Grundlagen

Aufbau: Applikation -> DBMS(Datenbank-Management-System) -> Datenbanken

Persistenz: nicht-flüchtig gespeichert nach Transaktionsende

Semantische Integrität: korrekt aus Fachsicht

Konsistenz: Widerspruchsfrei, durch Integritätsregeln

Operationale Integrität: konsistenz / Integrität während Systembetrieb erhalten

Redundanz: Mehrfaches vorkommen von Daten => Normalisierung

Isolation: Keine Beieinflussung von Nutzern untereinander

ACID: Atomicity, Consistency,Isolation,Durability; Gegenpart bei Verteilten: BASE

Datenmodell: definiert Operatoren und Objekte (relational, netzwerk,xml,...)

Schema:

Logisches Schema: Struktur der Daten

Physisches Schema: Indizes etc,

Entity Integrity: eindeutigkeit der PKs

Referentielle Integrität: Datensätze erst löschen, wenn nicht mehr referenziert

Domänen Integrität: Werte liegen in Definierten Wertebereichen (falls CHECK) und sind atomar

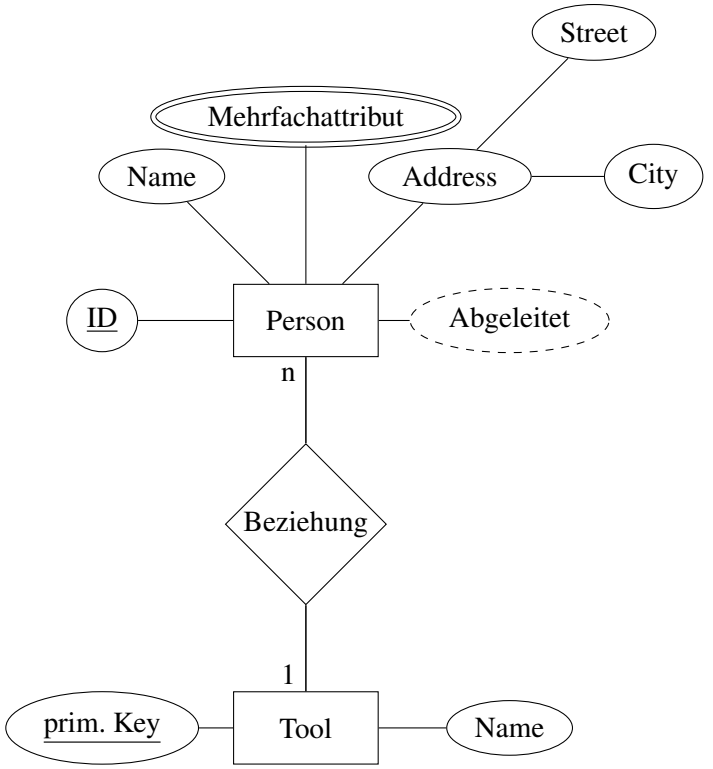
Surrogat-Schlüssel: künstlicher Primärschlüssel (id)

CAP-Theorem: Konsistenz, Verfügbarkeit, Toleranz gegen Netzausfall: nur 2 von 3 möglich

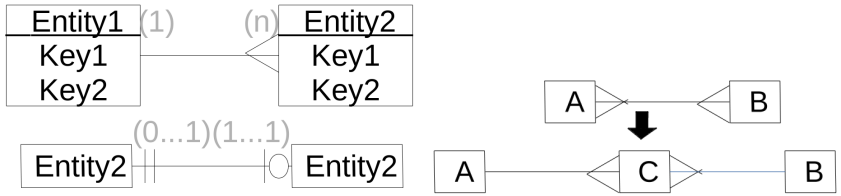
Modellierung

3Schichten-Modell: Externe Ebene(Views) - Logische Ebene(Schema) - Interne Ebene (Indizes)
=> Physische Datenunabhänigkeit: Anwendung von Änderungen auf Interner Ebene unberührt
=> Logische Datenunabhängigkeit: äuf Logische Ebene "kaum"berührt
Kardinalitäten: 1:1, 1:n, n:m, + optionalität
Identifizierende Beziehungen: schwache Entität setzt andere Entität voraus (z.B. Bestellpositionen Bestellung)

Chen-Notation



Krähenfuß



Umsetzung von Vererbung:

