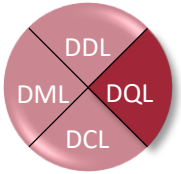
Programmierung von Systemen – 13 – SQL 2

Matthias Tichy & Stefan Götz | SoSe 2020

Ziele

- Umsetzung von JOINS und restliche Algebra-Operatoren (Vereinigung, Differenz, Schnitt) in SQL kennen
- Subqueries und Existenzbedingungen formulieren können
- Gruppierungs- und Aggregationsfunktionen kennen
- rekursive Anfragen gehört haben

Vereinigung, Differenz, Schnitt



SFW-Ausdruck
TABLE *Relationsname*
Tabellenausdruck

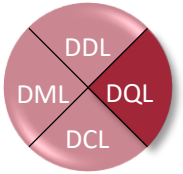
UNION
EXCEPT
INTERSECT

[**ALL**] **CORRESPONDING** [**BY** $Attr_1$, [$Attr_2$, …]]

SFW-Ausdruck
TABLE *Relationsname*
Tabellenausdruck

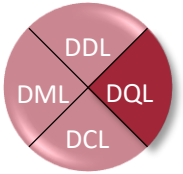
[**ORDER BY** *Attributliste*]

Vereinigung, Differenz, Schnitt



- Anzahl und Typ der Spalten der beteiligten Relationen müssen identisch sein
- **CORRESPONDING ohne BY** …
  Verknüpfung nur über gemeinsame Attribute
- **CORRESPONDING mit BY** $Attr_1$, $Attr_2$,
  Zunächst Projektion auf die angegebenen Attribute
- **ALL** verhindert automatische Duplikateliminierung
- Falls Attributnamen der „Operanden-Relationen“ unterschiedlich, gelten die Attributnamen der ersten Operanden-Relation

Beispiele



S27: *„Gib Firmenname und Stadt von allen Firmen aus, zu denen wir geschäftliche Kontakte als Kunden oder Lieferanten haben, sortiert nach Firmenname und Stadt“*

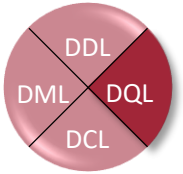
```
SELECT KdName AS Firmenname, KdStadt AS Stadt
FROM      Kunden

UNION

SELECT LiefName AS Firmenname, LiefStadt AS Stadt
FROM      Lieferanten

ORDER BY Firmenname, Stadt
```

Beispiele



S28: *„Gib alle Firmen aus (Ausgabe: Firmenname, sortiert), die sowohl Kunden als auch Lieferanten unserer Firma sind“*

(Vereinfachende Annahme:
Der Firmenname sei in der Kunden- und der Lieferanten-Relation jeweils exakt gleich geschrieben)

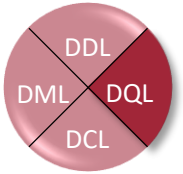
```
SELECT      LiefName AS Firmenname
FROM        Lieferanten

INTERSECT

SELECT      KdName AS Firmenname
FROM        Kunden

ORDER BY Firmenname
```

Beispiele



S29: *„Ermittle alle Teile (Ausgabe komplettes Teile-Tupel), zu denen es keinen (externen) Preis gibt“*

ohne Verwendung von OUTER JOIN

Vorüberlegungen zur Lösung:

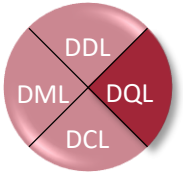
Lösung = Menge aller Teile ./. Menge der Teile, die einen externen Preis haben

wie bestimmen?

