eine autonome Verarbeitungsfunktionalität und kann lokale Applikationen ausführen. Jeder Knoten partizipiert an globalen Applikationen, welche Datenzugriff auf die verschiedenen Knoten benötigen.

### Anforderungen

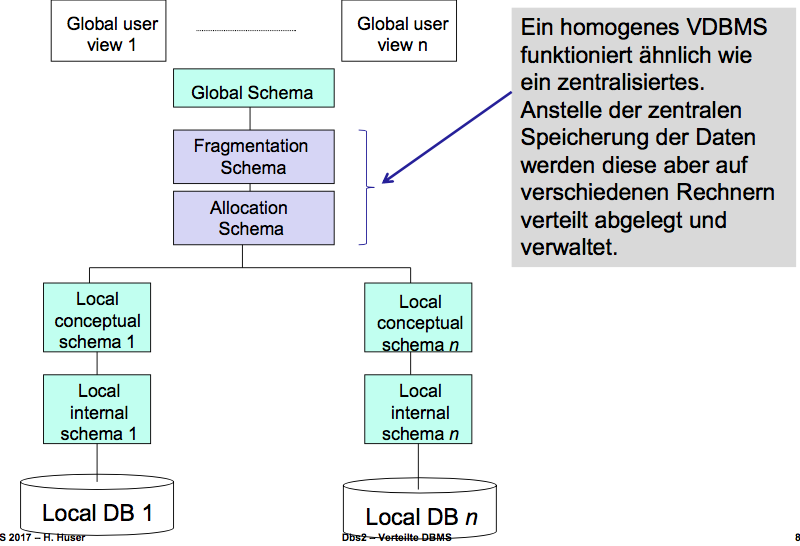
**Datenverteilungs-Transparenz**  
Die Benutzer sollen Queries formulieren können ohne auf die konkrete Verteilung der Daten zu achten.

**Atomarität von verteilten Transaktionen**  
Die Benutzer sollen Transaktionen schreiben, die auf Daten in den einzelnen Knoten zugreifen ohne sich um die Verteilung zu kümmern. Alle Änderungen auf den einzelnen Knoten sollen entweder festgeschrieben werden (commit) oder zurückgesetzt (abort/rollback) werden.

## Heterogene und homogene VDBMS

### Homogenes verteiltes DBMS

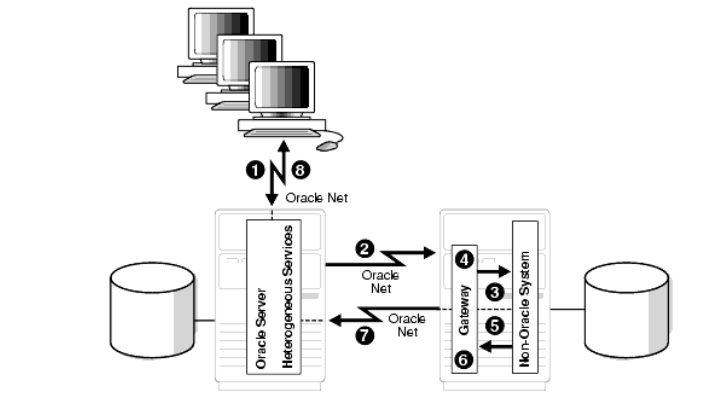
Hier haben alle Knoten die identische Software. Alle Knoten wissen voneinander und arbeiten zusammen um Benutzeranfragen zu verarbeiten. Es erscheint gegenüber dem Benutzer als ein System.



### Heterogenes verteiltes DBMS

Hier können alle Knoten unterschiedliche Software und Schemas aufweisen. Die Schemaunterschiede sind das Hauptproblem für die Ausführung verteilter Queries, während die Softwareunterschiede das Hauptproblem für die Ausführung verteilter Transaktionen ist.

Die Konten wissen evtl. nichts voneinander und können nunr beschränkte Funktionalitäten für die Kooperation anbieten.

Dazu sendet die Clientapplikation ein Query über das Oracle Net um Oracle DB Server. Der Gateway konvertiert das Query in ein Statement, dass auch nicht Oracledatenbanken verstehen. Der Gateway loggt sich dann an der anderen DB ein und erhält die Daten aus dem Query. Der Gateway konvertiert die Daten in Oracle um und gibt Sie an den Oracle Server weiter, welcher es dann an die Clientapplikation weitergibt.