- Single Table (eine Tabelle, die Kind-Attribute sind ggf. NULL):
  Vorteile: Redundanzfrei, keine JOINs, Auch Elemente speicherbar, die nur zur Eltern-Klasse gehören;
  Nachteile: Viel NULL, Große Tabelle, nicht über mehrere Ebenen
- Table-per-Class: jede Klasse eigene Tabelle
  Vorteile: Redundanzfrei, Auch Elemente speicherbar, die nur zur Eltern-Klasse gehören, Einfacher Zugriff auf typunabhängige Attribute der Oberklasse, keine NULL-Werte
  Nachteile: JOINs nötig => schlechtere Performance, Referentielle Integrität muss geprüft werden
- Table-per-Concrete-Class: nur Tabellen für Klassen mit Instanzen
  Vorteile: Typen leicht unterscheidbar, Objektinformation in einer Tabelle
  Nachteile: ggf. Redundante Informationen, Anfragen über alle Objeke durch UNION

Transaktionen

SQL: BEGIN, COMMIT, ROLLBACK;
Transaktionsabbrüche durch integritätsverletzungen, Konsistenzbedinungen, Speicher voll, Verbindungsabbruch, Systemausfall
Nebenläufigkeit:

1. Lost-Update: Überschriebene Änderungen
2. Dirty-Read: Lesen eines nicht commiteten Wertes
3. Unrepeatable Read: durch commit anderer Transaktion liefert die selbe anfrage nicht das gleiche Ergebnis
4. Phantom Read: Einfügen von Datensätzen durch andere Transaktion

Serialisierbar: Gleiches Ergebnis wie bei hintereinanderausführung der Transaktion $\Leftrightarrow$ kein Zyklus im Abhängigkeitsgraph z.B. $R_1(a), W_2(a), R_3(a), W_1(a)$

Sperren: Shared-Locks (s) beim Lesen, Exklusiv-Locks (X) beim Schreiben
Zwei-Phasen-Protokoll: Sperren werden erst am Transaktionsende (vor commit) wieder freigegeben

T1	T2
Slock(a), read(a);	
	Slock(a), read(a);
Wait(Xlock(a))	
	Wait(Xlock(a))
	DEADLOCK => Rollback;
write(a)	
unlock(a);	
commit;	

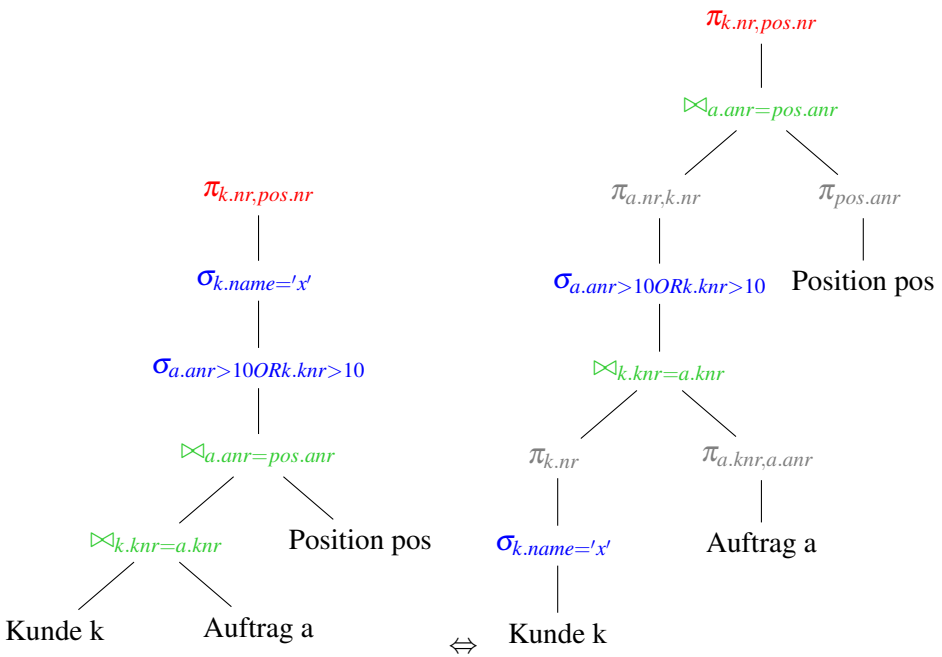
SQL-Isolationssteuerung: SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }
Multi-Version-Concurrency-Control: Snapshots: keine Lesesperren, änderungen erzeugen kopie

