3.3 Geschachtelte Anfragen

- **Kompositionsfähigkeit von SQL**
  - werterzeugende Unteranfrage (engl. **scalar subquery**)
    - In SQL kann überall dort, wo ein Wert verlangt wird, eine temporäre Relation mit einem Attribut und einer Spalte verwendet werden.
  - tupelerzeugende Unteranfrage (engl.: **row subquery**)
    - In SQL kann überall dort, wo ein Tupel stehen darf, eine Unteranfrage stehen, die genau ein Tupel produziert.
  - relationenerzeugende Unteranfrage (engl.: **table subquery**)
    - In SQL kann überall dort, wo bisher eine Relation steht, eine temporäre Relation in Form einer SQL-Anfrage verwendet werden.

3.3.1 Werterzeugende Unteranfragen

- Diese Unteranfragen produzieren eine Relation mit genau einem Tupel und einem Attribut.
  - Beispiel für eine Anfrage, die genau einen Wert liefert.

```
select avg(note)
from pmzuteilung;
```

- Diese Anfrage kann man als Unteranfrage in einem beliebigen Ausdruck statt eines Werts verwenden.
  - Beispiel:

```
select pnr
from pmzuteilung
where note > (select avg(note)
              from pmzuteilung)
```

! Unteranfrage muss in einem Klammerpaar stehen. !

Ausdrücke in SQL

- **Werterzeugende Unteranfragen werden in SQL in einem Ausdruck verwendet.**
  - Ein Ausdruck hat einen Typ.
- **Übersicht zu Ausdrücken in SQL**
  - Arithmetische Ausdrücke
  - Ausdrücke mit Zeichenketten
    - Hier gibt es die aus anderen Programmiersprachen bekannten Operationen.
  - Ausdrücke für spezielle Datentypen
    - Datum & Zeit
  - Boolesche Ausdrücke
    - Beachte dabei die dreiwertige Logik.
    - Wir werden später noch mehr über die Möglichkeiten von Unteranfragen in solchen Ausdrücken erfahren.

Beispiel

- **Anfrage**
  - Liefere alle Maschinen, die Angestellter mit pnr 67 besser bedient als der Durchschnitt.

```
select mnr
from PMZuteilung
where pnr = 67 and note < (select avg(note)
                           from PMZuteilung);
```

  - Diese Anfrage ist syntaktisch korrekt, da der Datentyp von dem Attribut Note zu dem Typ der Unteranfrage passt.

Unteranfragen in select-Klausel

- **In der select-Klausel können werterzeugende Unteranfragen verwendet werden.**

- **Ein einfacher Spezialfall sind Anfragen von dem Typ.**

```
select (select count(*) from PMZuteilung);
```

  - Solche Anfragen sind jedoch nicht von Bedeutung.

Korrelierte Unteranfrage

- **Sinnvoll sind solche Unteranfragen in der select-Klausel erst, wenn die Unteranfrage von der äußeren Anfrage abhängig ist.**

  ! Wichtig !

  - Man spricht dann von **korrelierten Unteranfragen**!
  - Beispiel:

```
select B.pnr,
       (select count(*) from Personal A
        where A.abt = B.abt and Lohn > B.Lohn)
from  Personal B;
```

    - Diese Anfrage liefert für jede Person B mit Personalnummer B.pnr genau einen Wert!
    - Dabei werden oft Tupelvariablen verwendet und die Auswertung läuft dann wie beim Tupelkalkül.

Group-by Ersatz

- **Mit solchen Unteranfragen kann jede group-by-Klausel nachgebaut werden.**
  - Historische Anmerkung:
    Diese Unterfragen sind erst seit SQL2003 im Standard.
- **Tatsächlich ist dies ausdrucksstärker als group-by**
  - Es sind mehrere Gruppierungen in einer Anfrage möglich.

```
select B.pnr,
       (select count(*) from Personal A
        where A.pnr = B.pnr and Lohn < 50000),
       (select count(*) from Personal A
        where A.pnr = B.pnr and Lohn > 60000)
from  Personal B;
```

- **Wenn das Gruppierungsattribut nicht eindeutig ist, sollte man select distinct verwenden.**

3.3.2 Relationenerzeugende Unteranfragen

- **Statt einer persistenten Relation kann eine Unteranfrage als temporäre Relation benutzt werden.**
  - Dabei wird die Unteranfrage geklammert.
- **Solche Anfragen können in der From- und Where-Klausel auftreten.**
  - In der From-Klausel muss in PostgreSQL für eine Unteranfrage stets eine **Tupelvariable** definiert werden.
- **Beispiel**

```
select v.mnr
from (select mnr, note from PMZuteilung where pnr < 100) v
where v.note < 3;
```

Unteranfrage

Tupelvariable

Tupelvariablen

- **Tupelvariablen müssen bei Unteranfragen deklariert werden.**
  - Deklaration in der from-Klausel
  - Mit dieser Variable kann der Bezug zu den Variablen der temporären Relation hergestellt werden.

Having-Ersatz mit Unteranfragen

- **Mit Unteranfragen ist es möglich die having-Klausel nachzubauen.**
  - Pro Gruppe müssen dann ggf. weitere Aggregate für den Ausdruck in der having-Klausel produziert werden.
  - Diese Aggregate können dann durch eine Projektion wieder entfernt werden.