## Verteilte Daten

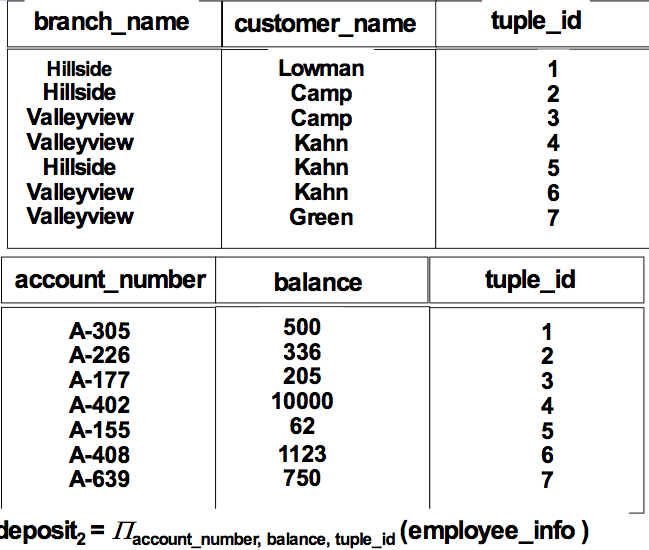
### Fragmentierung

Unterteilung einer Relation r in Fragmente r1, r2.. rm, welche genügend Informationen enthalten um die Relation r zu rekonstruieren.

#### Horizontale Fragmentierung

Jedes Tupel wird auf ein oder mehrere Gragmente (Shard) aufgeteilt. Dies wird auch Sharding genannt. 🡺 Wichtig bei NoSQL DBMS.

#### Vertikale Fragmentierung

Das Schema einer Relation r ist in mehrere kleinere Schematas aufgeteilt. Alle Schematas müssen einen gemeinsamen Schlüssel enthalten um die sogennante „lossless join“ Bedingung zu erfüllen.

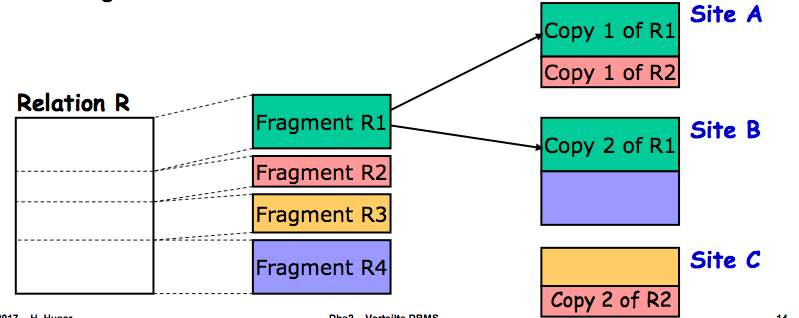
### Replikation

Eine Relation (oder ein Fragment einer Relation) ist repliziert, wenn Sie (es) redundant in mehreren Knoten gespeichert ist.

**Vorteile der Replikation**  
Verfügbarkeit (auch bei Ausfall eines Repliaks) zudem kann Parallelität bei der Ausführung von Queries genutzt werden. Zuletzt findet ein reduzierter Datentransfer beim lokalen Zugriff auf die replizierten Daten statt.

**Nachteile der Replikation**  
Die Update-Kosten sind um einiges höher, da jedes Replikat aktualisiert werden muss. Zudem entsteht eine komplexere Synchronisation (Die Änderungen können zu temporär inkonsistenten Daten führen).

### Transparenz der Datenverteilung

Die Daten-Verteilung sollte für den Benutzer möglichst transparent (nicht sichtbar) bleiben. Das heisst Fragmentation transparency, Replication transparency und Location transparency.

