



**Hasso  
Plattner  
Institut**

IT Systems Engineering | Universität Potsdam

Datenbanksysteme I  
SQL

Felix Naumann

# SQL – Historie

2

- SEQUEL (1974, IBM Research Labs San Jose)
- SEQUEL2 (1976, IBM Research Labs San Jose)
  - System R
- SQL (1982, IBM)
- ANSI-SQL (SQL-86; 1986)
  - Erste genormte Version
- ISO-SQL (SQL-89); 1989;
  - drei Sprachebenen: Level 1, Level 2, + IEF
- (ANSI / ISO) SQL2 (als SQL-92 verabschiedet)
- (ANSI / ISO) SQL3 (als SQL-99 verabschiedet)
  - Objektrelationale Erweiterungen
- (ANSI / ISO) SQL:2003
- Trotz Standardisierung: teilweise Inkompatibilitäten zwischen Systemen der einzelnen Hersteller



# SQL:2003

3

- Part 1: SQL/Framework (92 Seiten)
  - Überblick
- Part 2: SQL/Foundation (1310 Seiten)
  - Datenmodell, DDL, DML, Abfragen
- Part 3: SQL/CLI (Call-Level Interface; 414 Seiten)
  - Zugriff auf DBMS mittels Funktionsaufrufen
- Part 4: SQL/PSM (Persistent Stored Modules); 182 Seiten)
  - Prozedurale Erweiterungen
- Part 9: SQL/MED (Management of External Data; 504 Seiten)
  - Neue Datentypen und Funktionen
- Part 10: SQL/OLB (Object Language Bindings; 382 Seiten)
  - Java
- Part 11: SQL/Schemata (Information and Definition Schemata; 284 Seiten)
- Part 13: SQL/JRT (Java Routines und Types; 212 Seiten)
  - Externe Java Routinen
- Part 14: SQL/XML (XML-related Specifications; 154 Seiten)
  - XML Datentyp und Erweiterung von SQL um XQuery

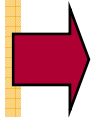
**Zusammen:  
3534 Seiten**

# Motivation

4

- Verbreitetste Datenbankabfragesprache
- Ad-hoc und einfach
- Deklarativ
  - Nicht prozedural
  - Optimierbar
- Very-High-Level language
- Anfragen an relationale Algebra angelehnt
  - Hinzu kommt DML
  
- Achtung: Syntax kann sich von System zu System leicht unterscheiden.
- Achtung: Funktionalität kann sich von System zu System leicht unterscheiden.

5



- Einfache Anfragen
- Anfragen über mehrere Relationen
- Geschachtelte Anfragen
- Operationen auf einer Relation
- Datenbearbeitung (DML)
- Schemata (DDL)
- Sichten



# Beispielschema

6

- Filme(Titel, Jahr, Länge, inFarbe, StudioName, ProduzentID)
- spielt\_in(FilmTitel, FilmJahr, Name)
- Schauspieler(Name, Adresse, Geschlecht, Geburtstag)
- Manager(Name, Adresse, ManagerID, Gehalt)
- Studios(Name, Adresse, VorsitzenderID)

# Der Grundbaustein

7

```
SELECT ...  
FROM ...  
WHERE ...
```

- SELECT \*  
FROM Filme  
WHERE StudioName = ‚Disney‘ AND Jahr= 1990;

Lesereihenfolge (und Schreibreihenfolge):

- FROM: Relation(en) aus denen die Daten stammen
- WHERE: Bedingung(en) an die Daten
- SELECT: Schema der Ergebnisrelation
  - \*: Alle Attribute der Inputrelationen

Ausführung

- Für jedes Tupel aus „Filme“ prüfe die Bedingungen und gebe gültige Tupel aus.

# Projektion in SQL (SELECT)

8

## Spezifikation in der SELECT Klausel

- SELECT \*
  - Alle Attribute
- SELECT Titel, Jahr, inFarbe
  - Projektion auf die drei Attribute

## Erweiterte Projektion

- Umbenennung:
  - SELECT Titel AS Name, Jahr AS Zeit
- Arithmetischer Ausdruck:
  - SELECT Titel, Länge \* 0.016667 AS Stunden
- Konstanten:
  - SELECT Titel, Länge \* 0.016667 AS Stunden, ,std.` AS inStunden



# Selektion in SQL

9

## Spezifikation in der WHERE Klausel

- Bedingungen wie in einer Programmiersprache
- Sechs Vergleichsoperatoren
  - =, <>, <, >, <=, >=
- Operanden
  - Konstanten und Attributnamen
  - Auch Attribute, die nicht in der SELECT Klausel genannt werden.
- Arithmetische Ausdrücke für numerische Attribute
  - Z.B.: (Jahr - 1930) \* (Jahr - 1930) < 100
- Konkatenation für Strings
  - ‚Star‘ || ‚Wars‘ entspricht ‚StarWars‘

# Selektion in SQL

10

Ergebnis ist Boole'scher Wert

- TRUE oder FALSE
- Können mit AND, OR und NOT verknüpft werden.
  - Klammerungen sind erlaubt.

Beispiele

- ```
SELECT Titel
FROM Filme
WHERE Jahr > 1970 AND NOT inFarbe;
```
- ```
SELECT Titel
FROM Filme
WHERE (Jahr > 1970 OR Länge < 90) AND StudioName = ,MGM`;
```

# Stringvergleiche

11

- Datentypen
  - Array fester Länge, Buchstabenliste variabler Länge, Konstanten
- `foo _____ = foo = ,foo``
- Vergleiche mit `<`, `>`, `<=`, `>=`
  - Lexikographischer Vergleich
  - `,fodder` < ,foo``; `,bar` < ,bargain``
- Patternmatching
  - string LIKE pattern bzw. string NOT LIKE pattern
  - Pattern hat spezielle Zeichen
    - ◇ `,%``: Beliebige Sequenz von 0 oder mehr Zeichen
    - ◇ `,_``: Ein beliebiges Zeichen
  - `SELECT Titel FROM Filme WHERE Titel LIKE ,Star _____``;
    - ◇ Star Wars und Star Trek
  - `SELECT Titel FROM Filme WHERE Titel LIKE ♥,%`s%``;

# Datum und Uhrzeit

12

## Spezielle Datentypen und Repräsentationen

- Datumskonstante:
  - `DATE ,YYYY-MM-DD``
  - `DATE ,1948-05-14``
- Zeitkonstante
  - `TIME ,HH:MM:SS.S``
  - `TIME ,15:00:02.5``
- Zeitstempel
  - `TIMESTAMP , 1948-05-14 15:00:02.5``
- Zeitvergleiche
  - `TIME ,15:00:02.5` < TIME ,15:02:02.5`` ergibt TRUE
  - `DATE ,1948-05-14` >= DATE ,1949-11-12`` ergibt FALSE