- **Beispiel**
  - Wie gut können Maschinen im Durchschnitt bedient werden? Liefere nur die Maschinen, die von mindestens vier Angestellten bedient werden können!

    → siehe Vorlesung

Unteranfragen in der Where-Klausel

- **Relationenerzeugende Unteranfragen mit exists**
  - In der where-Klausel sind Unteranfragen erlaubt, die einen Booleschen Wert zurückliefern. Hierzu wird das Schlüsselwort exists genutzt.
    - exists <Subquery>
      - true ⟷ Unteranfrage ist nicht leer.
      - Ansonsten wird der Wert false geliefert.

- **Bedeutungen von exists**
  - exists steht für den Existenzquantor in SQL
  - Man kann exists auch als Aggregatoperation ansehen, die zu einer Menge die Werte true oder false liefert.

Korrelierte Unteranfragen in der where-Klausel

- **Unteranfragen in der where-Klausel treten sehr oft in korrelierter Weise auf.**
  - Welche Abteilungen haben Angestellten, welche die Maschine 84 bedienen können?

Äußere Anfrage

Innere Anfrage

```
select distinct abt
from Personal P
where exists  (select pnr
               from PMZuteilung
               where P.pnr = pnr and mnr = 84)
```

Korrelation

Das ist doch wie im Tupelkalkül!

Sichtbarkeit von Tupelvariablen in korrelierten Unteranfragen

- **Eine korrelierte Unteranfrage ist abhängig von einer Tupelvariable der äußeren Anfrage.**
  - Ähnlich dem Konzept bei inneren und äußeren Klassen in Java stellt sich die Frage, welche Variablen in einer Unteranfrage gültig sind.

- **Sichtbarkeit einer Tupelvariable**
  - Eine Tupelvariable ist zunächst in allen zugehörigen Unteranfragen gültig.
  - Wird die Tupelvariable erneut deklariert, ist die äußere Deklaration nicht mehr sichtbar.
  - Sucht man die Deklaration einer Tupelvariable geht man von innen nach außen bis zur ersten Deklaration.

Auswertung von korrelierten Unteranfragen

- **Eine korrelierte Unteranfrage ist abhängig von einer Tupelvariable der äußeren Anfrage.**

- **Die Auswertung entspricht dem im Tupelkalkül**
  - Für jeden möglichen Wert der Tupelvariable, muss die korrelierte Unteranfrage einmal ausgeführt werden.

  - Diese Auswertung ist i. A. **sehr teuer** und es stellt sich deshalb die Frage, ob man diese Anfragen noch anders ohne eine korrelierte Unteranfrage berechnen kann.
    - Wenn es gelingt, spricht man vom **Dekorrelieren der Unteranfrage**.

Unkorrelierte Unteranfragen

- **Wenn eine Unteranfrage unabhängig von der äußeren Anfrage ist, spricht man von einer unkorrelierten Unteranfrage.**
  - Im Gegensatz zu einer korrelierten Anfrage ist dann eine einmalige Auswertung der Unteranfrage möglich.
- Die Kosten bei der Auswertung solcher Unteranfragen sind i. A. niedrig.

Vermeidung korrelierter Unteranfragen

- **Häufig lassen sich korrelierte Unteranfragen in gewöhnliche Umfragen transformieren.**
  - Beispiel
    - Welche Angestellte können keine Maschine mit Note 2 und besser bedienen.

```
select *
from Personal p
where not exists  (select pnr
                   from PMZuteilung
                   where p.pnr = pnr and note < 3);
```

    - Diese Anfrage lässt sich auch über einen Left-Outer Join ausdrücken.
      - Wie?

Äquivalente Anfragen

- Gegeben
  - Prädikat p und zwei Relationen r und s.
- Dann liefern die folgenden Anfragen das gleiche Ergebnis.
  - select * from s
    where not exists (
      select * from r
      where s.K = r.K and p)

  - select s.*
    from s left outer join
        (select * from r where p) t on s.K = t.K
    where t.K is null

Mengenwertige Unteranfragen

- Durch das Schlüsselwort in kann getestet werden, ob ein Attribut einen Wert in einer Menge annimmt.
  - Beispiel

```
select pnr, PName
from Personal
where pnr in
        (select pnr from PMZuteilung)
```

Das muss zueinander passen!

- Durch Negation lässt sich auch not in testen.

Differenz mit Unteranfragen

- Differenz zwischen zwei Relationen
  - Verwendung von except
  - Unteranfrage mit not in
  - Unteranfrage mit not exists
  - …
- Beispiel:
  - Berechne alle Angestellten, die derzeit keine Maschine bedienen können.
    - select *
      from Personal
      where pnr not in (select pnr from PMZuteilung)
    - select *
      from Personal p
      where not exists (select pnr from PMZuteilung where pnr = p.pnr)

3.3.3 Allquantifizierte Anfragen

- Existenzquantor in SQL mit exists
  - Überprüfung, ob es mindestens ein Tupel in einer Unteranfrage gibt.
- Für den Benutzer sind jedoch Anfragen besonders wichtig, ob alle Tupel einer Relation (Unteranfrage) etwas erfüllen.
  - Berechne die Angestellten, die alle Maschinen bedienen können.
  - Berechne die Studenten, die alle Vorlesungen bei Prof. Taentzer gehört haben.
  - …
- Gibt es dafür einen Allquantor in SQL?
  - Nein, den gibt es in seiner allgemeinen Form nicht.
  - Nur für sehr spezielle Fälle, gibt es diesen Operator.