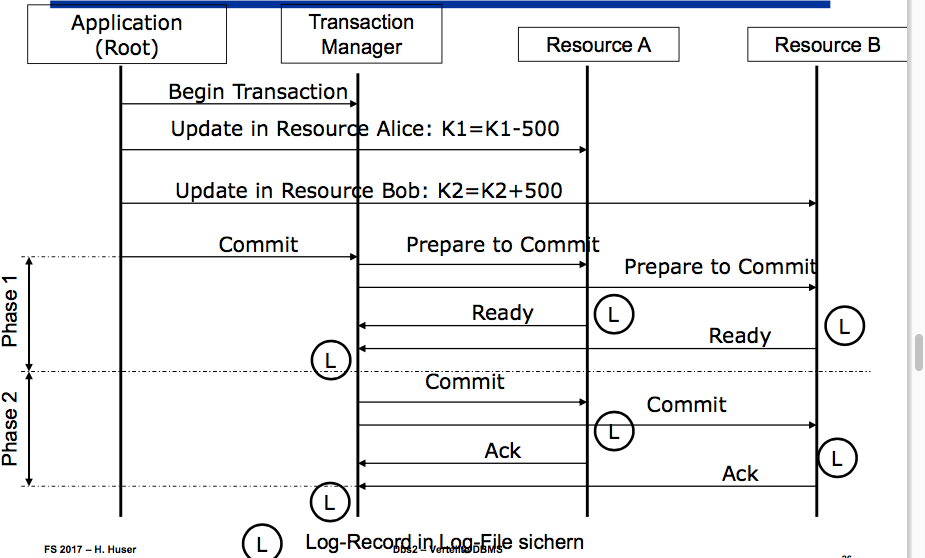
Der Benutzer sieht eine globale Sicht, er kenn die Replikation der Fragmente nicht. Die Queries werden auf der Relation definiert, nicht auf den Fragmenten.

## Verteilte Transaktionen

Eine Transaktion auf einer Datenbank ist immer lokal zu einer Session. Was passiert, wenn eine Transaktion mehrere lokale Transaktionen umfasst? Dann kommt das Zwei-Phasen Commit Protokoll zum Einsatz, welches eine atomare Ausführung von verteilten Transaktionen erlaubt.

### Two Phase Commit (2 PC)

Dieses Protokoll sicher die Atomarität von verteilten Transaktionen. Es besteht aus folgenden Komponenten:

* **Transaction Manager**: koordiniert Transaktionen über mehrere Resource Managers
* **Resource Manager** (DBMS): unterstützt lokale Transaktionen
* **Applikation**: steuert Transaktionen

#### Phasen in 2PC

**Phase 1 “prepare to commit”**  
TM fragt alle RM an, ob sie in der Lage sind die T mit einem Commit abzuschließen. Im positiven Fall gehen die RM in den Zustand „prepared“.

**Phase 2 „commit“**  
Hat der TM von allen RM’s ein READY erhalten, kann der TM ein COMMIT an alle RM schicken mit der Aufforderung die Änderungen von T lokale festzuschreiben. Antwortet einer der RM mit Abort oder gar nicht innerhalb einer bestimmten Zeit (timeout), schickt der TM ein ABORT an alle RM’s und diese machen die Änderungen der Transaktion rückgängig. Haben die Agenten ihre lokale EOT-Behandlung abgeschlossen, schicken Sie eine ACK-Nachricht an den Korrdinator.

#### Transkationsmanager

Der zentrale TM muss sicherstellen, dass alle RM‘s einer Transaktion die lokale Transaktion entweder mit Commit oder mit Rollback abschliessen. Es darf nicht vorkommen, dass ein RM die Transaktion mit Commit und ein anderer RM die Transaktion mit Rollback zurücksetzt. Ausserdem muss der TM mit allen Fehlersituationen umgehen können, die durch das Kommunikationssystem verursacht werden.

|  |  |
| --- | --- |
| **Zustandsübergang 2PC – TM** | **Zustandsübergang 2PC – RM** |

#### Fehlersituationen

**Absturz eines Transaktionsmanagers**  
Bei einem Absturz vor dem Senden einer COMMIT-Nachricht findet ein Rückgängigmachen der Transaktion durch Versenden einer ABORT-Nachricht statt.

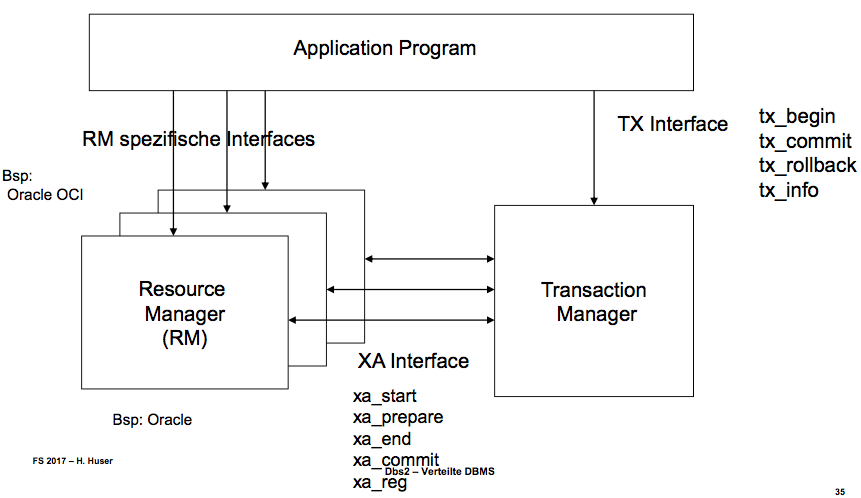
Bei einem Absturz nachdem die RM ein Ready mitgeteilt haben führt zur Blockierung der RM. Dies stellt das Hauptproblem des 2PC-Protokolls dar, da dadurch die Verfügbarkeit des RM bezüglich anderer globaler und lokaler Transaktionen drastisch eingeschränkt ist.