# Nullwerte

13

- Darstellung: NULL bzw.  $\perp$
- Mögliche Interpretationen
  - Unbekannter Wert
    - ◇ Geburtstag eines Schauspielers
  - Wert unzulässig
    - ◇ Ehegatte eines unverheirateten Schauspielers
  - Wert unterdrückt
    - ◇ Geheime Telefonnummer
- Regeln für Umgang mit Nullwerten
  - Arithmetische Operationen mit NULL ergeben NULL
  - Vergleich mit NULL ergibt Wahrheitswert UNKNOWN
  - NULL ist keine Konstante, sondern erscheint nur als Attributwert
- Beispiel (sei der Wert von x NULL):
  - $x+3$  ergibt NULL.
  - $NULL+3$  ist kein zulässiger Ausdruck.
  - $x = 3$  ergibt UNKNOWN.
- Abfrage von Nullwerten
  - Geburtstag IS NULL

# Wahrheitswerte

14

and	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

or	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

not	
true	false
unknown	unknown
false	true

## Eselsbrücke

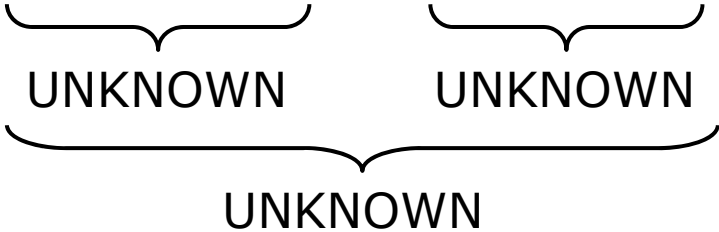
- TRUE = 1, FALSE = 0, UNKNOWN = 1/2
- AND: Minimum der beiden Werte
- OR: Maximum der beiden Werte
- NOT: 1 - Wert
- Beispiel
  - TRUE AND (FALSE OR NOT(UNKNOWN))  
 = MIN(1, MAX(0, (1 - 1/2)))  
 = MIN(1, MAX(0, 1/2))  
 = MIN(1, 1/2) = 1/2.

# Wahrheitswerte

15

Titel	Jahr	Länge	inFarbe	Studio	ProduzentID
Total Recall	1990	NULL	True	Fox	12345

- Überraschendes Verhalten

- `SELECT *`  
`FROM Filme`  
`WHERE Länge <= 90 OR Länge > 90;`  
  
`UNKNOWN`
- Tupel erscheint nicht im Ergebnis.

# Sortierung

16

ORDER BY Klausel ans Ende der Anfrage

- ORDER BY <Attributliste> DESC/ASC

- ASC (aufsteigend) ist default

- SELECT \*

FROM Filme

WHERE StudioName = ‚Disney‘ AND Jahr = 1990

ORDER BY Länge, Titel;

- SELECT \*

FROM Filme

WHERE StudioName = ‚Disney‘ AND Jahr = 1990

ORDER BY Länge ASC, Titel DESC;



# Groß- und Kleinschreibung

17

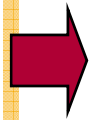
In SQL wird Groß- und Kleinschreibung nicht beachtet

- From = FROM = from = FrOm
- Auch bei Attribut- und Relationennamen
- Aber
  - ‚FROM‘ ≠ ‚from‘ ≠ from

# Überblick

18

- Einfache Anfragen
- Anfragen über mehrere Relationen
- Geschachtelte Anfragen
- Operationen auf einer Relation
- Datenbearbeitung (DML)
- Schemata (DDL)
- Sichten



# Motivation

19

- Hauptkraft der Relationalen Algebra ist die Kombination von Relationen
- Erst mit mehreren Relationen sind viele interessante Anfragen möglich.
- Nennung der beteiligten Relationen in der FROM Klausel

# Kreuzprodukt und Join

20

- Filme(Titel, Jahr, Länge, inFarbe, StudioName, ProduzentID)
- Manager(Name, Adresse, ManagerID, Gehalt)
  
- SELECT Name  
FROM Filme, Manager  
WHERE Titel = ‚Star Wars‘ AND ProduzentID = ManagerID;
- Semantik
  - Betrachte jedes Tupelpaar der Relationen Filme und Manager.
  - Wende Bedingung der WHERE Klausel auf jedes Tupelpaar an
  - Falls Bedingung erfüllt, produziere ein Ergebnistupel.
- Kreuzprodukt gefolgt von Selektion: Join

# Uneindeutige Attributnamen

21

- Schauspieler(Name, Adresse, Geschlecht, Geburtstag)
- Manager(Name, Adresse, ManagerID, Gehalt)

Bei gleichen Attributnamen aus mehreren beteiligten Relationen:

- Relationenname als Präfix:
  - `SELECT Schauspieler.Name, Manager.Name  
FROM Schauspieler, Manager  
WHERE Schauspieler.Adresse = Manager.Adresse;`
- Präfix ist auch erlaubt wenn Attributname eindeutig ist.
  - Erleichtert das Lesen von SQL Anfragen

# Tupelvariablen

22

Zur eindeutigen Kennzeichnung von Tupeln beteiligter Relationen

- „Alias“ einer Relation
- Insbesondere: Bei der mehrfachen Verwendung einer Relation in einer Anfrage
- Gesucht: Schauspieler, die zusammen leben
  - `SELECT Star1.Name, Star2.Name`  
`FROM Schauspieler Star1, Schauspieler Star2`  
`WHERE Star1.Adresse = Star2.Adresse`
- Auch sinnvoll als abkürzenden Schreibweise
  - `SELECT S.Name, M.Name`  
`FROM Schauspieler S, Manager M`  
`WHERE S.Adresse = M.Adresse;`
- Ohne explizites Angeben einer Tupelvariablen wird der Attributname selbst als Tupelvariable verwendet.