Join zwischen Teile und Preisliste,
Preisliste wirkt hierbei als „Filter“,
Projektion des Resultats auf Teile-Attribute

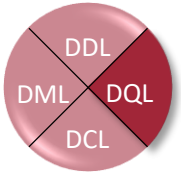
```
SELECT   * FROM Teile
EXCEPT Corresponding By (TeileNr)
SELECT   * FROM Teile NATURAL JOIN Preisliste
```

Kalkül-orientierte Konstrukte



- Bisher: Umsetzung von Operatoren der Relationenalgebra
- Keine Existenzbedingungen in Abfragen
- Subqueries/Existenzbedingungen aus Relationenkalkül übernommen
- Neuigkeit: Subqueries innerhalb der WHERE-Klausel

Existenzbedingungen

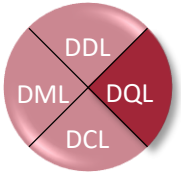


Variante 1

```
SELECT      Attributliste
FROM        Rel1, Rel2, …, Reln
WHERE NOT   EXISTS ( Tabellenausdruck )
```

Der Subquery-Ausdruck (EXISTS) liefert nur TRUE (mind. ein Treffer gefunden) oder FALSE (keine Treffer)

Existenzbedingungen



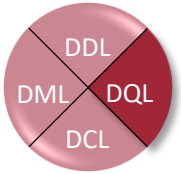
Variante 2

SELECT *Attributliste*
FROM $Rel_1, Rel_2, \ldots, Rel_n$

WHERE { $Rel_i.Attr$ | ($Rel_i.Attr, R_j.Attr$) } { = | <> | < | <= | >= | > } [{ **ANY** | **ALL** }] (*Tabellenausdruck*)

Ohne ANY-/ALL-Zusatz muss der Subquery-Ausdruck genau einen Attributwert oder ein Tupel zurückliefern. Dieser wird als konstanter Vergleichswert im äußeren Query-Ausdruck verwendet.

Existenzbedingungen

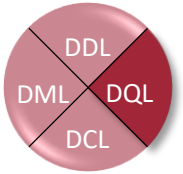


Variante 3

SELECT *Attributliste*
FROM $Rel_1, Rel_2, \ldots, Rel_n$
WHERE { $Rel_i.Attr$ | ($Rel_i.Attr, R_j.Attr$) } [**NOT**] **IN** (*Tabellenausdruck*)

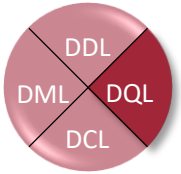
- Variante 3 ist Spezialfall von Variante 2
- **IN** ist äquivalent zu **= ANY**
- **NOT IN** ist äquivalent zu **<> ALL**

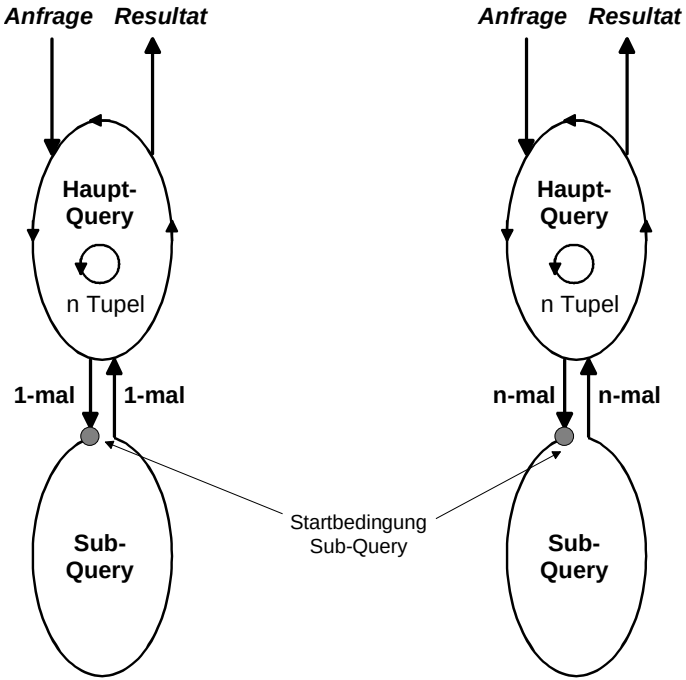
Anmerkungen



- Subquery-Ausdrücke können geschachtelt sein, d.h. sie können wiederum Subquery-Ausdrücke enthalten
- In Subquery-Ausdrücken der Variante 1 wird in der Regel auf Relationen der äußeren Query-Ausdrücke Bezug genommen: korrelierte Subqueries
- Bei den anderen beiden Varianten ist dies nicht der Fall: unkorrelierte Subqueries