**Absturz eines Resource Mangers**Antwortet ein RM innerhalb eines Timeout-Intervalls nicht auf die PREPARE-Nachricht, gilt der RM als abgestürzt; der Koordinator bricht die Transaktion ab und schickt eine ABORT-Nachricht an alle Agenten. Der abgestürzte Agent schaut beim Wiederanlauf in seine Log-Datei.

* Wenn kein Ready-Eintrag bzgl. Transaktion T 🡺 RM führt ein abort durch und teilt dies dem Koordinator mit (FAILED Nachricht)
* Wenn ein Ready-Eintrag aber kein Commit-Eintrag vorhanden ist 🡺 RM fragt TM, was aus Transaktion T geworden ist. TM teilt COMMIT oder ABORT mit, was beim RM zu einem Redo oder Undo der Transaktion führt.
* Wenn ein Commit-Eintrag vorhanden ist 🡺 RM weiss ohne Nachfragen, dass ein (lokales) Redo der Transaktion nötig ist.

Nachricht verloren  
PREPARE-Nachricht des TM an einen RM geht verloren oder READY-(oder FAILED-)Nachricht eines RM geht verloren 🡺 nach Timeout-Intervall geht TM davon aus, dass betreffender RM nicht funktionsfähig ist und sendet ABORT-Nachricht an alle RM (Transaktion gescheitert). RM erhält im Zustand Ready keine Nachricht vom Koordinator 🡺 RM ist blockiert, bis COMMIT- oder ABORT-Nachricht vom TM kommt, da der RM nicht selbst entscheiden kann (deshalb schickt der RM eine „Erinnerung“ an den TM).

#### X/Open XA (DTM)

 Standardisierte Schnittstellen zwischen Komponenten in verteilten Systemem, Wird von praktisch allen kommerziellen Systemen unterstützt.

Die JTA (Java Transaction API) ist eine Schnittstelle basiered auf dem XA. Es gibt zudem mehrere weitere Implementationen, welche darauf basieren: JOTM (Java Open Transaction Manager) oder JBoss.

#### Wertung

Es hat sich bewährt, von allen Herstellern (als RM) wird es unterstützt. Es ist sicher, garantiert Atomarität und Konsistenz. Zum Nachteil ist es aber relativ langsam und skaliert schlecht. Zudem kann es zu Blockaden kommen.

Daher nur Einsetzen wenn Atomarität und Konsistenz wirklich notwendig sind.

## Synchronisation in VDBMS

### Serialisierbarkeit

## Lokale Serialisierbarkeit an jeder der an den Transaktionen beteiligten Stationen reicht nicht aus.= 🡺 deshalb muss man bei der Mehrbenutzersynchronisation auf globaler Serialisierbarkeit bestehen.

Dabei wird das 2 PL auf verteilte Systeme adaptiert. Locken der Objekte, dann das Lesen/Modifizieren und abschliessend die Freigabe des Locks. Jeder Knoten partizipiert in der Ausführung des Commit-Protokolls um globale Serialisierbarkeit zu gewähren.

#### Globaler Lock-Manager

Das VDBMS unterhält einen globalen Lockmanager z.B. auf dem Knoten Si. Wenn eine Transaktion T ein Datum sperren möchte, dann fragt T den globalen Lockmanager an. Dieser entscheidet sofort, ob der Lock gewährt werden kann. Falls ja, sendet der Lockmanager eine Meldung an den anfragenden Knoten. Falls nein, wird die Anfrage verzögert bis der Lock gewährt werden kann.