# Tupelvariablen

23

Name	Adresse	Geschlecht	Geburt
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Brad Pitt	123 Maple St., Hollywood	M	7/7/77

- ❑ SELECT Star1.Name, Star2.Name  
FROM Schauspieler Star1, Schauspieler Star2  
WHERE Star1.Adresse = Star2.Adresse;
- ❑ SELECT Star1.Name, Star2.Name  
FROM Schauspieler Star1, Schauspieler Star2  
WHERE Star1.Adresse = Star2.Adresse  
AND Star1.Name <> Star2.Name;
- ❑ SELECT Star1.Name, Star2.Name  
FROM Schauspieler Star1, Schauspieler Star2  
WHERE Star1.Adresse = Star2.Adresse  
AND Star1.Name < Star2.Name;

Star1.Name	Star2.Name
Carrie Fisher	Carrie Fisher
Carrie Fisher	Brad Pitt
Brad Pitt	Carrie Fisher
Brad Pitt	Brad Pitt
Mark Hamill	Mark Hamill

Star1.Name	Star2.Name
Carrie Fisher	Brad Pitt
Brad Pitt	Carrie Fisher

Star1.Name	Star2.Name
Brad Pitt	Carrie Fisher

# Interpretation von Anfragen

24

## Drei Interpretationsvarianten für Anfragen mit mehreren Relationen

- Nested Loops (geschachtelte Schleifen)
  - Bei mehreren Tupelvariablen: Eine geschachtelte Schleife für jede Variable
- Parallele Zuordnung
  - Alle Kombinationen werden parallel bezüglich der Bedingungen geprüft.
- Relationale Algebra
  - Bilde Kreuzprodukt
  - Wende Selektionsbedingungen auf jedes Resultat-Tupel an



# Interpretation von Anfragen

25

- Gegeben drei Relationen:  $R(A)$ ,  $S(A)$  und  $T(A)$
- Gesucht:  $R \cap (S \cup T)$ 
  - `SELECT R.A`  
`FROM R, S, T`  
`WHERE R.A = S.A OR R.A = T.A;`
  
- Problemfall:  $T$  ist leer, hat also kein Tupel
- Vermeintliches Resultat:  $R \cap S$
- Tatsächliches Resultat: leere Menge
  - Nested Loops
  - Parallele Zuordnung
  - Relationale Algebra

# Mengenoperationen in SQL

26

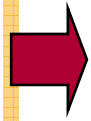
- Vereinigung: UNION
- Schnittmenge: INTERSECT
- Differenz: EXCEPT
- Mengenoperationen nur zwischen (geklammerten) Anfrageergebnissen
  - Schauspieler(Name, Adresse, Geschlecht, Geburtstag)
  - Manager(Name, Adresse, ManagerID, Gehalt)
  
  - (SELECT Name, Adresse FROM Schauspieler  
WHERE Geschlecht = ‚F‘)  
INTERSECT  
(SELECT Name, Adresse FROM Manager  
WHERE Gehalt > 1.000.000);

# Mengenoperationen in SQL

27

- Schauspieler(Name, Adresse, Geschlecht, Geburtstag)
- Manager(Name, Adresse, ManagerID, Gehalt)
- Filme(Titel, Jahr, Länge, inFarbe, StudioName, ProduzentID)
- spielt\_in(FilmTitel, FilmJahr, Name)
  
- (SELECT Name, Adresse FROM Schauspieler)  
    EXCEPT  
    (SELECT Name, Adresse FROM Manager);
- (SELECT Titel, Jahr FROM Filme)  
    UNION  
    (SELECT FilmTitel AS Titel, FilmJahr AS Jahr FROM spielt\_in)

- Einfache Anfragen
- Anfragen über mehrere Relationen
- Geschachtelte Anfragen
- Operationen auf einer Relation
- Datenbearbeitung (DML)
- Schemata (DDL)
- Sichten



# Motivation

29

Eine Anfrage kann Teil einer anderen Anfrage sein

- Beliebig tiefe Schachtelung

Anwendungsmöglichkeiten

- Subanfrage erzeugt einen einzigen Wert, der in der WHERE-Klausel mit einem anderen Wert verglichen werden kann.
- Subanfrage erzeugt eine Relation, die auf verschiedene Weise in WHERE-Klausel verwendet werden kann.
- Subanfrage erzeugt eine Relation, die in der FROM Klausel verwendet werden kann.
  - Wie jede normale Relation

# Skalare Subanfragen

30

Allgemeine Anfrage produzieren Relationen.

- Mit mehreren Attributen
  - Zugriff auf ein bestimmtes Attribut ist möglich
- i.A. mit mehreren Tupeln
- Manchmal (garantiert) nur ein Tupel
  - „Skalare Anfrage“
  - Verwendung wie eine Konstante möglich

# Skalare Subanfragen

31

- Manager(Name, Adresse, ManagerID, Gehalt)
- Filme(Titel, Jahr, Länge, inFarbe, StudioName, ProduzentID)
- Gesucht: Produzent von Star Wars
  - SELECT Name  
FROM Filme, Manager  
WHERE Titel = ‚Star Wars‘ AND Jahr = ‚1977‘  
AND ProduzentID = ManagerID;
- Oder aber
  - SELECT Name  
FROM Manager  
WHERE ManagerID =  
( SELECT ProduzentID  
FROM Filme  
WHERE Titel = ‚Star Wars‘ AND Jahr = ‚1977‘ );
- Wir erwarten genau ein Tupel als Ergebnis der Teilanfrage
  - Falls kein Tupel: Laufzeitfehler
  - Falls mehr als ein Tupel: Laufzeitfehler

# Bedingungen mit Relationen

32

Bestimmte SQL Operatoren auf Relationen erzeugen Boole'sche Werte

- EXISTS R
  - TRUE, falls R nicht leer
- x IN R
  - TRUE falls x gleich einem Wert in R ist (R hat nur ein Attribut)
  - Verallgemeinerung auf Tupel gleich
  - x NOT IN R: TRUE falls x keinem Wert in R gleicht
- x > ALL R
  - TRUE falls x größer als jeder Wert in R ist (R hat nur ein Attribut)
  - Alternativ: <, >, <=, >=, <>, =
  - x <> ALL R: Entspricht x NOT IN R
- x > ANY R
  - TRUE falls x größer als mindestens ein Wert in R ist (R hat nur ein Attribut)
  - Alternativ: <, >, <=, >=, <>, =
  - x = ANY R: Entspricht x IN R
- Negation mit NOT ist immer möglich.