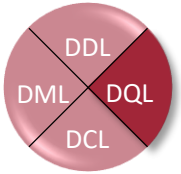
Visualisierung





unkorrelierte Subquery **korrelierte Subquery**

Beispiele

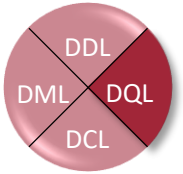


S30: *„Gib alle Kunden aus, die uns aktuell Aufträge erteilt haben“*

Formuliert mit Subquery-Variante 1

```
SELECT   *
FROM     Kunden AS k
WHERE    EXISTS
           ( SELECT     *
             FROM   Auftraege AS a
             WHERE a.KdNr = k.KdNr )
```

Beispiele

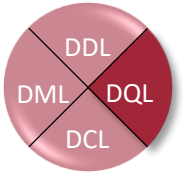


S30: *„Gib alle Kunden aus, die uns aktuell Aufträge erteilt haben“*

Formuliert mit Subquery-Variante 2

```
SELECT   *
FROM     Kunden
WHERE    KdNr = ANY
           (SELECT KdNr
            FROM   Auftraege)
```

Beispiele

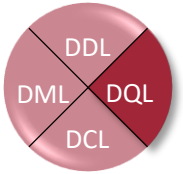


S30: *„Gib alle Kunden aus, die uns aktuell Aufträge erteilt haben“*

Formuliert mit Subquery-Variante 3

```
SELECT   *
FROM     Kunden
WHERE    KdNr IN
           (SELECT  KdNr
            FROM    Auftraege)
```

Negation

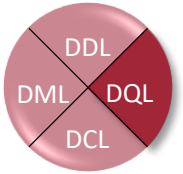


S31: *„Gib alle Kunden aus, die uns aktuell keine Aufträge erteilt haben“*

Formuliert mit Subquery-Variante 1

```
SELECT   *
FROM     Kunden AS k
WHERE    NOT EXISTS
           ( SELECT     *
             FROM   Auftraege AS a
             WHERE a.KdNr = k.KdNr )
```

Negation

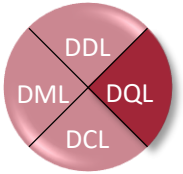


S31: *„Gib alle Kunden aus, die uns aktuell keine Aufträge erteilt haben“*

Formuliert mit Subquery-Variante 2

```
SELECT   *
FROM     Kunden
WHERE    KdNr <> ALL
           (SELECT KdNr
            FROM   Auftraege)
```

Negation

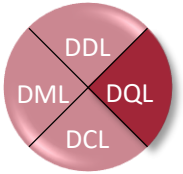


S31: *„Gib alle Kunden aus, die uns aktuell keine Aufträge erteilt haben“*

Formuliert mit Subquery-Variante 3

```
SELECT   *
FROM     Kunden
WHERE    KdNr NOT IN
           ( SELECT KdNr
             FROM   Auftraege)
```

Beispiele

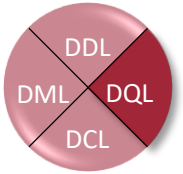


S32: *Gib den Mitarbeiter (gesamtes Tupel) mit dem höchsten Gehalt aus“*

Formuliert mit Subquery-Variante 1

```
SELECT   *
FROM     Mitarbeiter AS m1
WHERE    NOT EXISTS
           (SELECT *
            FROM   Mitarbeiter AS   m2
            WHERE  m2.Gehalt > m1.Gehalt )
```

