

Datenbanksysteme

Kap 6: Weitere Datenbank-Objekte

Wichtige fortgeschrittene DB-Objekte

- System Catalog/Data Dictionary
 - Vom DBS gespeicherte Strukturinformationen
- Sequence
 - Generiert eindeutige Werte
 - Nicht in SQL2 spezifiziert, aber von fast allen DBS unterstützt, wobei Syntax der Verwendung variiert
- Schema
 - Namespaces zum Trennen von Usern/Anwendungen
 - In SQL2 gefordert, aber ungenau spezifiziert
 - DBS-spezifische Unterschiede im Detail
- View
 - Select-Statement als virtuelle Tabelle wiederverwenden
 - In SQL2 spezifiziert
 - Wesentlicher Bestandteil aller relationalen Datenbanksysteme

System Catalog/Data Dictionary

- Strukturinformationen werden vom DBS in Tabellen gespeichert
- Sammlung dieser Tabellen heißt System Catalog oder Data Dictionary
- Beispiel: Tabelle `pg_attribute`

pg_attribute: PostgreSQL column meta data	
<i>attrelid</i>	The table this column belongs to (references <i>pg_class.oid</i>)
<i>attname</i>	Column name
<i>atttypid</i>	The data type of this column (references <i>pg_type.oid</i>)
...	...

System Catalog in PostgreSQL

- System Catalog in PostgreSQL umfasst folgende Tabellen

Catalog Name	Purpose
<i>pg_attribute</i>	table columns (“attributes” , “fields”)
<i>pg_class</i>	tables, indexes, sequences (“relations”)
<i>pg_database</i>	databases within this database cluster
<i>pg_group</i>	groups of database users
<i>pg_index</i>	additional index information
<i>pg_relcheck</i>	check constraints
<i>pg_trigger</i>	triggers
<i>pg_type</i>	data types
<i>pg_user</i>	database users

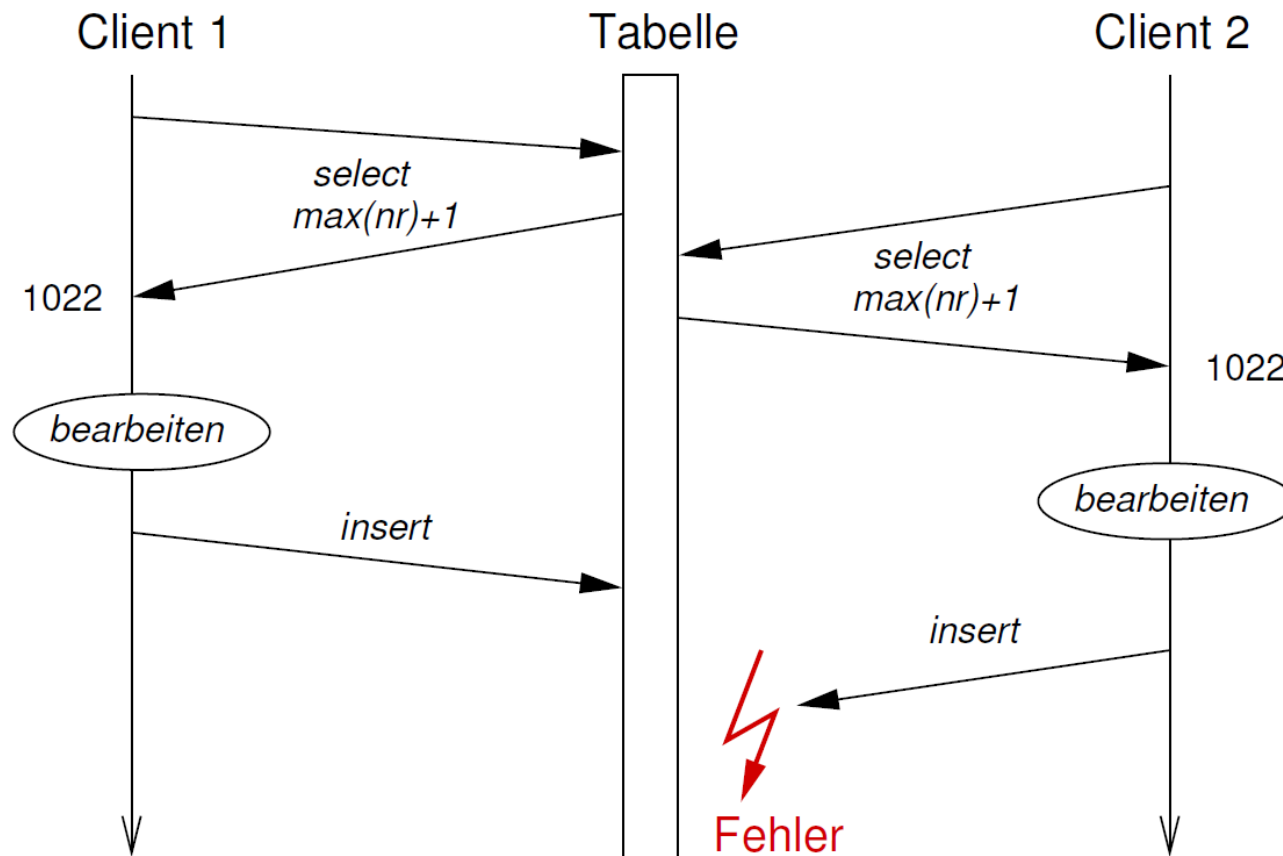
- In psql können Beschreibungen mit \d abgefragt werden
- \set ECHO_HIDDEN (oder psql -E) gibt Abfragen mit aus