# Bedingungen mit Tupeln

33

## Verallgemeinerung von IN, ALL und ANY auf Tupel

- $t \text{ IN } R$ 
  - TRUE falls  $t$  ein Tupel in  $R$  ist (mehr als ein Attribut möglich)
  - Setzt gleiche Schemata voraus
  - Setzt gleiche Reihenfolge der Attribute voraus
- $t > \text{ALL } R$ 
  - TRUE falls  $t$  größer als jedes Tupel in  $R$  ist
  - Vergleiche in Standardreihenfolge der Attribute
- $t <> \text{ANY } R$ 
  - TRUE falls  $R$  mindestens ein Tupel hat, das ungleich  $t$  ist

# Bedingungen mit Tupeln

34

- ```
SELECT Name
FROM Manager
WHERE ManagerID IN
  ( SELECT ProduzentID
    FROM Filme
    WHERE (Titel, Jahr) IN
      ( SELECT FilmTitel, FilmJahr
        FROM spielt_in
        WHERE SchauspielerName = ‚Harrison Ford‘
      ));
```
- Analyse am besten von innen nach außen
- Namen von Produzenten von Filmen mit Harrison Ford
- Alternative Formulierung
  - ```
SELECT Name
FROM Manager, Filme, spielt_in
WHERE ManagerID = ProduzentID
AND Titel = FilmTitel
AND Jahr = FilmJahr
AND SchauspielerName = ‚Harrison Ford‘;
```

# Korrelierte Subanfragen

35

Bisher: Subanfragen einmalig ausführen und das Ergebnis weiterverwenden

- Korrelierte Subanfragen werden mehrfach ausgeführt, einmal pro Bindung der korrelierten Variable der äußeren Anfrage

- ```
SELECT Titel, Jahr
FROM Filme Alt
WHERE Jahr < ANY
  ( SELECT Jahr
    FROM Filme
    WHERE Titel = Alt.Titel);
```

Scope: Attributnamen gehören i.d.R. zur Tupelvariablen der aktuellen Anfrage. Sonst: Suche von innen nach außen.

- Ausführung der Subanfrage für jedes Tupel in Filme
- Alle mehrfachen Filme mit Ausnahme der jeweils jüngsten Ausgabe sind im Ergebnis.

SQL entfernt idR keine Duplikate!

## Subanfragen in FROM-Klausel

36

Bisher: Nur Subanfragen in WHERE-Klausel

- Anstelle einfacher Relation steht eine geklammerte Subanfrage
- Es muss ein Alias vergeben werden.

- SELECT M.Name

```
FROM Manager M, (SELECT ProduzentID AS ID
```

```
FROM Filme, spielt_in
```

```
WHERE Titel = FilmTitel
```

```
AND Jahr = FilmJahr
```

```
AND Schauspieler = ‚Harrison Ford‘) Produzent
```

```
WHERE M.ManagerID = Produzent.ID;
```

# Joins

37

Man kann Joins auch auf andere Weise ausdrücken.

- Besonders praktisch für Teilanfragen
- Filme CROSS JOIN spielt\_in
  - Kreuzprodukt
  - Doppelte Attributnamen werden mit Präfix der Relation aufgelöst
- Filme JOIN spielt\_in ON Titel = FilmTitel AND Jahr = FilmJahr
  - Theta-Join
  - SELECT Titel, Jahr, Länge, inFarbe, StudioName, ProduzentID, SchauspielerName  
FROM Filme JOIN spielt\_in ON Titel = FilmTitel AND Jahr = FilmJahr;
  - Eliminiert redundante Attribute FilmTitel und FilmJahr
- Schauspieler NATURAL JOIN Manager
  - Natural Join; Eliminiert redundante Attribute

# Outer Joins

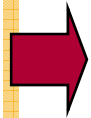
38

- Schauspieler(Name, Adresse, Geschlecht, Geburtstag)
- Manager(Name, Adresse, ManagerID, Gehalt)
- Schauspieler NATURAL FULL OUTER JOIN Manager;
- Schauspieler NATURAL LEFT OUTER JOIN Manager;
- Schauspieler NATURAL RIGHT OUTER JOIN Manager;
  
- Filme FULL OUTER JOIN spielt\_in  
ON Titel = FilmTitel AND Jahr = FilmJahr;
  - Widerspruch in sich? Es sollen schliesslich alle Tupel erhalten bleiben...
- Filme LEFT OUTER JOIN spielt\_in  
ON Titel = FilmTitel AND Jahr = FilmJahr;
- Filme RIGHT OUTER JOIN spielt\_in  
ON Titel = FilmTitel AND Jahr = FilmJahr;

# Überblick

39

- Einfache Anfragen
- Anfragen über mehrere Relationen
- Geschachtelte Anfragen
- Operationen auf einer Relation
- Datenbearbeitung (DML)
- Schemata (DDL)
- Sichten



# Duplikateliminierung

40

Relationale DBMS verwenden idR Multimengensemantik, nicht Mengensemantik.

- Duplikate entstehen durch
  - Einfügen von Duplikaten in Basisrelation
  - Veränderung von Tupeln in Basisrelation
  - Projektion in Anfragen
  - Durch Subanfragen (UNION)
  - Vermehrung von Duplikaten durch Kreuzprodukt
- Duplikateliminierung durch SELECT DISTINCT Attributnamen
  - Kosten sind hoch: Sortierung



# Duplikateliminierung

41

- SELECT ManagerID, Name  
 FROM Manager  
 WHERE ManagerID IN  
 ( SELECT ProduzentID  
 FROM Filme  
 WHERE (Titel, Jahr) IN  
 ( SELECT FilmTitel, FilmJahr  
 FROM spielt\_in  
 WHERE SchauspielerName =  
       'Harrison Ford`'  
 ));
- SELECT ManagerID, Name  
 FROM Manager, Filme, spielt\_in  
 WHERE ManagerID =  
 ProduzentID  
 AND Titel = FilmTitel  
 AND Jahr = FilmJahr  
 AND SchauspielerName =  
       'Harrison Ford`';
- SELECT DISTINCT ManagerID,  
 Name  
 FROM Manager, Filme, spielt\_in  
 WHERE ManagerID =  
 ProduzentID  
 AND Titel = FilmTitel  
 AND Jahr = FilmJahr  
 AND SchauspielerName =  
       'Harrison Ford`';

- Mengenoperationen in SQL entfernen Duplikate
  - UNION, INTERSECT, EXCEPT
    - ◇ wandeln Multimengen in Mengen um und verwenden Mengensemantik
  - Duplikateliminierung verhindern durch ALL
    - ◇ (SELECT Titel, Jahr, FROM Filme)  
UNION ALL  
(SELECT FilmTitel AS Titel, FilmJahr AS Jahr FROM spielt\_in);
    - ◇ Film mit drei Schauspielern erscheint also 4 Mal im Ergebnis
  - R INTERSECT ALL S
  - R EXCEPT ALL S