Beispiele

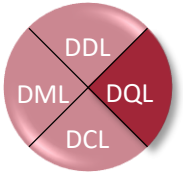


S32: *Gib den Mitarbeiter (gesamtes Tupel) mit dem höchsten Gehalt aus“*

Formuliert mit Subquery-Variante 2

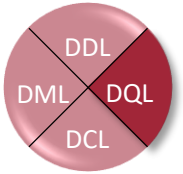
```
SELECT   *
FROM     Mitarbeiter
WHERE    Gehalt >= ALL
           (SELECT Gehalt
            FROM   Mitarbeiter )
```

Aggregationsfunktionen



Funktion	Bedeutung
COUNT(*)	Anzahl der Tupel in der Relation
COUNT ([DISTINCT] *Attributname*)	Anzahl der [unterschiedl.] Attributwerte
MAX(*Attributname*)	Maximum der Attributwerte
MIN(*Attributname*)	Minimum der Attributwerte
AVG([DISTINCT] *Attributname*)	Durchschnittswert der [unterschiedl.] Attributwerte
SUM([DISTINCT] *Attributname*)	Summe der [unterschiedl.] Attributwerte

Beispiele



S33: *„Gib die Anzahl aller Lieferanten aus“*

```
SELECT COUNT(*)
FROM   Lieferanten
```

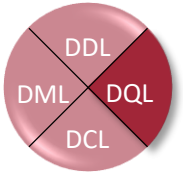
S34: *„Gib die Anzahl aller Lieferanten aus Ulm aus“*

```
SELECT COUNT(*)
FROM   Lieferanten
WHERE LiefStadt = 'Ulm'
```

S35: *„Gib das durchschnittl. Gehalt aller Mitarbeiter aus“*

```
SELECT AVG(Gehalt)
FROM   Mitarbeiter
```

Beispiele

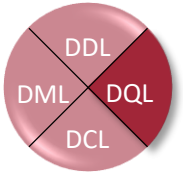


Bei Verwendung einer Aggregatfunktion wird genau ein Resultat-Tupel erzeugt!

S36: *„Gib das höchste, das niedrigste und das durchschnittliche Gehalt sowie die Summe aller Gehälter aus“*

```
SELECT  MAX(Gehalt) AS  Maximalgehalt,
        MIN(Gehalt) AS  Minimalgehalt,
        AVG(Gehalt) AS  Durchschnittsgehalt,
        SUM(Gehalt) AS  Gehaltssumme
FROM    Mitarbeiter
```

Beispiele



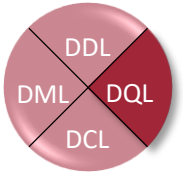
S37: *„Gib die Personalnummer und das Gehalt des Mitarbeiters (oder der Mitarbeiter) mit dem niedrigsten Gehalt aus“*

```
SELECT  PersNr, Gehalt
FROM    Mitarbeiter
WHERE   Gehalt =
        ( SELECT  MIN(Gehalt)
          FROM    Mitarbeiter )
```

S38: *„Gib aus, aus wie vielen verschiedenen Städten unsere Kunden kommen“*

```
SELECT  COUNT( DISTINCT KdStadt )
FROM    Kunden
```

Beispiele

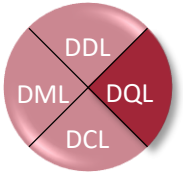


S39: *„Gib alle Kunden aus, die uns aktuell Aufträge erteilt haben“*

vorher Lösung mittels Subqueries, jetzt mit Aggregrationsfunktion:

```
SELECT  *
FROM    Kunden AS k
WHERE   ( SELECT    COUNT(*)
          FROM      Auftraege AS a
          WHERE     a.KdNr = k.KdNr ) > 0
```

Gruppierung



S40: *„Gib aus, aus welchen Städten jeweils wie viele Kunden kommen“*

```
SELECT    KdStadt, COUNT … ?
FROM      Kunden
```