Sequence

- Was ist eine Sequence?
 - Sequence ist ein Zähler
 - Wesentliche Eigenschaft: einmal vergebener Wert wird nicht nochmal vergeben (auch nicht in anderen Transaktionen)
 - Sequence-Werte sind über Transaktionsgrenzen hinweg eindeutig
- Anwendungsgebiete
 - Automatische Generierung Primärschlüsselwerte
 - Erzeugung eindeutiger Namen für temporäre Tabellen
 - Oft besser:
 - Verwendung von `CREATE LOCAL TEMPORARY TABLE`

Naiver Ansatz für Primärschlüsselerzeugung

- Clients benötigen Primärschlüssel für neuen Datensatz
→ bisherigen Maximalwert ermitteln und inkrementieren

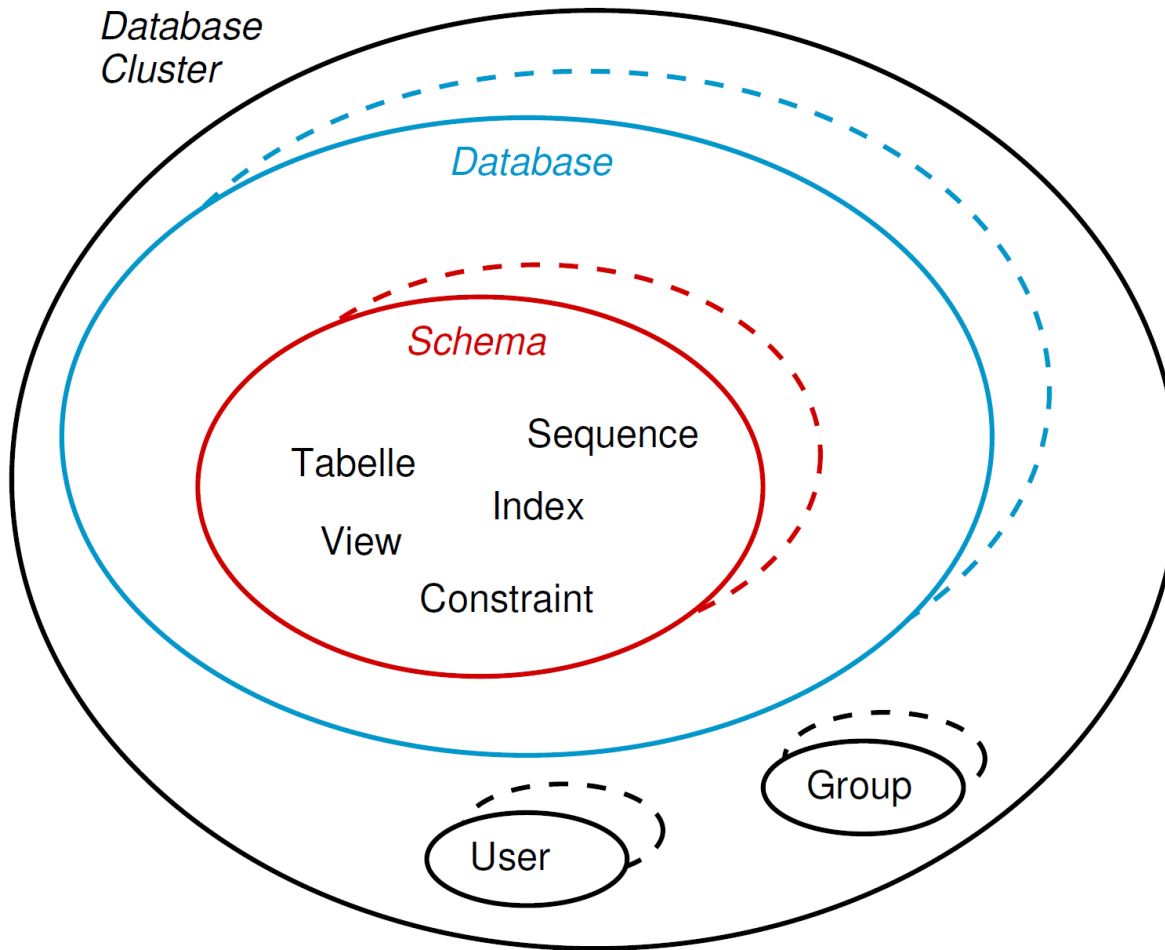


Verwendung von Sequence

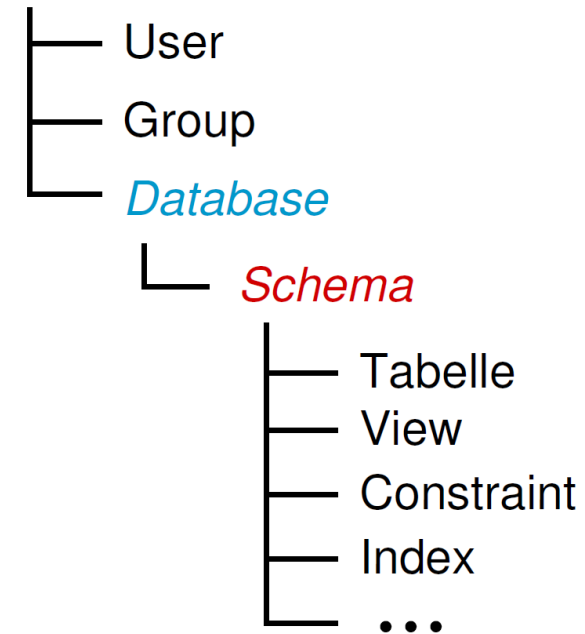
- Anlegen der Sequenz
 - `CREATE SEQUENCE seq_person
START 100000 INCREMENT 1;`
- Verwendung als Default-Wert für Primärschlüssel
 - `CREATE TABLE person (
nr numeric(6) DEFAULT nextval('seq_person'),
name varchar(30),
/* ... */
PRIMARY KEY (nr)
);`
- Bemerkung:
 - PostgreSQL Datentyp SERIAL macht das automatisch

Schema

- Hierarchieebenen einer Datenbank



Database Cluster



Datenbank-Cluster und Datenbank

- Datenbank-Cluster
 - Sammlung mehrerer Datenbanken, die von einem Datenbank-Serverprozess verwaltet werden
 - User und Gruppen auf Clusterebene, aber einstellbar wer auf welche Datenbank zugreifen darf
 - (PostgreSQL: pg hba.conf, Oracle: grant/revoke connect)
- Datenbank
 - Sammlung von Tabellen, Views, Constraints, Indizes, ..., die in Schemas zusammengefasst sind
 - Eine Verbindung zum DB-Server wird immer mit genau einer Datenbank hergestellt
 - Datenbankübergreifende SQL-Statements sind nach SQL2 nicht möglich, können aber in Oracle mit Datenbank-Links emuliert werden (auch über Clustergrenzen hinweg!)