# Aggregation

43

- Standardaggregationsoperatoren
  - SUM, AVG, MIN, MAX, COUNT
  - Angewendet auf einzelne Attribute in der FROM-Klausel
- Typische weitere Aggregationsoperatoren
  - VAR, STDDEV
- COUNT(\*) zählt Anzahl der Tupel
  - in der Relation, die durch die FROM und WHERE Klauseln definiert wird.
- Kombination mit DISTINCT
  - COUNT(DISTINCT Jahr)
  - SUM(DISTINCT Gehalt)

# Aggregation

44

- `SELECT AVG(Gehalt)  
FROM Manager;`
- `SELECT COUNT(*)  
FROM spielt_in;`
- `SELECT COUNT(Schauspieler)  
FROM spielt_in;`
- `SELECT COUNT(DISTINCT Schauspieler)  
FROM spielt_in;`

# Gruppierung

45

- Gruppierung mittels GROUP BY nach der WHERE-Klausel
- SELECT StudioName, SUM(Länge)  
FROM Filme  
GROUP BY StudioName
- In SELECT-Klausel zwei Sorten von Attributen
  - Gruppierungsattribute
  - Aggregierte Attribute
  - Nicht-aggregierte Werte der SELECT-Klausel müssen in der GROUP BY-Klausel erscheinen.
  - Beide Sorten müssen nicht erscheinen
- SELECT StudioName  
FROM Filme  
GROUP BY StudioName
- SELECT SUM(Länge)  
FROM Filme  
GROUP BY StudioName

# Gruppierung

46

Gruppierung bei mehreren Relationen wird am Schluss durchgeführt.

- SELECT Name, SUM(Länge)  
FROM Manager, Filme  
WHERE ManagerID = ProduzentID  
GROUP BY Name
- Reihenfolge der Ausführung
  - FROM-Klausel
  - WHERE-Klausel
  - GROUP BY-Klausel
  - SELECT-Klausel

# Gruppierung

47

- Einschränkung der Ergebnismenge nach der Gruppierung durch HAVING
  - Einschränkung der Ergebnismenge
    - ◇ SELECT Name, SUM(Länge)  
FROM Manager, Filme  
WHERE ManagerID = ProduzentID  
AND Gehalt > 1000000  
GROUP BY Name
    - ◇ SELECT Name, SUM(Länge)  
FROM Manager, Filme  
WHERE ManagerID = ProduzentID  
GROUP BY Name  
HAVING SUM(Länge) > 1000
    - ◇ SELECT Name, SUM(Länge)  
FROM Manager, Filme  
WHERE ManagerID = ProduzentID  
GROUP BY Name  
HAVING SUM(Länge) > 1000
  - Aggregationen in HAVING Klausel beziehen sich nur auf aktuelle Gruppe.
  - Nur Gruppierungsattribute dürfen unaggregiert in HAVING Klausel erscheinen (wie bei SELECT-Klausel).

# Zusammenfassung

48

## Grundbausteine einer SQL Anfrage

- SELECT
- FROM
- WHERE
- GROUP BY
- HAVING
- ORDER BY
  
- SELECT ... FROM ... sind Pflicht.
  - Ausnahme: z.B. `SELECT 7 + 3`
- HAVING darf nur in Kombination mit GROUP BY erscheinen.



# Überblick

49

- Einfache Anfragen
- Anfragen über mehrere Relationen
- Geschachtelte Anfragen
- Operationen auf einer Relation
- ➔ ■ Datenbearbeitung (DML)
- Schemata (DDL)
- Sichten



# Überblick

50

- Einfügen
  - **INSERT INTO ... VALUES...**
- Löschen
  - **DELETE FROM ... WHERE ...**
- Ändern
  - **UPDATE ... SET ... WHERE ...**

# Einfügen

51

## Grundbaustein

- `INSERT INTO R(A1, ..., An) VALUES (v1, ..., vn);`
  
- Bei fehlenden Attributen
  - Default-Wert (NULL, falls nicht anders angegeben)
  
- Beispiel
  - `INSERT INTO spielt_in(FilmTitel, FilmJahr, Schauspieler) VALUES (`The Maltese Falcon`, 1942, `Sydney Greenstreet`);`
  - Reihenfolge der Werte und Attribute wird beachtet.
  
- Falls alle Attribute gesetzt werden:
  - `INSERT INTO spielt_in VALUES (`The Maltese Falcon`, 1942, `Sydney Greenstreet`);`
  - Reihenfolge entsprechend der Spezifikation des Schemas (CREATE TABLE ...)

# Einfügen per Anfrage

52

Füge in Studios alle Studios der Filme-Relation ein

- Filme(Titel, Jahr, Länge, inFarbe, StudioName, ProduzentID)
- Studios(Name, Adresse, VorsitzenderID)

```
■ INSERT INTO Studios(Name)
  SELECT DISTINCT StudioName
  FROM Filme
  WHERE StudioName NOT IN
    (SELECT Name
     FROM Studios);
```

Adresse und VorsitzenderID  
bleiben NULL.

# Ausführungsreihenfolge beim Einfügen

53

- Wann wird eingefügt?
  - Nach vollständiger Ausführung der SELECT FROM WHERE Anfrage?
  - Sofort?
    - ◇ Schnellere Implementation
  - Was passiert jeweils bei Anfrage 1?
  - Was passiert jeweils bei Anfrage 2?
- ```
INSERT INTO Studios(Name)
SELECT DISTINCT
StudioName
FROM Filme
WHERE StudioName NOT IN
(SELECT Name
FROM Studios);
```
- ```
INSERT INTO Studios(Name)
SELECT StudioName
FROM Filme
WHERE StudioName NOT IN
(SELECT Name
FROM Studios);
```

# Löschen

54

## Grundbaustein

- **DELETE FROM  $R$  WHERE ...**
- Lösche alle Tupel in  $R$ , für die die Bedingung wahr ist.
- **DELETE FROM spielt\_in**  
**WHERE FilmTitel = ,The Maltese Falcon`**  
**AND FilmJahr = 1942**  
**AND Schauspieler = ,Sydney Greenstreet`;**
- Tupel können im Gegensatz zum Einfügen nicht direkt angegeben werden, sondern müssen umschrieben werden.
- **DELETE FROM Manager**  
**WHERE Gehalt < 10000000;**
- Alle Manager-Tupel löschen: **DELETE FROM Manager;**

# Verändern (update)

55

## Grundbaustein

- **UPDATE *R* SET ... WHERE ...**

- **SET Klausel**

  - Wertzuweisungen

  - Komma-separiert

- **UPDATE Manager**

```
SET Name = ,Pres. ` || Name
```

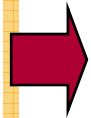
```
WHERE ManagerID IN
```

```
(SELECT PräsidentID FROM Studios);
```

# Überblick

56

- Einfache Anfragen
- Anfragen über mehrere Relationen
- Geschachtelte Anfragen
- Operationen auf einer Relation
- Datenbearbeitung (DML)
- Schemata (DDL)
- Sichten





# Überblick

57

- Datentypen
- Tabellen
- Default-Werte
- Indizes

# Datentypen

58

Jedes Attribut muss einen Datentyp haben.