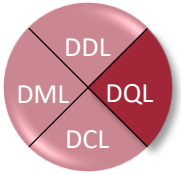
Lösung: Gruppierung

```
SELECT    KdStadt, COUNT(*) AS Anzahl
FROM      Kunden
GROUP BY  KdStadt
```

Alternativ:

```
SELECT    KdStadt, COUNT(KdStadt) AS Anzahl
FROM      Kunden
GROUP BY  KdStadt
```

Gruppierungsfunktionen



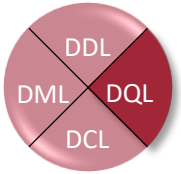

```
SELECT      Attributliste
FROM        Tabellenausdruck
[ WHERE     ( Selektionsbedingung ) ]
[ GROUP BY  Gruppierungsattribut(e) ]
[ HAVING    Gruppenbedingung ]
[ ORDER BY  Attributliste ]
```

Beachte:

- **WHERE** selektiert Zeilen
- **HAVING** wählt Gruppen aus

KDSTADT	ANZAHL
Augsburg	1
Bochum	1
Bretten	1
Flensburg	1
Hamburg	1
Karlsruhe	1

Gruppierungsfunktionen

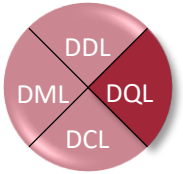


- **GROUP BY** bewirkt (interne) Gruppierung der Ergebnis-Relation entsprechend dem Gruppierungsattribut bzw. den –attributen (➔ Menge von Mengen)
- Aggregationsfunktionen werden jeweils auf diesen Teilmengen angewandt
- Joins werden vor Anwendung der Gruppierung ausgeführt
- Abarbeitungsreihenfolge:
  FROM – WHERE – GROUP BY – HAVING – SELECT – ORDER BY
- In der SELECT-Klausel können bei Angabe von GROUP BY neben Aggregationsfunktionen auch normale Attribute stehen, wenn nach diesen (ebenfalls) gruppiert wurde
- Gruppenbedingungen werden in HAVING-Klausel ausgedrückt; **HAVING** kann nur in Verbindung mit **GROUP BY** auftreten

KDSTADT	ANZAHL
Augsburg	1
Bochum	1

Bretten	1
Flensburg	1
Hamburg	1
Karlsruhe	1

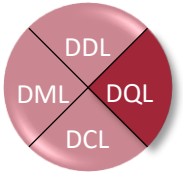
Beispiele



S41: *„Gib zu allen Bestellungen (alle Attribute) die Anzahl der Bestellpositionen aus“*

```
SELECT    b.*, COUNT(*) AS AnzahlBestPos
FROM      Bestellungen AS b
          JOIN BestellPos AS p ON b.BestNr = p.BestNr
GROUP BY  b.BestNr, b.LiefNr, b.BestDatum
```

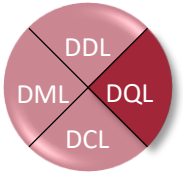
Beispiele



S42: *„Gib alle Lieferanten (LiefNr, LiefName, Anzahl_Teile) aus, die weniger als 3 Teile liefern können“*

```
SELECT      l.LiefNr, l.LiefName,
            COUNT(*) AS Anzahl_Teile
FROM        Lieferanten AS l
            JOIN Liefert AS lf ON l.LiefNr = lf.LiefNr
GROUP BY    l.LiefNr, LiefName
HAVING      COUNT(*) < 3
```

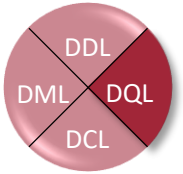
Beispiele



S43: *„Gib aus, welche externen Lieferanten (d.h. LiefNr > 0) mehr als 3 Teile in ihrem Lieferprogramm für uns haben, die mehr als 2,00 EUR kosten.*
Gib LiefNr, LiefName und Anzahl dieser Teile sowie deren maximalen Preis aus.“

```
SELECT   l.LiefNr, l.LiefName, COUNT(*) AS Anzahl,
         MAX(lf.Preis) AS Maximalpreis
FROM     Lieferanten AS l
         JOIN Liefert AS lf ON l.LiefNr = lf.LiefNr
WHERE    l.LiefNr > 0 AND lf.Preis > 2.00
GROUP BY l.LiefNr, l.LiefName
HAVING   COUNT(*) > 3
```