**Vorteil**  
Die Deadlock-Erkennung kann wie im Ein-Rechner-Fall vorgenommen werden.

**Nachteil**  
Jede Sperranforderung einer Transaktion, die nicht auf dem zentralen Knoten läuft, verursacht eine Nachricht, auf die die Transaktion synchron warten muss 🡺 schlechter Durchsatz und Antwortzeit. Der zentrale Knoten ist aber der Engpass für Leistung und Verfügbarkeit („Singe point of failure“). Zudem gibt es keine Knotenautonomiere (ohne Lockmanager läuft nichts).

#### Lokaler Lock-Manager

Der verteilte Algorithmus greift auf die lokalen Lock-Manger auf den einzelnen Knoten zu. Der lokale Lock-Manager kontrolliert den Zugriff auf die lokalen Data-Items. Locks werden im Rahmen des 2PC-Protokolls allokiert und freigegeben.

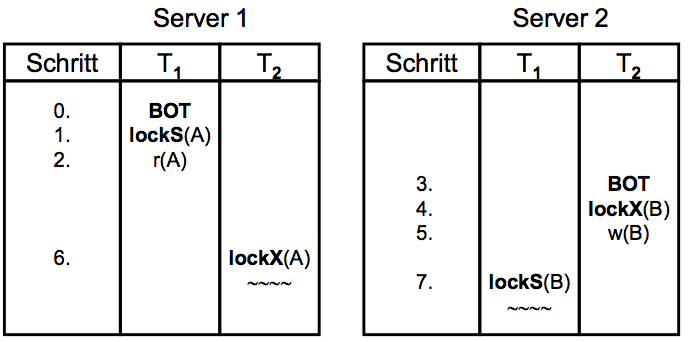
**Vorteil**  
Arbeit ist verteilt und ist robuster gegen Ausfälle (failures)

**Nachteil**  
Die Deadlock-Detektion ist schwierig. Die Lockmanager kooperieren dazu mit einem verteilten Protokoll für die Detektion. Die Performance kann stark beeinträchtigt werden.

#### Zeitstempel basierte Synchronisation

Jeder Transaktion bekommt einen global eindeutigen Zeitstempel. Das Hauptproblem besteht bei der Generierung einer global eindeutigen Zeit. Jeder Knoten geniert dazu einen lokalen eindeutigen Zeitstempel. Der globale eindeutige Zeitstempel wird durch Konkatentation des lokalen Zeitstempels mit der Rechner-ID generiert.

#### Deadlocks in VDBMS

**Erkennen mit Timeouts**  
Die betreffende Transkation wird dann einfach zurückgesetzt und erneut gestartet 🡺 einfach zu realisieren. Das Problem ist die richtige Wahl des Timeout-Intervalls. Wenn zu lang, kommt es zu einer schlechten Ausnutzung der Systemressourcen, wenn zu kurz findet eine Deadlock-Erkennung statt, wo gar keine Verklemmung vorliegt.

**Zentralisierte Deadlock-Erkennung**  
Die Stationen melden lokal vorliegende Wartebeziehungen an neutralen Knoten, der daraus globalen Wartegraphen aufgebaut (Zyklus im Graphen ⬄ Deadlock) 🡺 sichere Lösung. Nachteilig ist, dass daraus ein hoher Aufwand in Form von vielen Nachrichten entsteht. Zudem kommt es zur Entstehung von Phantom-Deadlocks durch „Überholen“ von Nachrichten im Kommunikationssystem.

|  |  |  |
| --- | --- | --- |
| **Lokal** | **Lokal** | **Global** |

**Deadlock-Vermeidung**  
*Durch optimistische Mehrbenutzersynchronisation*, dass heisst nach Abschluss der Transaktionsbearbeitung wird eine Validierung durchgeführt.

*Durch Zeitstempel-basierende Synchronisation*, also der Zuordnung eines Lese-/Schreib-Stempels zu jedem Datum. Es wird dann entschieden, ob beabsichtigte Operation durchgeführt werden kann ohne Serialisierbarkeit zu verletzten oder ob Transaktion abgebrochen wird (abort).

## Distributed Query Processing

In einem zentralisierten System werden die Konsten der Query durch die Anzahl Disk-I/Os bestimmt. In einem verteilten System werden die Kosten der Query durch die Anzahl Netzwerk I/O und durch den Performance-Gewinn der Parallelität bestimmt.