    → einfache allquantifizierte Anfragen

Einfache allquantifizierte Anfragen

- Teste, ob ein Attribut mit allen Elementen einer Menge in einer bestimmten Beziehung steht.
  - Verwendung des Schlüsselworts all
  - Beispiel
    - Suche für alle Maschinen die Angestellten mit den besten Noten!

```
select *
from PMZuteilung L
where note <= all (   select note
                      from PMZuteilung
                      where mnr = L.mnr)
```

Unterschied all und some

- Neben dem Schlüsselwort all gibt es noch some bzw. any.
- Dabei gilt folgenden Semantik (3-wertige Logik !!)

Semantik ALL	
• alle Elem. true:	**true**
• mind. 1 Elem. false:	**false**
• sonst:	**unknown**

Semantik SOME bzw. ANY	
• alle Elem. false:	**false**
• mind. 1 Elem. true:	**true**
• sonst:	**unknown**

- Beispiel
  - Suche für alle Maschinen die Angestellten, die besser sind als der/die mit der schlechtesten Note!

```
select *
from PMZuteilung L
where note < some(  select note
                    from PMZuteilung
                    where mnr = L.mnr)
```

Allgemeine allquantifizierte Anfragen

- **Es gibt keinen Allquantor in SQL !!**
  - Stattdessen müssen die Anfragen auf den Existenz-Operator exists abgebildet werden.
  - Für eine Boolesche Funktion f mit freier Variable x gilt:

$$\forall x: f(x) \quad \Leftrightarrow \quad \neg(\exists x: \neg f(x))$$

  - Nach diesem Lösungsmuster können nun allquantifzierte Anfragen beantwortet werden.
- **Beispiel**
  - Welche Mitarbeiter können alle Maschinen bedienen, die Mitarbeiter 114 bedienen kann?
    - Anfrage im Tupelkalkül?

Lösung mit Unteranfragen

- **Anfrage**
  - Welche Mitarbeiter können alle Maschinen bedienen, die Mitarbeiter 114 bedienen kann?

```
select distinct T.pnr
from PMZuteilung T
where not exists   (
    select U.mnr
    from PMZuteilung U
    where U.pnr = 114 and not mnr in (
                select V.mnr
                from PMZuteilung V
                where V.pnr = T.pnr))
```

Alle Maschinen, die Person 114 bedienen kann, aber nicht pnr.

Alle Maschinen, die Person pnr bedienen kann.

Allquantifizierte Anfragen und Aggregate

- **Bei einigen allquantifizierte Anfragen können Aggregate genutzt werden.**
  → Dadurch entstehen i. A. etwas einfachere SQL-Anfragen
- **Anfrage**
  - Welche Angestellte können alle Maschinen bedienen?

```
select pnr
from PMZuteilung
group by pnr
having count(*) = (select count(*) from
                        (select distinct mnr
                        from PMZuteilung) t
                  );
```

Test auf eindeutige Existenz in SQL

- Mit Unteranfragen ist es ziemlich unkomfortabel, zu überprüfen, ob genau ein Objekt mit einer bestimmten Eigenschaft existiert.

- Prädikatenlogischer Hintergrund: Der $\exists!$-Operator wird "simuliert" mittels

$$\exists!\ X:\ F \quad \Leftrightarrow \quad \exists\, x:\ F \ \wedge\ [\ \forall\, y:\ F \Rightarrow (y=x)\ ] \quad \text{bzw.}$$
$$\ldots \qquad\qquad \exists\, x:\ F \ \wedge\ [\neg\, \exists\, y:\ F \wedge \neg\, (y=x)\ ]$$

- Welche Möglichkeiten gibt es dafür noch in SQL?

- Beispiel: Welche Maschinen können von genau einem Angestellten aus der Abteilung A64 bedient werden?

Existenzbedingungen: Resümee

Warum diese ganze "Wirbelei" mit Quantoren in SQL ????

- **Nur sehr einfache Anfragen lassen sich ohne die genauere Kenntnis von Existenzbedingungen formulieren.**
- **Sowie das Wörtchen "alle" bzw. "jeder" ins Spiel kommt, wird es schwierig und Quantoren sind nahezu unvermeidbar!**
- **Auch "genau ein"-Anfragen haben es in sich und treten in der Praxis auch öfters auf.**
- **Graphische Anfrage-Editoren (à la MS Access) lassen einen bei diesen Anfragetypen "im Regen stehen": Ohne SQL-Formulierung geht es dann nicht!**

Ohne gute Grundkenntnisse der Prädikatenlogik (insbesondere der Quantoren) kann man niemals „vernünftig“ SQL anwenden !

3.4 Rekursive Anfragen

- **Motivation**
  - In unserer Datenbank können Angestellte Vorgesetzte von anderen Angestellte sein.