- CHAR(n)
  - String fester Länge (n)
- VARCHAR(n)
  - String variabler Länge, maximal n Zeichen
- BIT(n) bzw. BIT VARYING(n)
  - Wie CHAR, aber Bits
- BOOLEAN
  - TRUE, FALSE oder UNKNOWN
- INT / INTEGER bzw. SHORTINT
- FLOAT / REAL bzw. DOUBLE PRECISION
- DECIMAL (n,d)
- DATE bzw. TIME

} Sind kompatibel

# Tabellen

59

## Grundbaustein zum Erzeugen

- `CREATE TABLE R ...`
- `CREATE TABLE Schauspieler (  
    Name CHAR(30),  
    Adresse VARCHAR(255),  
    Geschlecht CHAR(1),  
    Geburtstag DATE );`
- Löschen
  - `DROP TABLE Schauspieler;`
- Verändern
  - `ALTER TABLE Schauspieler ADD Telefon CHAR(16);`
    - ◇ Nullwerte entstehen
  - `ALTER TABLE Schauspieler DROP Geburtstag;`

# Default-Werte

60

- `CREATE TABLE Schauspieler (  
    Name CHAR(30),  
    Adresse VARCHAR(255),  
    Geschlecht CHAR(1) DEFAULT ,?`,  
    Geburtstag DATE DEFAULT DATE ,0000-00-00`);`
- `ALTER TABLE Schauspieler ADD Telefon CHAR(16) DEFAULT  
    ,unbekannt`;`

# Ausblick – Constraints und Trigger

61

## Weitere Optionen für Tabellen

- **PRIMARY KEY**
- **UNIQUE**
- **FOREIGN KEY ... REFERENCES ...**
- **NOT NULL**
- **CHECK**
- **CREATE ASSERTION**
- **CREATE TRIGGER**

Siehe auch nächster Foliensatz.

# Indizes

62

- Ein Index auf einem Attribut *A* ist eine Datenstruktur, die es dem DBMS erleichtert, Tupel mit einem bekannten Wert des Attributs zu finden.
  - Nicht SQL-Standard, aber in fast jedem DBMS verfügbar.
  
- Motivation
  - **SELECT \***  
**FROM Filme**  
**WHERE StudioName = ,Disney` AND Jahr = ,1990`;**
  - Variante 1: Alle 10.000 Tupel durchsuchen und WHERE Bedingung prüfen
  - Variante 2: Direkt alle 200 Filme aus 1990 betrachten und auf ,Disney` prüfen.
    - ◇ **CREATE INDEX JahrIndex ON Filme(Jahr);**
  - Variante 3: Direkt alle 10 Filme aus 1990 von ,Disney` holen.
    - ◇ **CREATE INDEX JahrStudioIndex ON Filme(Jahr, StudioName);**

# Indizes

63

- Indizes auf einzelnen Attributen
  - `CREATE INDEX JahrIndex ON Filme(Jahr);`
  
- Indizes auf mehreren Attributen
  - `CREATE INDEX JahrStudioIndex ON Filme(Jahr, StudioName);`
  - Reihenfolge wichtig! Warum?
  
- Löschen
  - `DROP INDEX JahrIndex;`

# Indexwahl

64

- Tradeoff
  - Index beschleunigt Punkt- (und Bereichs-) Anfragen und Joinanfragen erheblich.
  - Index verlangsamt das Einfügen, Löschen und Verändern von Tupeln der Relation.
    - ◇ Index muss jeweils aktualisiert werden.
  - Indizes benötigen Speicherplatz.
  
- Wahl der besten Indizes ist eine der schwierigsten Aufgaben des Datenbankdesigns.
  - Vorhersage der *query workload* und update-Frequenz
  - Wahl der Attribute
    - ◇ Häufiger Vergleich mit Konstanten
    - ◇ Häufiges Joinattribut



# Indexwahl – Vorüberlegungen

65

Relationen sind typischerweise über mehrere Diskblöcke gespeichert.

- Primäre Datenbankkosten sind die Anzahl der Diskblöcke, die in den Hauptspeicher gelesen werden müssen.
- Bei Punktanfragen mit Index müssen statt aller Blöcke nur ein Block gelesen werden.
- Aber Index selbst muss ebenfalls gespeichert und gelesen werden.
- Updates kostet sogar doppelt: Lesen und Schreiben

# Indexwahl – Beispiel

66

- `spielt_in(FilmTitel, FilmJahr, Schauspieler)`
- Drei typische Anfragen
  - `SELECT FilmTitel, FilmJahr FROM spielt_in WHERE Schauspieler = s;`
  - `SELECT Schauspieler FROM spielt_in WHERE FilmTitel = t AND FilmJahr = j;`
  - `INSERT INTO spielt_in VALUES(t, y, s);`
- `spielt_in` ist auf 10 Blöcke verteilt.
- Durchschnittlich hat jeder Film 3 Schauspieler und jeder Schauspieler spielt in 3 Filmen.
  - Echte Daten später.
- 3 Schauspieler/Filme sind auf 3 Blöcke verteilt
- Index ist auf einem Block gespeichert.
- Lesen und Schreiben kostet 1, Update und Insert kosten 2.