- Was ist ein Schema?
 - Ein Schema ist ein Namespace
 - Derselbe Tabellename kann parallel in verschiedenen Schemas verwendet werden
 - Jede Tabelle ist genau einem Schema zugeordnet
 - Angesprochen wird Tabelle mit `schemaname.tabellename`
 - User kann in derselben Sitzung (Datenbank-Verbindung) Objekte aus mehreren Schemas ansprechen
 - Auf Schemas können Zugriffsrechte erteilt werden
- Wozu braucht man Schemas?
 - Mehrere User konfliktfrei auf derselben Datenbank
 - Mehrere Applikationen auf derselben Datenbank
 - Logische Gruppierung von Objekten mit leichter Verwaltung

Benutzung von Schemas

- Schemaanlage
 - `CREATE SCHEMA schemaname;`
 - Per Default vorhanden: Schema `public`
- Tabellenanlage
 - `CREATE TABLE [schemaname.]tabellenname (...);`
 - Ohne `schemaname` wird Tabelle in erstem (existierenden) Schema aus Suchpfad angelegt
- Schema Suchpfad
 - Unqualifizierte Tabellennamen werden im Schema Suchpfad gesucht
 - Wie Suchpfad gesetzt wird, ist systemspezifisch
 - Typischer Defaultwert: `username, public`

- Typische Konfiguration:
 - Jeder User, der Tabellen anlegt (das ist normalerweise pro Applikation nur ein einziger User!) hat ein eigenes Schema mit seiner Userid als Namen
 - Alle Tabellen der Applikation in diesem Schema anlegen;
 - Suchpfad Applikationsaccount beginnt mit Usernamen
 - Endanwender (andere Accounts!) müssen Tabellen qualifizieren und dürfen DML aber kein DDL ausführen
- Emulation schemalose Datenbank:
 - Erforderlich zwecks Kompatibilität zu DBS, die keine Schemas unterstützen (z.B. PostgreSQL vor Version 7.3)
 - Keine expliziten Schemas anlegen und nur das Schema `public` benutzen (→ alle User im selben Namespace)

- Was ist ein View?
 - "Virtuelle" Tabelle, deren Inhalt dynamisch über eine Anfrage (rel. Algebra oder SQL) berechnet wird
 - Im relationalen Modell als *abgeleitete Relation* bezeichnet, im Gegensatz zu *Basisrelation* (Tabelle)
- Verhält sich aus Anwendersicht wie Tabelle:
 - Abfrage mit SELECT
 - Explizite Rechtevergabe mit GRANT/REVOKE
 - Aber: Änderung (INSERT, UPDATE, DELETE) im allg. nicht möglich

Definition eines Views

produkt

pnr#	name	preis	einfuehrung	auslauf
------	------	-------	-------------	---------



select ... from produkt ...

produkt_aktuell

pnr#	name	preis	einfuehrung
------	------	-------	-------------

- View, der nur aktuelle Produkte enthält:

```
CREATE VIEW produkt_aktuell AS
  SELECT pnr,name,preis,einfuehrung
  FROM produkt
  WHERE auslauf > current_date
  OR auslauf IS NULL;
```

Angabe der Attributnamen im View

- Implizit über Liste selektierter Attribute:
 - `CREATE VIEW produkt_aktuell AS
SELECT pnr, name AS produkt, ...
FROM produkt WHERE ...`
- Explizite Angabe hinter View-Namen:
 - `CREATE VIEW produkt_aktuell (pnr, produkt, ...)
AS
SELECT pnr, name, ...
FROM produkt WHERE ...`

Umsetzung von Views

Bei der Umsetzung von Abfragen über Views durch das DBS gibt es zwei verschiedene Ansätze:

- SQL Substitution
 - SQL kennt den Tabellenkonstruktor (SELECT ...) *name*
 - Ersetze einfach in Abfrage vorkommende Views durch einen entsprechenden Tabellenkonstruktor mit der View-Definition
 - Vorteil: Implementierungsaufwand gering
- Materialized Views
 - Erzeuge Tabelle, die einen Cache der View-Abfrage enthält
 - Cache muss dann auf Aktualität geprüft und ggf. neu berechnet werden
 - Vorteil: Performancegewinn

Wozu sind Views gut?