Beispiele



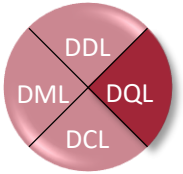
S44: *„Gib alle Bestellungen (LiefNr, BestNr, BestWert) aus, die einen Bestellwert von mehr als 500 EUR haben“*

Vorüberlegungen:

- Bestellwert einer Bestellung = Summe der Werte der Bestellpositionen
- Aber: Relation BestellPos enthält nicht direkt den Wert der Bestellposition
- Sehr nützlich wäre eine Tabelle folgender Art (Warum?):

LIEFNR	BESTNR	BESTDATUM	BESTPOS	TEILENR	FARBE	POSWERT
527	103	2013-02-10	1	K28	1	125.00
527	103	2013-02-10	2	K28	2	75.00
500	104	2013-02-11	1	FA	1	10.00
500	104	2013-02-11	2	FA	2	5.00
572	105	2013-02-15	1	TL134	1	25.20
572	105	2013-02-15	2	TR134	1	25.20
572	105	2013-02-15	3	TL134	2	8.80

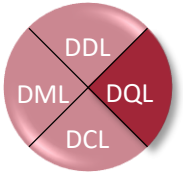
Beispiele



Erzeugung der Zwischentabelle

```
SELECT b.LiefNr, b.BestNr, b.BestDatum,
       bp.BestPos, bp.TeileNr, bp.Farbe,
       bp.Anzahl * l.Preis AS PosWert
FROM   Bestellungen AS b
       JOIN BestellPos AS bp ON b.BestNr = bp.BestNr
       JOIN Liefert AS l ON (b.LiefNr, bp.TeileNr, bp.Farbe)
                          = (l.Liefnr, l.TeileNr, l.Farbe)
```

Beispiele

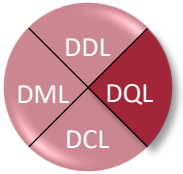


Angenommen, diese Relation existiert und heißt PosWertRelation.

Wie lautet dann die SQL-Formulierung für die Anfrage:
„Gib alle Bestellungen (LiefNr, BestNr, BestWert) aus, die einen Bestellwert von mehr als 500 haben“?

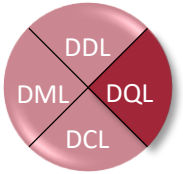
```
SELECT      pw.LiefNr, pw.BestNr,
            SUM(pw.PosWert) AS BestellWert
FROM        PosWertRelation AS pw
GROUP BY    pw.LiefNr, pw.BestNr
HAVING      SUM(pw.PosWert) > 500
```

Gesamtlösung



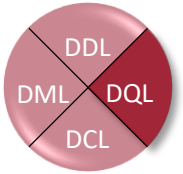

```
SELECT      pw.LiefNr, pw.BestNr,
            SUM(pw.PosWert) AS BestellWert
FROM    ( SELECT  b.LiefNr, b.BestNr, b.BestDatum,
                  bp.BestPos, bp.TeileNr, bp.Farbe,
                  bp.Anzahl * l.Preis AS PosWert
            FROM    Bestellungen AS b
                  JOIN BestellPos AS bp
                     ON b.BestNr = bp.BestNr
                  JOIN Liefert AS l ON (b.LiefNr, bp.TeileNr,
                     bp.Farbe) = (l.Liefnr, l.TeileNr, l.Farbe)
        )  AS pw
GROUP BY    pw.LiefNr, pw.BestNr
HAVING      SUM(pw.PosWert) > 500
```

Vergleich mit Nullwerten



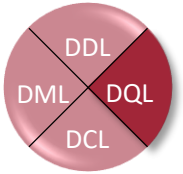

```
...
WHERE { Ausdruck | Attributname } IS [NOT] NULL
```