# Indexwahl – Beispiel

67

| Anfrage                           | Kein Index    | SchauspielerIndex | Film-Index | Beide Indizes |
|-----------------------------------|---------------|-------------------|------------|---------------|
| Schauspieler = s                  | 10            | 4                 | 10         | 4             |
| FilmTitel = t<br>AND FilmJahr = j | 10            | 10                | 4          | 4             |
| INSERT INTO<br>spielt_in          | 2             | 4                 | 4          | 6             |
| Gesamtkosten                      | $2+8p_1+8p_2$ | $4+6p_2$          | $4+6p_1$   | $6-2p_1-2p_2$ |

- $p_1$ : Anteil Anfrage 1
- $p_2$ : Anteil Anfrage 2
- $1-p_1-p_2$ : Anteil Anfrage 3

# Verteilung in IMDB (Real-World Daten)

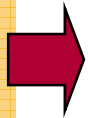
68

- WITH  
m AS (select count(\*) AS ZahlMovies FROM imdb.movie),  
actress AS( select count(\*) AS ZahlActress FROM imdb.actress),  
actor AS (select count(\*) AS ZahlActor FROM imdb.actor),  
actors AS (select (ZahlActress + ZahlActor) AS GesamtActors  
FROM actress, actor)  
SELECT DOUBLE(actors.GesamtActors) /DOUBLE(m.ZahlMovies)  
FROM m, actors
- Schauspieler pro Spielfilm: 8,7
- WITH  
actors AS (SELECT \* FROM imdb.actor UNION  
SELECT \* FROM imdb.actress),  
counts AS (SELECT name, count(movie\_id) AS m  
FROM actors GROUP BY name)  
SELECT AVG(DOUBLE(m)) FROM counts
- Spielfilme pro Schauspieler: 4,2

# Überblick

69

- Einfache Anfragen
- Anfragen über mehrere Relationen
- Geschachtelte Anfragen
- Operationen auf einer Relation
- Datenbearbeitung (DML)
- Schemata (DDL)
- Sichten



# Virtuelle Relationen

70

- Relationen aus **CREATE TABLE** Ausdrücken existieren tatsächlich (materialisiert, physisch) in der Datenbank.
  - Persistenz
  - Updates sind möglich
- Die Daten aus Sichten (*views*) existieren nur virtuell.
  - Sichten entsprechen Anfragen, denen man einen Namen gibt. Sie wirken wie physische Relationen.
  - Updates sind nur manchmal möglich.

# Sichten in SQL

71

- **CREATE VIEW** *Name* **AS** *Anfrage*
- **CREATE VIEW** `ParamountFilme` **AS**  
    **SELECT** `Titel, Jahr`  
    **FROM** `Filme`  
    **WHERE** `StudioName = ,Paramount`;`
  - Auch mehr als eine Relation möglich!
- Semantik
  - Bei jeder Anfrage an die Sicht wird die SQL Anfrage der Sicht ausgeführt.
  - Die ursprüngliche Anfrage verwendet das Ergebnis als Relation.
- Daten der Sicht ändern sich mit der Änderung der zugrundeliegenden Relationen.
- Entfernen der Sicht: **DROP VIEW** `ParamountFilme`
  - Basisdaten bleiben unverändert.

# Anfragen an Sichten

72

- **CREATE VIEW ParamountFilme AS**  
**SELECT Titel, Jahr**  
**FROM Filme**  
**WHERE StudioName = ,Paramount`;**
- **SELECT Titel**  
**FROM ParamountFilme**  
**WHERE Jahr = 1979;**
- Umwandlung der ursprünglichen Anfrage in eine Anfrage an Basisrelationen
  - **SELECT Titel**  
**FROM Filme**  
**WHERE StudioName = ,Paramount` AND Jahr = 1979;**
  - Übersetzung durch DBMS
- Anfrage zugleich an Sichten und Basisrelationen möglich
  - **SELECT DISTINCT Schauspieler**  
**FROM ParamountFilme, spielt\_in**  
**WHERE Titel = FilmTitel AND Jahr = FilmJahr;**



# Anfragen an Sichten

73

- Filme(Titel, Jahr, Länge, inFarbe, StudioName, ProduzentID)
- Manager(Name, Adresse, ManagerID, Gehalt)
- **CREATE VIEW FilmeProduzenten AS**  
    **SELECT Titel, Name**  
    **FROM Filme, Manager**  
    **WHERE ProduzentID = ManagerID;**
- Anfrage
  - **SELECT Name**  
    **FROM FilmeProduzenten**  
    **WHERE Titel = ,Gone with the Wind`**
- Übersetzung
  - **SELECT Name**  
    **FROM Filme, Manager**  
    **WHERE ProduzentID = ManagerID**  
    **AND Titel = ,Gone with the Wind`;**

# Attributumbenennung in Sichten

74

- Umbenennung von Attributen

- `CREATE VIEW FilmeProduzenten(FilmTitel, Produzentename) AS  
SELECT Titel, Name  
FROM Filme, Manager  
WHERE ProduzentID = ManagerID;`

- Oder Sicht einfach nur zur Umbenennung

- `CREATE VIEW Movies(title, year, length, inColor, studio, producerID) AS  
SELECT *  
FROM Filme;`

# Diskussion

75

## Vorteile

- Vereinfachung von Anfragen
- Strukturierung der Datenbank
- Logische Datenunabhängigkeit
  - Sichten stabil bei Änderungen der Datenbankstruktur
- Beschränkung von Zugriffen (Datenschutz)
- Später: Optimierung durch materialisierte Sichten

## Probleme

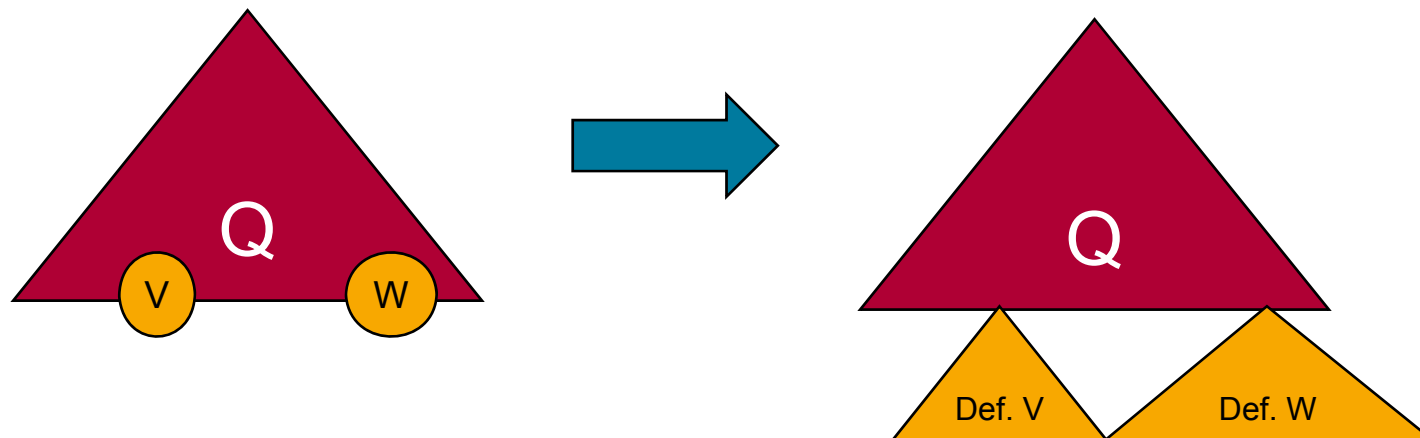
- Automatische Anfragetransformation schwierig
- Durchführung von Änderungen auf Sichten
- Updatepropagierung für materialisierte Sichten