### Verteilte Anfragen

### Beispiel

#### Horizontale Fragmentierung

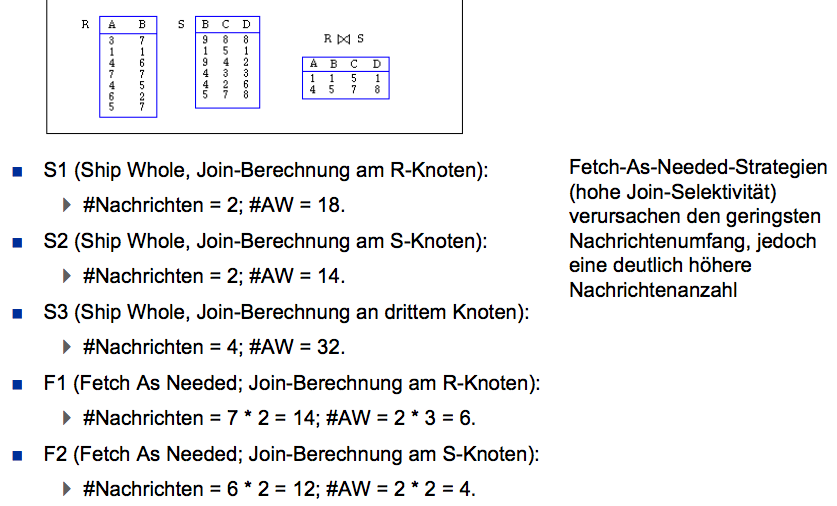
Die Tuples mit rating < 5 in Zürich und die Tuples mit rating >= 5 in Genf. Bei der Ausführung wird die Query in Genf und Zurück ausgeführt und das Resultat anschliessend zusammengeführt. Wenn die WHERE-Klausel S.rating > 6 hiesse, dann wäre nur der Knoten Genf beteiligt.

#### Vertikale Fragmentierung

Tuples mit tid, sid und rating in Zürich, Tuples mit tid, sname und age in Genf. Die Relation muss hier über Tupel-Identifier tid durch Join rekonstruiert werden. Dann die Auswertung der Query.

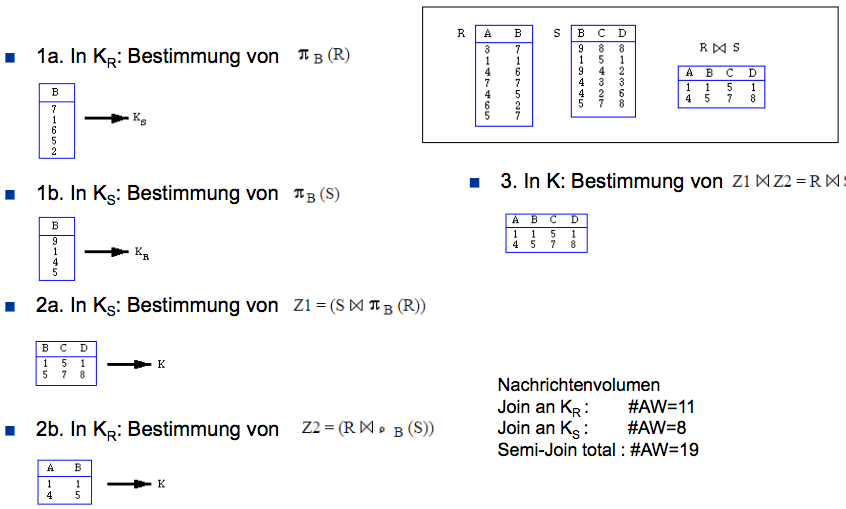
### Verteilte JOINS

#### Einfache Joins

R ist in der Site N1, S in der Site N2 gespeichert. Für jede Anfrage vom Knoten Ni muss das System die Resultate auf Ni liefern. Dabei wird der Join auf eine Konten ausgeführt. Bei „Ship Whole“ wird die Relation vollständig an den Join-Knoten übertragen (Minimale Nachrichtenanzahl, hohes Datenvolumen). Bei „Fetch as needed“ wird für jedes Tupel der lokal vorliegenden Relation der Wert des Join-Attributes an den anderen Rechner geschickt, welcher das passende Tupel überträgt (Hohe Nachrichtenanzahl, minimales Datenvolumen).

#### Semi Join

Reduktion des Overhead des Fetch-As-Needed-Ansatzes. Überträge sämliche Verbundattributwerte der ersten Relation in einer Nachricht an den Konten der zweiten Relation. Daraufhin können alle Verbundpartner in einer Nachricht an den esten Knoten zurückgesendet werden, wo dann die Join-Bindung erfolgt.

**Beispiel  
**