Fehlerbehandlung

Lokaler Fehler in Transaktion: Rollback der Transaktion
Verlust des Hauptspeichers:Durch Logging der Aktionen, nach Crash Undo nicht abgeschlossener Aktionen, Redo abgeschlossener.
Verlust des externen Speichers: Backup einspielen, inkrementell log-einspielen
Write-Ahead-Logging: festhalten aller änderungen vor commit, bei rollback wiederherstellung aus transaktionslog;
Log beinhaltet Undo- und Redo-Informationen; (vor und nach zustand)
logisches Logging: protokollierung der Befehle
physisches Logging: Zustandkopien
Steal-noforce: Steal: gepufferte seiten können von anderer Transaktion eingelagert werden, zusammen mit zugehörigem Undo-logeintrag NoForce: committete Seiten müssen nicht sofort auf platte geschrieben, aber im Redo-log vermerkt werden.
Ablauf: Redo-lauf (durchführen der geloggten änderungen); Undo-lauf: rollback nicht committeter änderungen
Recovery time objective: maximale ausfallzeit; recovery point objective: maximaler Datenverlust

Anfrageverarbeitung

Heap-Dateiorganisation: Speichern von Datensätzen in Einfügereihenfolge => unsortiert Einfügen am ende, löschen suchen und setzen eines Löschbits, suchen: sequenziell oder index;
sequentielle-Dateiorganisation: sortierte speicherung; einfügen: suchen, anhängen und dann sortieren der Seite; löschen: suchen und löschbit setzen; suchen über index
Index: B-Baum-Struktur; clustered Index: Segmente selbst sind sortiert; CREATE INDEX <name> ON <tabelle>(<spalte(n)>); DROP INDEX <name>
SQL -> Query Execution Plan: Parsen der Anfrage -> Operatorengraph; // hierbei Standardisieren und vereinfachen (KNF = (a and b) or (c and d); deMorgan: $\overline{aANDb} = \bar{a}OR\bar{b}$
SELECT k.name, pos.nr // $\pi_{k.nr,pos.nr}$
FROM Kunde k JOIN Auftrag a ON k.knr = a.knr // $\bowtie_{k.knr=a.knr}$
JOIN Position pos ON a.anr =pos.anr // $\bowtie_{a.anr=pos.anr}$
WHERE k.name = 'x' AND // $\sigma_{k.name='x'}$
(a.anr > 10 OR k.knr >10) // $\sigma_{a.anr>10ORk.knr>10}$
⇔



Kostenschätzung

Anhand von Statistiken, Histogrammen, etc.; evtl. Hints für Optimizer
Kardinalität: |a|: Anzahl der gelieferten Datensätze
Selektivität: Sel(a): % der Datensätze im vergleich zu gesamtzahl;
NumBlocks: $\frac{|R| \cdot Laenge_{Datensatz}}{Blocksize}$ Levels(I(R,A)): Höhe des Index auf A
Sel(P) = $\frac{|\sigma_P(R)|}{|R|}$
Attribut = 'sth' => Sel(A) = 1/|A| Attribut IN $\{c_1, c_2, \ldots, c_n\}$ => Sel(A) = n / |A| A >c => Sel(A) = $\frac{A_{max} - c}{A_{max} - A_{min}}$ P_1ANDP_2 => $Sel(P_1) * Sel(P_2)$ P_1ORP_2 => $Sel(P_1) + Sel(P_2) + Sel(P_1ANDP_2$

Plan-Operatoren

- Full-Table-Scan: durchsuche gesamte Tabelle: Cost = NumBlocks(R)
- Index-Scan: Suche anhand index: Cost = Levels(Index) + Sel(P) *|R|
- Nested-Loop-Join: für jeden Block in einer Tabelle: durchlaufe die Gesamte andere Tabelle Ohne Index: Cost= NumBlocks(R) * NumBlocks(S) ; mit Index Cost= NumBlocks(R) * Cost(IndexScan)
- Merge-Join: Sortiere die Relationen nach Join-Attribut; wechselndes Ablaufen der sortierten Relationen; Cost: Cost(Sort(R)) +Cost(Sort(S)) + NumBlocks(S) + NumBlocks(R)
- Hash-Join: Teile kleinere Relation R in x Abschnitte, die im RAM gehalten werden können; für jeden Abschnitt: Hashen der Datensätze in R; prüfe für jeden Datensatz der größeren Relation die Join Bedinung beim entsprechenden Hasheintrag; Cost: NumBlocks(R) + x * NumBlocks(S)