# Anfrageplanung mit Sichten

81

## Baumdarstellung von Anfragen

- Blätter repräsentieren Relationen
  - Basisrelationen
  - Sichten
- Ersetzung der Sichten durch die Sichtdefinition
  - Als Subanfrage



# Anfrageplanung mit Sichten

82

## ■ Sicht

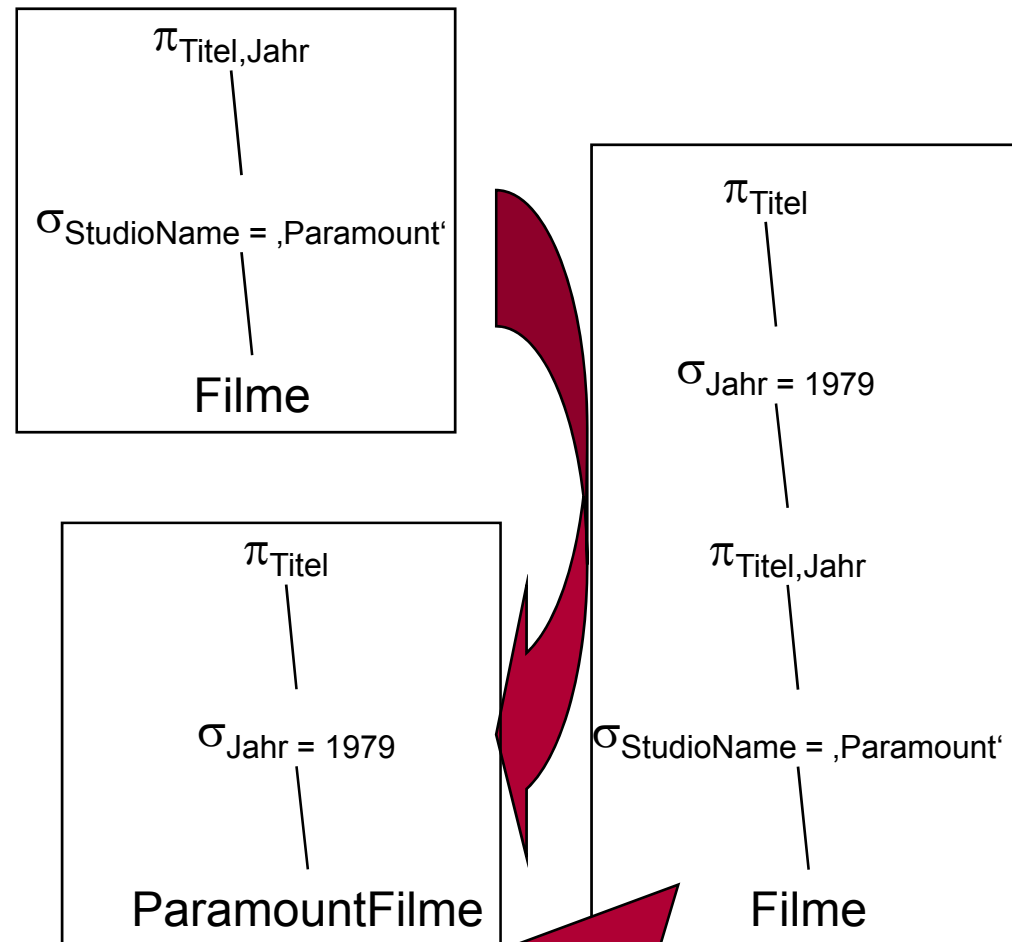
```

□ CREATE VIEW
  ParamountFilme AS
  SELECT Titel, Jahr
  FROM Filme
  WHERE StudioName =
    ,Paramount`;
  
```

## ■ Anfrage

```

□ SELECT Titel
  FROM ParamountFilme
  WHERE Jahr = 1979;
  
```



Weitere Verbesserungen möglich!

# Materialisierte Sichten

83

Viele Anfragen an eine Datenbank wiederholen sich häufig

- Business Reports, Bilanzen, Umsätze
- Bestellungsplanung, Produktionsplanung
- Kennzahlenberechnung

Viele Anfragen sind Variationen mit gemeinsamem Kern

=> Einmaliges Berechnen der Anfrage als Sicht

=> Automatische, transparente Verwendung in folgenden Anfragen

=> Materialisierte Sicht (*materialized view*, MV)

*Drei Folien nach Prof. Ulf Leser, HU Berlin*

# MV – Themen und Probleme

84

## Wahl von Views zur Materialisierung

- MVs kosten: Platz und Aktualisierungsaufwand
- Wahl der optimalen MVs hängt von Workload ab
- Auswahl der „optimalen“ Menge von MVs

## Automatische Aktualisierung von MVs

- Aktualisierung bei Änderungen der Basisrelationen
- U.U. schwierig: Aggregate, Joins, Outer-Joins, ...
- Algorithmen zur inkrementellen Aktualisierung

## Automatische Verwendung von MV

- „*Answering Queries using Views*“
- Umschreiben der Anfrage notwendig
- Schwierigkeit hängt von Komplexität der Anfrage / Views ab
- Algorithmen zur transparenten und kostenoptimalen Verwendung der materialisierten Sichten

# „Answering Queries using Views“

85

## Gegeben

- Eine Anfrage Q
- Eine Menge V von Sichten

## Fragen

- Kann man Q **überhaupt** unter Verwendung von V beantworten?
- Kann man Q **nur** mit V beantworten?
- Kann man Q mit V **vollständig** beantworten?
- Ist es günstig, Sichten aus V zur Beantwortung von Q zu verwenden? Welche?



- Die Anfragesprache SQL
- Der SFW Block
- Subanfragen
  - In FROM und WHERE
  - EXISTS, IN, ALL, ANY
- Mengenoperationen
  - UNION, INTERSECT, EXCEPT
- Joins und Outerjoins
- Nullwerte
- Mengen vs. Multimengen
- Gruppierung und Aggregation
  - MIN, MAX, COUNT
  - GROUP BY, HAVING
- Datenbankveränderungen
  - INSERT, UPDATE, DELETE
- Schemata und Datentypen
  - CREATE TABLE
  - ALTER TABLE
- Indizes
- Sichten
  - Updates und Anfragen
  - Materialisierte Sichten