    ➔ Es kann eine Mitarbeiterhierarchie mit unbekannter Zahl von Hierarchieebenen geben.
  - Man spricht dann von einer Hierarchie oder einem Netzwerk, die/das es in vielen Anwendungen gibt.
    - Dateien in einem Dateisystem
    - Aufgaben (und Teilaufgaben) in einem Projekt
    - Verbindungen in sozialen Graphen
  - Anfragen in Hierarchien und Netzwerken, wie z. B.
    - Berechne für eine Person alle Vorgesetze

  benötigen die Möglichkeit Anfragen **rekursiv** zu formulieren.

Historische Anmerkungen

- Rekursion war ursprünglich nicht in SQL vorgesehen und ist auch heute nur rudimentär umgesetzt.
- Lange Zeit gab es keine Standardisierung für rekursive Anfragen.
  - In Oracle gab und gibt es ein connect-by-Befehl.
- Seit SQL:1999 gibt es die **WITH-Klausel** für rekursive Anfragen im Standard.
  - Dieser Standard ist aber nur von einigen Systemen wie z. B. SQL Server, PostgreSQL, SQLite, … implementiert.

WITH-Klausel

- **WITH unterstützt, dass *temporäre* Relationen mit einem Namen angelegt werden können.**
  - Nutzung der Relation nur in dem darauffolgenden SQL-Befehl.
    - ➔ Vereinfachung von komplexen Anfragen
- **Beispiel**
  - ```
    with MitarbeiterStat(pnr, perf) as
      ( select pnr, avg(note)
        from PMZuteilung group by pnr)
    select pnr
    from MitarbeiterStat
    where perf < (select avg(perf) from MitarbeiterStat);
    ```

Beachte: with ist nur zusammen mit einer Anfrage erlaubt.
    ```

WITH Recursive

- **Der wirkliche Nutzen von WITH wird erst in der rekursiven Variante deutlich!**
  - WITH RECURSIVE in Kombination mit UNION [ALL]
- **Rekursives Erzeugen von temporären Relation**
- **Beispiel**

```
with recursive gauss(n) as (
        values (1)
  union
        select n+1 from gauss where n < 10)

select sum(n) from gauss;
```

Erzeugung einer Relation mit einer Spalte und einem Tupel.

Selbstbezug ist erlaubt!

Auswertung rekursiver Anfragen

1. **Auswertung des nicht-rekursiven Terms**
   - Die Tupel sind Antworten der rekursiven Anfrage.
   - Abspeicherung der Tupel in einer ***temporären Arbeitsrelation***
2. **Solange die Arbeitsrelation nicht leer ist:**
   - Auswertung des rekursiven Terms für alle Tupel in der Arbeitsrelation.
     - Im Fall von union (gilt nicht bei union all) werden Duplikate mit zuvor berechneten Antworten beseitigt.
     - **Neue Ergebnisse** kommen in eine ***temporäre Relation***
   - Setze die Arbeitsrelation auf die temporäre Relation.

Darauf ist zu achten

- **Bei rekursiven Anfragen muss die Arbeitsrelation irgendwann leer sein (Fixpunktsemantik)**
  - Ansonsten bricht die Rekursion nicht ab!
  - union statt union all hilft dabei (→ aber teuer ??)

- **Noch ein Beispiel**
  - Berechne alle Angestellten mit Lohn von mindestens 60000 und deren Vorgesetzte.
    - Berechnung der Angestellten mit Lohn 60000 und höher

Typische Einschränkungen

- **The following items are not allowed in a recursive with-clause (SQL server):**
  - SELECT DISTINCT
  - GROUP BY
  - HAVING
  - Scalar aggregation
  - TOP (= Limits)
  - LEFT, RIGHT, OUTER JOIN (INNER JOIN is allowed)
  - Subqueries

3.5 Einfügen, Löschen und Ändern

- **Einfügen**
  - Einzelnes Tupel
    - insert into <Relationen-Name>
      [(<Attributname> [, <Attributname>]*)]
      values (<Konstante> [, Konstante]*)

Verwendung der values-Syntax

  - Mengenweises Einfügen

    insert into <Relationen-Name>
    [(<Attributname> [, <Attributname>]*)]
    <SQL-Anfrage>

- **Beispiel**

  insert into BestOfPMZ (pnr, mnr, Note)
  select from PMZuteilung where note < 3;

Was bei values wirklich passiert!

- **values-Syntax erzeugt konstante Relation**
  - values (<Konstante> [, Konstante]*)[, (<Konstante> [, Konstante]*)]*
- **Beispiel**
  - values (1, ’eins’), (2, ’zwei’), (3, ’drei’);
    - Resultat ist eine konstante Relation mit zwei Spalten (mit Namen column1 und column2) und drei Tupeln.
  - Man könnte diese Relation auch mit folgendem SQL-Befehl erzeugen:

```
select 1 as column1, ’eins’ as column2
                union all
select 2, ’zwei’
                union all
select 3, ’drei’
```

Löschen und Ändern

- **Löschen**
  - delete
    from <Relationen-Name>
    [where <Bedingung>]
  - Mit Vorsicht verwenden!
    - Bedingung in der where-Klausel ist entscheidend!
- **Ändern**
  - update <Relationen-Name>
    set <Attributname> = <Ausdruck> [, <Attributname> = <Ausdruck>]*
    [where <Bedingung>]
  - Auch hierbei werden i. A. mehrere Datensätze verändert!
  - Beispiel