BigData: 3+1Vs :

- Volume: große Datenmengen
- Velocity: hohe Erzeugungsgeschwindigkeit
- Variety: strukturierte, semistrukturierte, unstrukturierte Daten
- Veracity: geringe Qualität/Glaubwürdigkeit

SQL - Beispiele

Studenten

MatrikelNr	Name	Vorname	Vorname2	Geburt	Ort	SgNr	Bafoeg
1001	Schmidt	Hans	Peter	24.2.1990	Nürnberg	2	200
1002	Meisel	Dirk	Helmut	17.8.1989	Fürth	3	500
1003	Schmidt	Amelie		19.9.1992	Wendelstein	1	0
1004	Krause	Christian	Johannes	3.5.1990	Nürnberg	1	100
1005	Schäfer	Julia		30.3.1993	Erlangen	5	0
1006	Rasch	Lara		30.3.1992	Nürnberg	3	0
1007	Bakowski	Juri		15.7.1988	Fürth	4	400

Studiengaenge

SgNr	Kuerzel	Name	Fak
1	IN	Informatik	IN
2	WIN	Wirtschaftsinformatik	IN
3	MIN	Medieninformatik	IN
4	BW	Betriebswirtschaftslehre	BW
5	ET	Elektrotechnik	EFI

Projektion (Spaltenauswahl) und Selektion

$\pi_{Name,Vorname,Ort}(\sigma_{Name='Schmidt'}(Studenten))$

```
SELECT Name, Vorname, Ort
FROM Studenten
WHERE Name='Schmidt'
ORDER BY Ort, Name, Vorname
```

Sortierung (ORDER BY) gibt es in der relationalen Algebra nicht.
Schlüsselwort DISTINCT nach SELECT eliminiert Duplikate.

```
SELECT DISTINCT Name
FROM Studenten
```

Projektion mit Funktionen

Für einen Übersicht der Funtionen in MySQL siehe:
http://dev.mysql.com/doc/refman/5.7/en/func-op-summary-ref.html

```
SELECT
  COALESCE(Vorname2, 'kein 2. Vorname'), -- Nullwert-Behandlung
  CONCAT(Vorname, ' ', Name) AS Name, -- SQL Standard
  Vorname + ' ' + Name AS Name2, -- nur MS SQL Server
  YEAR(GebDat) AS GebJahr,
  TIMESTAMPDIFF(YEAR, GebDat, CURRENT_DATE) AS AlterJahre,
  UPPER(Name) AS NameGross,
  CASE WHEN LENGTH(Vorname) > 5 THEN 'lang' ELSE 'kurz' END AS Länge
FROM Studenten
```

Selektionen mit komplexen Bedingungen

```
WHERE Bafoeg + 100 < 1000 AND Bafoeg != 0

WHERE Bafoeg BETWEEN 100 AND 500

WHERE SgNr IN (1, 2, 3)

WHERE Name LIKE 'M%' OR Name NOT LIKE '%M'

WHERE GebDat > '1990-01-01'

WHERE Vorname2 IS NULL;
```

Joins

Inner Join

$\pi_{s.Name,\ sg.Fak}$ (Studenten $\bowtie_{Studenten.SgNr=Studiengaenge.SgNr}$ Studiengaenge)

```
SELECT s.Name, sg.Fak
FROM Studenten s JOIN Studiengang sg ON s.SgNr = sg.SgNr
```

Äquivalente Version über kartesisches Produkt:

$\pi_{s.Name,\ sg.Fak}$ ($\sigma_{Studenten.SgNr=Studiengaenge.SgNr}$(Studenten x Studiengaenge))

```
SELECT s.Name, sg.Fak
FROM Studenten s, Studiengaenge sg
WHERE s.SgNr = sg.SgNr
```

Outer Join

```
SELECT s.Name, sg.Fak
FROM Studenten s RIGHT JOIN Studiengang sg ON s.SgNr = sg.SgNr
```

Right Join gibt alle Datensätze der rechten Tabelle aus, auch wenn kein passender Datensatz in linker Tabelle vorhanden. Bei Left Join ist es umgekehrt.