Behandlung von Nullwerten



- Sortieren: Tupel mit Nullwerten stehen stets am Anfang der Ergebnis-Tabelle, egal ob ASC oder DESC
- arithmetische Ausdrücke: Wenn einer oder beide Operanden nullwertig sind, ist das Ergebnis NULL
- In booleschen Ausdrücken:
  TRUE $\wedge$ NULL $\Rightarrow$ NULL
  TRUE $\vee$ NULL $\Rightarrow$ TRUE
  const VerglOp NULL $\Rightarrow$ Laufzeitfehler
- Bei Gruppierung wird Nullwert wie ein normaler Wert behandelt und bildet ggf. eine eigene Gruppe
- Bei Anwendung von Aggregations-Funktionen werden Nullwerte bei der Auswertung ignoriert

Beispiel



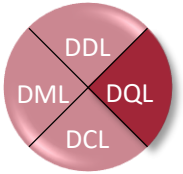
S45: *„Gib für alle Lieferanten aus, wie viele Teile sie für uns liefern können. Ausgabe: LiefNr, LiefName, Anzahl“*

```
SELECT    l.LiefNr, l.LiefName, COUNT(lf.Preis) AS Anzahl
FROM      Lieferanten AS l LEFT OUTER JOIN
          Liefert AS lf ON l.LiefNr = lf.LiefNr
GROUP BY  l.LiefNr, l.LiefName
```

Fragen:

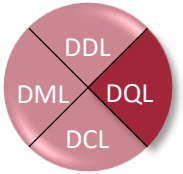
- Hätten wir auch COUNT(*) verwenden können?
- Hätten wir auch andere Attribute in der COUNT()-Funktion verwenden können?

Rekursive Anfragen



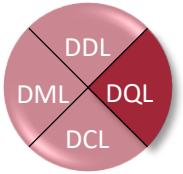
- Es gibt (einfache) Anfragen, die mit relational vollständigen Sprachen nicht formulierbar sind
- Beispiele
  - *Vorfahren* (*ancestors*)
    gegeben: Relation Eltern (Elternteil, Kind)
    gesucht: alle Vorfahren einer Person
  - *Stückliste* (*parts explosion, bill of material* (BOM))
    gegeben: Relation Stueckliste (Oberteil, Unterteil, Anzahl)
    gesucht: alle Einzelteile (evtl. mit Anzahl) zu einem End- oder Zwischenprodukt
  - *Wegsuche* (*path queries*)
    gegeben: Relation Strecke (von, nach, Entfernung)
    gesucht: kürzester oder längster Weg von A nach B

Rekursive Anfragen



- Problem:
  - Relationale Operationen können jeweils nur einen Schritt der Ableitung berechnen
  - Es fehlt ein „Schleifen“-Konstrukt zur Hüllenberechnung („transitive Hülle“)
  - Möglicher Ansatz:
    Rekursive Formulierung ➔ Fixpunktberechnung

Rekursive Anfragen



- Wunsch: Erweiterung von relationalen DBMS um Regeln + Inferenzfähigkeiten
- Ab Ende 70er Jahre diverse Entwicklungen im Forschungsbereich
  ➔ deduktive Datenbanken
  ➔ diverse erweiterte Anfragesprache z.B. das von Prolog abgeleitete Datalog
- Aber Transfer in industrielle Praxis nicht gelungen
  ➔ keine Datalog-Unterstützung in aktuellen DBMS

- „Antwort“ von SQL:
  Rekursive Vereinigung ➔ **`UNION ALL`**

Ziele

- Umsetzung von JOINS und restliche Algebra-Operatoren (Vereinigung, Differenz, Schnitt) in SQL kennen
- Subqueries und Existenzbedingungen formulieren können
- Gruppierungs- und Aggregationsfunktionen kennen
- rekursive Anfragen gehört haben