```
update PMZuteilung
set note = note - 1
where pnr = 114 and note > 1;
```

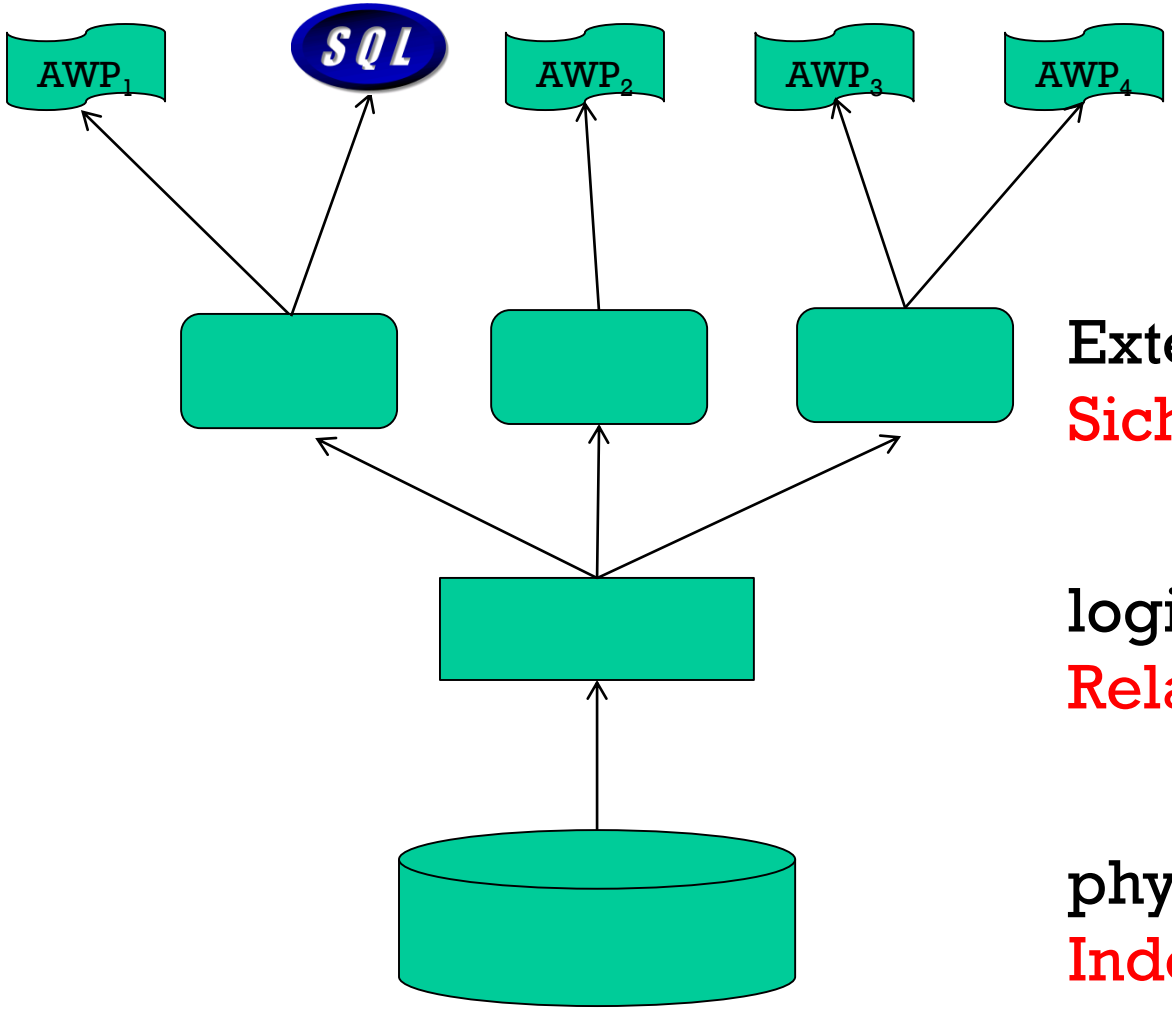
Loading (Bulk Import)

- **Typisches Problem**
  - Viele Datensätze sollen aus einer Textdatei in eine Datenbank eingefügt werden.
    - z. B. im CSV-Datei (Comma-Separated Values)
- **Lösungen**
  1. Aufruf von Insert für jeden Datensatz
  2. Nutzung von Spezialfunktionalität („Loader“) des DBMS

     ➔ Erheblich schneller als der erste Lösungsansatz
- **Loader-Befehl in PostgreSQL: copy**
  - Beispiel (für Windows)

    copy Maschinen from ‘D://Machinen.txt‘
  - Andere DBMS bieten ähnliche Möglichkeiten an!

3.6 Anlegen von Datenstrukturen



Externe Ebenen:
Sichten

logische Ebene:
Relationen

physische Ebene:
Indexe

3.6.1 Indexe

- **Indexe sind Bestandteil der physischen Ebene.**
  - Vorteile
    - Verbesserung der Antwortzeiten
      - Statt O(n) nur noch O(log n)
    - Keinen Einfluss auf die Semantik einer Anfrage
      - Maß für die Effizienz in Datenbanken ist immer noch die Anzahl der Zugriffe auf den Externspeicher.
- **Realisierung eines Index als Suchbaum**
  - Spezielle Suchbäume, die für Daten auf dem Externspeicher entwickelt wurden.
    - B+-Bäume → siehe später
  - Ein Index benötigt eine Ordnungsrelation
    - Typischerweise durch Angabe eines oder mehrerer Attribute (wie in der order-by-Klausel).

Anlegen eines Index

- **Anlegen eines Index**
  - create [unique] index <Index-Name> on <Relationen-Name>(<Attributname> [<Ordnung>] [,<Attributname>[<Ordnung>]]*)
  - <Ordnung>::= Asc | Desc
  - Option unique
    - Es gibt keine zwei Tupel mit den gleichen Werten in den Attributen, auf denen der Index erzeugt wurde.
- **Indexe und Speicherung der Relation**
  - Ein normaler Index verändert nicht die Speicherorganisation der Relation.
    - *Es können deshalb mehrere Indexe pro Relation angelegt werden.*
  - In anderen DBMS besteht die Möglichkeit pro Relation einen ClusterIndex anzulegen, der dann auch die Speicherung der Relation bestimmt.

Beispiel (Index)

- **Beispiel:**

  create unique index PersonalIndex on
  Personal (PName,Vorname);

  - Was ist das Ordnungskriterium?

- **Löschen eines Index**

  drop index <Index-Name>

3.6.2 Sichten

- **Motivation**
  - Datenschutz
    - Durch Sichten (engl. views) bekommen Benutzer einen spezifischen Ausschnitt der Datenbank zugeordnet.
  - Logische Datenunabhängigkeit
    - Durch Sichten können Änderungen an den Relationen vorgenommen werden, ohne dass die Änderung auf Anwendungsprogramme sich durchschlägt.

Anlegen und Löschen

- **Sichten anlegen**
  - create view <Sichtname> [(<Attributname>[,<Attributname>]*)] as <Subquery> [with check option]
  - Ein Sicht ist ein ***rein logisches*** Konstrukt.
    - Die Anfrage wird nur definiert und nicht ausgeführt.
    - Anfrageergebnisse werden nicht gespeichert!
  - Beispiel:

```
create view Top_PMZ as
        select * from PMZuteilung where Note < 3
        with check option;
```

- **Sichten löschen**
  - drop view <Sicht-Name>

Sichten und (lesende) Anfragen

- **Bei lesenden Anfragen**
  - Keine Unterscheidung zwischen Sichten und Relationen
- **Umsetzung bei einer Anfrage**
  - DBMS erkennt beim Übersetzen der Anfrage, dass eine Sicht benutzt wird.
    - Der Name der Sicht wird dann durch deren Definition ersetzt.

      ➔ Es entsteht dabei eine Unteranfrage.
  - Beispiel

```
select * from TopPMZ where mnr = 88;
```

```
select *
from (select * from PMZuteilung where Note < 3) t
where mnr = 88;
```

Sichten und Änderungsoperationen

- **Anforderung**
  - kein Unterschied zwischen Sicht und Relation
  - ➔ Unterstützung von Änderungsoperationen in Sichten.
- **Problem**
  - Beim „Einfügen eines Tupels in eine Sicht“ muss das Einfügen an die Relationen weiterdelegiert werden.
- **Standard erlaubt Änderungen auf Sichten.**
  - Schlüsselwort **with check option**
    - Nur die Datensätze können in eine Sicht eingefügt werden, die bei einer Suche wieder gefunden werden können.
      - ➔ Dies ist die einzig sinnvolle Variante einer Sicht.

Updates nicht immer möglich

- **Problem**
  - Eindeutige Delegation der Änderung an Relationen
  - Beispiel
    - create view agg_view as
      select pnr, count(*) from PMZuteilung group by pnr;
    - insert into agg_view values (77, **42**);
- **Oracle unterstützt Änderungen nur dann, wenn**
    - keine Aggregatfunktion
    - keine Anweisungen mit distinct, group by, having, union und minus
    - from-Klausel mit nur eine Relation
    - ein Schüssel der Relationen in der select-Klausel
  - Dies ist teilweise noch zu restriktiv!
    - Welche Sichten lassen noch Änderungen zu?

3.6.3 Relationen, ...

- **Wiederholung**
  - Anlagen einer neuen Relation

    create table <table_name> ….

- **Da gibt es noch ein paar Details**
  - Anlegen eines Namenraums
  - Anlegen eines Wertebereichs
  - Definition von Integritätsbedingungen
  - ….

Namensräume

- **Motivation**
  - Vermeidung von Namenskonflikten bei Tabellen, Sichten und Indexen
- **Anlegen eines neuen Namensraums**
  - create schema <Name> [authorization <Benutzer>] [schema_element]*
  - Beispiel

    create schema MyERP

    create table PMZuteilung ….

    create view TopPMZ ….

    create index indx_pmz ….

Wertebereiche

- **Anlegen neuer Wertebereiche**
  - Einschränkung bestehender Datentypen durch Hinzufügen von Integritätsbedingungen.
  - Verwendung in verschiedenen Relationen
- **Syntax**
  - create domain <Name> [as] <Datentyp> [<Defaultwert>] [<Integritätsbedingung>]*
- **Beispiel:**
  - create domain Adresse varchar(50) default 'Marburg'
- **Weitere Funktionalität**
  - alter domain …
  - drop domain …

Verfügbare Datentypen

- **Wie in einer Programmiersprache, so gibt es auch in SQL viele Datentypen.**
- **Eine kleine Auswahl (aus dem SQL Standard)**
  - bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time, timestamp, xml, json
- **Jedes System hat zusätzlich noch spezifische Datentypen**
  - PostgeSQL → siehe Benutzermanual
    - Beispiel: serial

3.7 Integritätsbedingungen

- **Anforderung**
  - Einschränkung der Datenbankzustände auf diejenigen, die tatsächlich nur in der realen Welt existieren können.
  - ➔ Umsetzung durch **Integritätsbedingungen**
    - Definition bei der Erstellung von Relationen (Wertebereichen)
- **Vorteile**
  - Einmalige Definition der Bedingungen in der Datenbank.
  - Automatische Überprüfung im Hintergrund
    - Keine lästige Überprüfung der Bedingungen durch AWPs
  - Verzögerte Überprüfung der Integritätsbedingungen aus Effizienzgründen
    - z. B. bei Masseneingabe von Daten

Klassen von Integritätsbedingungen

- **Unterscheidung zwischen**
  - **statische Bedingungen**
    - Definiert der erlaubten Datenbankzustände
  - **dynamische Bedingungen**