- Kapselung und Wiederverwendung komplexer Queries
 - Anwender braucht Abfrage nicht zu kennen
 - Abfrage kann geändert werden, ohne Applikation anzupassen
 - Evtl. bessere Performance ("Materialized Views")
- Einschränkung von Zugriffsrechten
 - Normalerweise Rechte über Zugriffsfrontend gesteuert
 - Wird kein anwendungsspezifisches Frontend verwendet (z.B. DB-Frontends aus Office-Paketen), trotzdem Rechtebeschränkung mit Views möglich
- Vermeidung Redundanzen
 - Abgeleitete Attribute können dynamisch berechnet werden

Rechtebeschränkung auf Views

- Problem:
 - Anwender benutzt Abfrage-Frontend, das keine Rechtebeschränkung ermöglicht (z.B. MS Access, SQL-Prompt)
- Lösung:
 - Richte Views ein, deren Select-Klausel das Rechteprofil des Anwenders berücksichtigen
 - Richte für Anwender eigenen Datenbank-User ein
 - Gib diesem User nur das Zugriffsrecht auf die Views und entziehe ihm den Zugriff auf alle anderen Tabellen

Beispiel: Rechtebeschränkung

- Ziel: System-Catalog, in dem jeder nur seine eigenen Tabellen sieht
- Mögliche Lösung:
 - Tabelle `all_tables` (`tblid`, `name`, `owner`,...) enthält Tabellen aller User
 - Definiere View, in dem jeder nur seine Tabellen sieht:

```
CREATE VIEW user_tables AS  
  SELECT * FROM all_tables  
  WHERE owner = current_user;
```
- Bemerkungen:
 - `current_user` ist die SQL2-Funktion für die aktuelle Benutzerkennung
 - obwohl alle auf denselben View zugreifen, sieht jeder User andere Daten

Data Dictionary in Oracle

- Vom System definierte Views, die Benutzerrechte berücksichtigen
 - Präfix USER: eigene Objekte
 - Präfix ALL: alle Objekte auf die User zugreifen darf
 - Präfix DBA: alle Objekte

View	Purpose
<i>*_tables</i>	Shows all relational tables
<i>*_tab_columns</i>	Shows all table and view columns
<i>*_sequences</i>	Lists all sequences in the database
<i>*_indexes</i>	Lists all indexes
<i>*_ind_columns</i>	Lists all indexed columns
<i>*_users</i>	Lists all users
<i>*_role_privs</i>	Lists all roles granted to users and other roles


Redundanzvermeidung

- Abhängigkeit von Attributen

- Manchmal ist der Wert eines Attributes durch die Werte anderer Attribute festgelegt

Produkt

<u>pnr</u>	name	netto	mwst	brutto
P001	Buch A	49.35	7.0	52,80
P002	Buch B	116.94	7.0	152.13
P003	Software	38.90	16.0	43.40



A red line connects the 'netto' and 'mwst' columns to the 'brutto' column, with a downward arrow pointing to 'brutto', indicating that 'brutto' is functionally determined by 'netto' and 'mwst'.

- Beispiel:

- Formal beschrieben durch funktionale Abhängigkeit

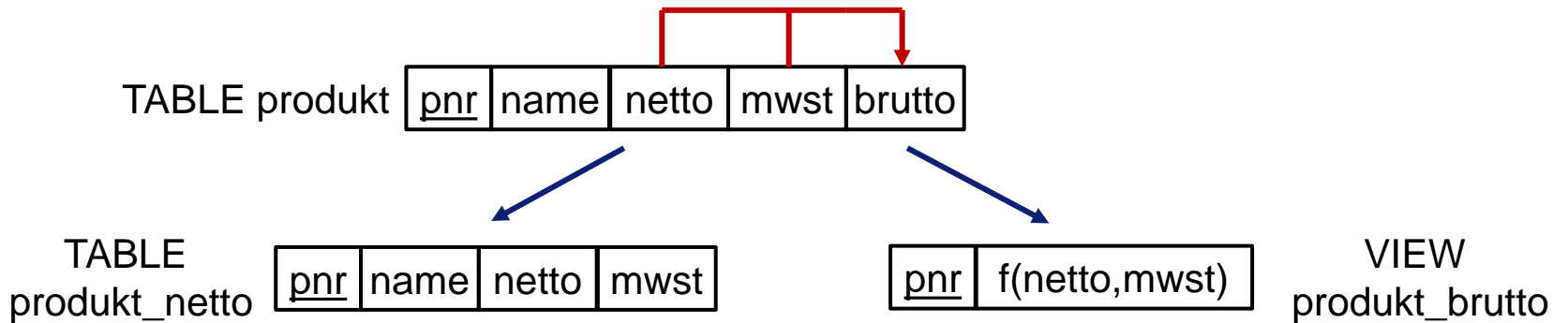
- $brutto = f(netto, mwst) := netto * \frac{100 + mwst}{100}$

- Redundanz kann zu Konsistenzproblemen führen

- Updateanomalie: Änderung von netto- und/oder mwst-Wert zieht Änderung von brutto nach sich
- Zwei Tupel mit gleichem netto/mwst-Wert müssen auch identischen brutto-Wert haben

Redundanzvermeidung durch Views

- Bei *berechenbarer* Abhängigkeit:
 - Spalte das berechenbare Attribut in einem View ab, der den Primärschlüssel und Berechnungsregel von f enthält



```
CREATE VIEW produkt_brutto AS  
  SELECT pnr, netto*(100+mwst)/100 AS brutto  
  FROM produkt_netto
```

- **Bemerkungen**
 - Rekonstruktion der ursprünglichen Tabelle über JOIN
 - Funktioniert nur bei berechenbarer Abhängigkeit
 - Ansonsten Normalisierung mittels projektiver Zerlegung (siehe Kap 7)

Änderungen auf Views

- Anforderungen:
 - Korrektheit
 - Änderung in Basisrelation(en) wirkt sich so aus, als ob der View direkt geändert würde
 - Eindeutigkeit und Minimalität
 - Welche Sätze zu ändern sind, darf nicht mehrdeutig sein
 - Diese Sätze werden minimal geändert für gewünschten Effekt
 - Integritätserhaltung
 - Änderung darf zu keinen Integritätsverletzungen führen
 - Keine Auswirkung auf "unsichtbare" Tupel der Basisrelationen
- Anforderungen im allgemeinen nicht alle erfüllbar
- Untersuche Bedingungen für Erfüllbarkeit

Projektionsviews (keine WHERE-Bedingung)

pnr#	name	preis	einfuehrung	auslauf	<i>produkt</i>
------	------	-------	-------------	---------	----------------

select ... from produkt;

- Probleme

- Bei insert wird für ausgeblendete Attribute NULL oder der bei Tabellenanlage angegebene DEFAULT eingesetzt
→ ggf. Integritätsverletzung (NOT NULL-Constraint)
- Bei Ausblendung Primary Key kein INSERT möglich
- Weitere Effekte bei Ausblendung Primary Key
 - verschiedene Tupel können als Doubletten im View auftreten
→ keine gezielte Änderung möglich
 - Bei SELECT DISTINCT entsprechen einem View-Tupel im allg. mehrere Basistupel

Selektionsviews (mit WHERE-Bedingung)

pnr#	name	preis	einfuehrung	auslauf	<i>produkt</i>
------	------	-------	-------------	---------	----------------

*billigprodukt := select * from produkt where preis < 5.0;*

- Probleme

- Änderung kann ausgeblendeten Teil betreffen

- DELETE FROM billigprodukt
WHERE **preis** > '2.0';

- Minimalitätsprinzip: keine Auswirkung auf unsichtbare Tupel

- Verschieben von sichtbar zu unsichtbar

- UPDATE billigprodukt SET **preis** = '8.5' ...