Tips zu Joins: Inner Joins sind performanter zu berechnen und meist ausreichend. Ein Outer Join wird nur benötigt, wenn explizit auch z.B. Studiengänge ohne Studenten ausgegeben werden sollen.

Aggregation

Anzahl der Studenten pro Ort:

```
SELECT Ort, COUNT(*)
FROM Studenten
GROUP BY Ort
```

Gesamtsumme des Bafög und Anzahl der verschiedenen Wohnorte pro Studiengang für alle Studiengänge mit mindestens 2 Studenten:

```
SELECT
  sg.Name, sg.Fak, SUM(s.Bafoeg), COUNT(DISTINCT s.Ort)
FROM Studenten s JOIN Studiengang sg ON s.SgNr = sg.SgNr
GROUP BY sg.Name, sg.Fak
HAVING COUNT(*) >= 2
```

Tips zu Aggregationsanfragen:

- Alle genannten Spalten in der SELECT-Klausel müssen entweder in einer Aggregatfunktion verwendet oder in die GROUP-BY-Klausel übernommen werden.
- In der GROUP-BY-Klausel tauchen meistens Attribute wie IDs, Namen, Bezeichnungen etc. auf. Numerische Attribute wie Mengen, Preise, Anzahlen usw. werden üblicherweise mit einer Aggregatfunktion wie SUM verrechnet. D.h. falls Sie in einer Anfrage "GROUP BY Menge" o.ä. stehen haben, ist wahrscheinlich etwas falsch.

Handout Datenbanken

Insert, Update, Delete

```
INSERT INTO Studiengaenge(SgNr, Kuerzel, Name, Fak)
VALUES (1, 'IN','Informatik','Inf')

UPDATE Studenten
SET Ort = 'Berlin', Bafoeg = 0
WHERE Name = 'Meier' AND Vorname = 'Hans'

DELETE
FROM Studenten
WHERE MatrikelNr = 1235
```

Unterabfragen

Unterabfragen in der WHERE-Klausel

Alle Studiengänge ohne Studenten:

```
SELECT *
FROM Studiengaenge
WHERE SgNr NOT IN (SELECT SgNr FROM Studenten)
```

Alle Studiengänge mit Studenten

```
SELECT *
FROM Studiengaenge sg
WHERE EXISTS (SELECT * FROM Studenten s WHERE s.SgNr = sg.SgNr)
```

Unterabfragen in der FROM-Klausel

Können wie eigene Tabellen verwendet werden:

```
SELECT MAX(Gesamtbafoeg_pro_SG)
FROM (
  SELECT SgNr, SUM(Baefoeg) AS Gesamtbafoeg_pro_SG)
  FROM Studenten
  GROUP BY SgNr
) AS tmp
```

Mengenoperationen
Für Mengenoperationen müssen die beiden eingehenden Tabellen (gebildet durch eingeschränkte Abfragemöglichkeiten) die gleiche Anzahl von Spalten mit kompatiblen Datentypen haben.

Vereinigung
Alle Sätze aus beiden Tabellen.

```
SELECT Name, Vorname, Email
FROM Leser
UNION
SELECT Name, Vorname, Email
FROM Autoren
```

UNION eliminiert automatisch Duplikate. UNION ALL behält Duplikate und ist daher aus Performance-Aspekten vorzuziehen, solange eine Duplikat-Elminierung nicht erforderlich ist.

Durchschnitt
Alle Sätze, die sowohl in der ersten als auch in der zweiten Tabelle vorkommen, Duplikate werden eliminiert.

```
SELECT Name, Vorname, Email
FROM Leser
INTERSECT
SELECT Name, Vorname, Email
FROM Autoren
```

Differenz
Nur die Sätze aus der ersten Tabelle, die nicht in der zweiten enthalten sind, Duplikate werden eliminiert.

```
SELECT Name, Vorname, Email
FROM Leser
EXCEPT     -- bei den meisten DBS MINUS
SELECT Name, Vorname, Email
FROM Autoren
```