    - Definition der erlaubten Zustandsänderungen in einer Datenbank.
      - Zwei Zustände können möglich sein, aber nicht der Übergang des einen Zustands in den anderen.

Statische Integritätsbedingungen

- Primärschlüssel (primary key)
- Eindeutigkeit (unique)
- Fremdschlüssel (foreign key)
- Check-Bedingungen

- Zeitpunkt der Überprüfung von Integritätsbedinungen

- Verwaltung von statischen Integritätsbedinungen

- Integritätsbedingungen für Datenbanken

3.7.1 Primärschlüssel

- **Schlüssel in einer Relation (mittels primary key)**
  - Syntax
    - Wenn der Schlüssel nur ein Attribut umfasst, kann primary key direkt an das Ende der Attributdeklaration geschrieben werden.

      → Beispiel für Bedingung für Attribute
    - Wenn der Schlüssel mehrere Attribute umfasst, gibt es eine eigene Klausel primary key (A_1,…,A_n)

      → Beispiel für Bedingung für Relationen
    - Es gibt nur einen Primärschlüssel in einer Relation.
  - Semantik
    - Der Primärschlüssel ist eindeutig.
    - Die Attribute dürfen **nicht** den Wert null annehmen.

3.7.2 Eindeutige Attribute

- **Wenn es neben dem Primärschlüssel noch ein anderes eindeutiges Attribut gibt:**
  - Syntax
    - Verwendung des Schlüsselworts unique
  - Semantik (SQL´92)
    - Im Gegensatz zum Primärschlüssel ist der Wert null bei unique-Attributen erlaubt.
    - Es können mehrere Tupel existieren, die im unique-Attribut den Wert null besitzen.
- **Durch Angabe von unique und primary key wird automatisch ein Index erstellt.**
  - Warum?

3.7.3 Fremdschlüssel

- **Definition**
  - Seien R und S Relationen mit Schema R_{RS} und S_{RS}. Sei $K \subseteq R_{RS}$ Primärschlüssel von R. Dann wird $F \subseteq S_{RS}$ **Fremdschlüssel von S** genannt, falls zu jedem Tupel $v \in S$ eine der folgenden Bedingungen gilt:
    - v[F] = NULL
    - es gibt ein Tupel $u \in R$, so dass v[F] = u[K] gilt.
- **Probleme bei Verletzung der Fremdschlüsseleigenschaft (referentiellen Integrität)**
    - Angestellte arbeiten in Abteilung, die es nicht gibt.
    - Angestellte sind an Maschinen ausgebildet, die nicht existieren.

Anlegen einer Fremdschlüsselbedingung

- Voraussetzung:
  - pnr ist Primärschlüssel oder unique in Personal
- Syntax
  - create table PMZuteilung(pnr int references Personal(pnr), …)

  oder
  - create table PMZuteilung (pnr int,,…,
    foreign key(pnr) references Personal(pnr));

  oder
  - create table PMZuteilung(pnr int,,…,
    FK_PNR foreign key(pnr) references Personal(pnr))
    - In diesem Fall wird die Bedingung mit dem Namen FK_PNR versehen.

      ➔ Temporäres Abschalten der Bedingung möglich

Überprüfung der Fremdschlüsselbedingung

- **Ausgangssituation**
  - Relation r mit Primärschlüssel K
  - Relation s mit Fremdschlüssel F → r(K)
- **Einhaltung referentieller Integrität bei Änderungen**
  - Einfügen eines Tupels v in Relation s.
    - wird akzeptiert, falls $v[F] \in \pi_K(r)$
  - Verändern des Attributwerts eines Tupels v aus S.
    - wird akzeptiert, falls …
  - Löschen eines Tupels u aus der Relation r.
    - wird akzeptiert, falls …
  - Verändern des Schlüssel von einem Tupel $u \in$ r.
    - wird akzeptiert, falls …

Optionen in SQL

- Defaulteinstellung
  - Löschen eines Tupels u aus einer Relation r ist i.A. nicht möglich, falls es noch ein Tupel aus anderen Relationen gibt, die über einen Fremdschlüssel an r gebunden sind.
- Kaskadierendes Löschen/Ändern
  - Wenn ein Tupel u aus einer Relation r gelöscht/geändert wird, können auch Datensätze aus anderen Relationen automatisch gelöscht/geändert werden, die sich über einen Fremdschlüssel auf das Tupel r beziehen.
- Setzen auf Null
  - Wird ein Tupel gelöscht, wird der Wert aller davon abhängigen Attribute auf null gesetzt.
- Beispiel
  - create table PMZuteilung(
    pnr int constraint fk_personal references Personal(pnr)
    on delete cascade on update set null,
    …
)

3.7.4 Check-Bedingungen

- **Beispiel:**