- Kann in SQL2 mit WITH CHECK OPTION unterdrückt werden

Selektionsviews mit Selbstbezug in Subquery

- Betrachte Viewdefinition über Subquery:

```
CREATE VIEW teuerstes_produkt AS  
  SELECT * FROM produkt WHERE preis = (  
    SELECT max(preis) FROM produkt  
  );
```
- Anforderung der Korrektheit für Änderungen nicht erfüllbar
 - Wie wäre nämlich z.B.

```
DELETE FROM teuerstes_produkt;
```


umzusetzen? Was ist mit updates und inserts?
 - Problem: where-Klausel wird durch Änderung mitverändert
- Views, die Subqueries mit Selbstbezug enthalten, sind daher in SQL2 nicht änderbar

Verbundviews (Joins)

hersteller

hnr#	hersteller	stadt
------	------------	-------

produkt

pnr#	produkt	preis	hnr
------	---------	-------	-----

*hersteller_produkt := select * from hersteller join produkt on ...;*

- Probleme
 - Änderungen nicht eindeutig einem Basistupel zugeordnet, z.B. Löschung eines View-Tupels auf drei Arten möglich:
 - Löschung des Produkts aus produkt
 - Löschung des Herstellers aus hersteller
 - Löschung Produkt und Hersteller
 - In letzten zwei Fällen ist Ergebnis nicht korrekt, da immer weitere Tupel aus hersteller_produkt mitgelöscht werden
- In SQL2 Änderungen auf Verbundsichten verboten

Ansätze für änderbare Views

- Automatisch änderbare Views
 - Definiere (hinreichende) Bedingungen, wann View änderbar ist
 - Solche Views sind änderbar gemäß festdefinierten Regeln
 - Bei allen anderen Views sind keine Änderungen zulässig
 - Diese Lösung wird von SQL2 gewählt
 - Bedingungen sind aber sehr restriktiv → geringer Nutzen
- Selbstdefinierbare Regeln für Änderungen
 - Ermögliche Definition von Regeln (*Rules*), was bei INSERT, UPDATE, DELETE gemacht werden soll
 - Nur Views mit solchen Rules sind änderbar
 - Diese Lösung wird von PostgreSQL gewählt
 - Flexibel, aber kein Automatismus für triviale Fälle

Änderbare Views in SQL2

- SQL2 unterscheidet nicht zwischen insert, update und delete, sondern spricht allgemein von *Updatable Views*
- Ein *Updatable View* ist ein SELECT [ALL] (kein SELECT DISTINCT) auf genau eine Basistabelle, mit folgenden Zusatzbedingungen:
 - Der View enthält keine berechneten Attribute
 - Gruppierung und Aggregation ist unzulässig
 - Subselect auf dieselbe Basistabelle ist unzulässig
 - Alle nicht im View enthaltenen Attribute dürfen in der Basistabelle NULL sein oder haben einen Default-Wert definiert (M.a.W. ein insert schlägt nicht fehl)
- CREATE VIEW bietet Parameter WITH CHECK OPTION, mit dem eine "Tupelmigration" in unsichtbaren Bereich der Basistabelle verhindert werden kann

Allgemeine Lösung mit Rules

- Nachteile SQL2 Lösung:
 - Bedingungen für Eindeutigkeit decken nur triviale Fälle ab
 - Mehrdeutige Fälle können prinzipiell nicht erfasst werden durch "automatische" Umsetzung Statements auf Basistabellen
- Allgemeinere Lösung mit Rules:
 - Rule redefiniert, was im Falle eines INSERT, UPDATE, DELETE gemacht werden soll
 - Nicht nur auf Views beschränkt, auch auf Tabellen anwendbar
 - Verwandt mit dem Trigger
 - Kein Bestandteil eines SQL-Standards, sondern PostgreSQL-spezifische Erweiterung

- Syntax:

```
CREATE [ OR REPLACE ] RULE name AS ON event
  TO table_name [ WHERE condition ]
  DO [ ALSO | INSTEAD ] {
    NOTHING | command | ( command ; command ... )
  }
```

- Weitergehende Literatur

- PostgreSQL 13 Documentation: CREATE Rule,
<https://www.postgresql.org/docs/13/sql-createrule.html>
- Stonebraker: The integration of rule systems and database systems. IEEE Transactions on Knowledge and Data Engineering 4, pp. 415-423, 1992