NoSQL

Vorteile: kompakt, änderungen betreffen nur eine Tabelle Nachteile: Konsistenzsicherung, eingeschränkte Abfragemöglichkeiten

key-value stores

reine zuordnung von key-value-paaren => einfaches Datenmodell, keine Integritätsbedingungen/Joins, schnell, gut aufteilbar, Abfragen ohne Key schwierig

Document stores

abspeichern von JSON/XML zu keys; => flexibles Datenmodell, ähnlich Key-Value

Column Family

Feste Column Familys (z.B. meta, posts,…) Row Key -> Column Familys -> liste von Key-Values

gute Kompresson, flexibles Schema, schnelles Schreiben

Graph DB

Knoten und Kanten, beide mit Eingenschaften

Graphalgorithmen (Tiefen/Weitensuche) anwendbar

Verteilte Architekturen

- shared Memory: Multicore-System
- shared Disk: mehrere Server nutzen zentrale Platten => Synchronisation aufwendig
- shared nothing: kein gemeinsamer speicher

Sharding: aufteilung der Datensätze; Typisch: Kombination aus Replikation/Sharding z.B. Tabelle(A,B,C,D) -> Server1(A,B), Server2(B,C), Server3(C,D), Server4(D,A)
Replikation: synchron vs asynchron.

Normalisierung: Vorgehensweise

Ausgangsrelation

Verkäufe(Datum, KundenID, Name, Vorname, Wohnort, ProdID, Produkt, Marke, PgID, Produktgruppe)

Datum	KundenID	Name	Vorname	Wohnort	ProdID	Produkt	Marke	Menge	PgID	Produktgruppe
17.06.12	K1	Nuhr	Dieter	Düsseldorf	100	S 4	Samsung	2	1	Smartphone
17.06.12	K2	Pelzig	Erwin	Würzburg	101	iPhone 5	Apple	1	1	Smartphone
31.08.12	K3	Gruber	Monika	Erding	702	iPad	Apple	2	2	Tablet
12.10.12	K2	Pelzig	Erwin	Würzburg	702	iPad	Apple	3	2	Tablet
18.12.12	K3	Gruber	Monika	Erding	115	Galaxy 10	Samsung	2	2	Tablet
23.12.12	K3	Gruber	Monika	Erding	366	MacBook	Apple	1	3	Notebook
23.12.12	K2	Pelzig	Erwin	Würzburg	587	Vaio	Sony	1	3	Notebook
27.12.12	K1	Nuhr	Dieter	Düsseldorf	100	S 4	Samsung	1	1	Smartphone

Überführung in 1 NF

Sofern notwendig: Auflösung mehrwertiger Attribute in Extra-Relationen

Überführung in 2NF: Eliminiere partielle Abhängigkeiten von Primärschlüssel-Teilen

Schritt 1: Volle funktionale Abhängigkeiten identifizieren (sofern nicht vorgegeben)
Voraussetzung: Identifikation des Primärschlüssels der Relation (hier: KundenID, ProdID, Datum).
Gibt es Attribute, die von unterschiedlichen Teilen des Primärschlüssels abhängen?

- KundenID, ProdID, Datum → Menge
- KundenID → Name, Vorname, Wohnort
- ProdID → Produkt, Marke, PgID, Produktgruppe

Schritt 2: Zerlegung

- Verkäufe_Neu(Datum, KundenID, ProdID, Menge)
- Kunden(KundenID, Name, Vorname, Wohnort)
- Produkte(ProdID, Produkt, Marke, PgID, Produktgruppe)

Schritt 3: Integritätsbedingungen angeben

- Verkäufe.KundenID referenziert Kunde.KundenID
- Verkäufe.ProdID referenziert Produkte.ProdID

Überführung in 3NF: Eliminierte funktionale Abhängigkeiten zwischen Nicht-Schlüssel-Attr.

Schritt 1: Transitive funktionale Abhängigkeiten identifizieren

- ProdID → PgID und PgID → Produktgruppe
- Betrifft nur Relation Produkte

Schritt 2: Zerlegung von Produkte

- Produkte_Neu(ProdID, Produkt, Marke, PgID)
- Produktgruppen(PgID, Produktgruppe)

Schritt 3: Integritätsbedingungen angeben

- Produkte_Neu.PgID referenziert Produktgruppen(PgID)