```
create table PMZuteilung(
      pnr int,
      mnr int,
      note int check(note > 0 and note < 7),
  …
  )
```

  - Diese Integritätsbedingungen können entsprechend den Bedingungen in einer where-Klausel von SQL definiert werden.
    - Dabei kann man sich direkt auf das Attribut beziehen.
    - Unteranfragen sind aber in Systemen nicht erlaubt.
      - Warum?

3.7.4 Zeitpunkt der Überprüfung

- In SQL gibt es noch die Möglichkeit den Zeitpunkt der Überprüfung einer Integritätsbedingung zu beeinflussen.
- Motivation
  - Sicherstellung, dass die durchschnittlichen Note eines Mitarbeiters genügend gut ist.
    - Jedes Quartal werden alle Noten angepasst.
- Anforderung
  - Bündelung von mehreren Operationen zu einer Einheit (**Transaktion → siehe späteres Kapitel)**
  - Überprüfung der Integritätsbedingung nur am Beginn / Ende einer Transaktion

Optionen in SQL

- Eine Integritätsbedingung kann nun mit folgenden Schlüsselworten versehen werden:
  - not deferrable
    - Sofortige Überprüfung nach einer Änderung (immer!)
  - deferrable
    - Verzögerte Überprüfung möglich
  - deferrable initially deferred
    - Überprüfung nur am Ende der Transaktion.
  - deferrable initially immediate
    - Überprüfung vor der Änderung.
- Eine mit dem Schlüsselwort deferrable versehene Bedingung mit einem Namen, sagen wir einfach MeineBedingung, kann zur Laufzeit angepasst werden:
  - set constraint MeineBedingung deferred
  - set constraint MeineBedingung immediate

Bedingungen für Attribute

- **Definition der Bedingungen** erfolgt direkt bei der Deklaration des Attributs.
  - Beispiele
    - create table Personal (pnr int primary key,
      PName char(20) unique, …)

- Dadurch findet die **Überprüfung dieser Bedingungen** immer dann statt, wenn das Attribut verändert wird.

Bedingungen für Relationen

- **Motivation**
  - Einschränkung der Werte, die ein Tupel bzgl. seine verschiedenen Attribute annehmen darf.
- **Auslöser für die Überprüfung**
  - Falls ein Tupel in die Relation eingefügt wird oder ein bestehendes Tupel geändert wird.
- **Erstellen solcher Bedingungen**
  - Die Deklaration einer solchen Bedingung erfolgt direkt bei der Deklaration der Relation oder zu einem späteren Zeitpunkt. Beispiel:
    - Stelle sicher, dass die Note besser als 5 ist.
      alter table PMZuteilung add check(5 > note)
      - Unteranfragen werden von einigen DBMS unterstützt, aber nicht von PostgreSQL!

3.7.5 Verwalten von Integritätsbedingungen

- Integritätsbedingungen können in SQL durch das Schlüsselworts constraint implementiert und dann mit einem Namen versehen werden.

- Hinzufügen/Löschen von Integritätsbedingungen
  - alter table PMZuteilung

    add constraint plus_const check (note > 1)
    - Dies ist stets eine Bedingung für die Relation.
  - alter table Personal

    add constraint name_unique unique PName
  - alter table <Name> drop constraint <CName>
    - Löschen wird für beliebige Bedingungen unterstützt.

3.7.6 Bedingungen für Datenbanken

- **SQL92 Standard**
  - Unterstützung von sogenannten **Assertions**
    - Eigenständige Datenbankobjekte zur Formulierung von Integritätsbedingungen.
      - Syntax
        create assertion <name> check <Bedingung>
      - Unterstützung von Bedingungen über mehrere Tabellen
      - Unterstützung von Unteranfragen
- **Umsetzung in Systemen**
  - Nahezu keine Unterstützung für Assertions
  - Alternative ist die Implementierung mit einem Trigger → siehe späteres Kapitel

Beispiel

- **So könnte eine Assertion aussehen.**

```
create assertionTopPerformer check (
        not exists (
                select * from Personal
                where Lohn > 60000 and pnr in
                        (select pnr
                        from PMZuteilung
                         where note > 3)));
```

Zusammenfassung

- **Vergleich von verschiedenen Bedingungen**

Bedingung für	Ort der Deklaration	Auslösen durch	Unteranfragen
Attribute	Attribut	Änderung des Attributs	nein
Relationen	Relation	Änderung eines Datensatzes	Nein
Datenbanken	unabhängig	Änderung der Relation	ja

- **Standard für Integritätsbedingungen**
  - Nur teilweise in den meisten DBMS implementiert.