



5. ANFRAGESPRACHE SQL

- Grundlagen: Anfragekonzept, Befehlsübersicht
- SELECT: mengenorientierte Anfragen deskriptiver Art
 - Selektions- und Projektionsanfragen, Join-Anfragen
 - geschachtelte Anfragen (Sub-Queries)
 - Aggregatfunktionen
 - Gruppenanfragen (GROUP BY, HAVING)
 - Prädikate LIKE, BETWEEN, IN, IS NULL
 - quantifizierte Prädikate (ALL/ANY, EXISTS)
 - Mengen-Operationen: UNION, INTERSECT, EXCEPT
- Änderungsoperationen INSERT, DELETE, UPDATE, MERGE
- Vergleich mit der Relationenalgebra



ENTWICKLUNG VON SQL

- unterschiedliche Entwürfe für relationale Anfragesprachen
 - SEQUEL: Structured English Query Language (System R) -> SQL
 - QUEL (Ingres), . . .
- SQL: vereinheitlichte Sprache für alle DB-Aufgaben
 - einfache Anfragemöglichkeiten für gelegentliche Benutzer
 - mächtige Sprachkonstrukte für besser ausgebildete Benutzer
 - spezielle Sprachkonstrukte für DBA
- Standardisierung von SQL durch ANSI und ISO
 - erster ISO-Standard 1987
 - verschiedene Addenda (1989)
 - 1992: „SQL2“ bzw. SQL-92 (Entry, Intermediate, Full Level)
 - 1999 und später: SQL:1999 („SQL3“), SQL:2003, SQL:2008, SQL:2011, SQL:2016 u.a. mit objektorientierten Erweiterungen (objekt-relationale DBS), XML-Unterstützung, temporalen Anfragen, etc.



ABBILDUNGSORIENTIERTE ANFRAGEN IN SQL

- SQL: strukturierte Sprache, die auf englischen Schlüsselwörtern basiert
 - Zielgruppe umfasst auch Nicht-Programmierer
 - *relational vollständig*: Auswahlvermögen umfasst das der Relationenalgebra
- Grundbaustein

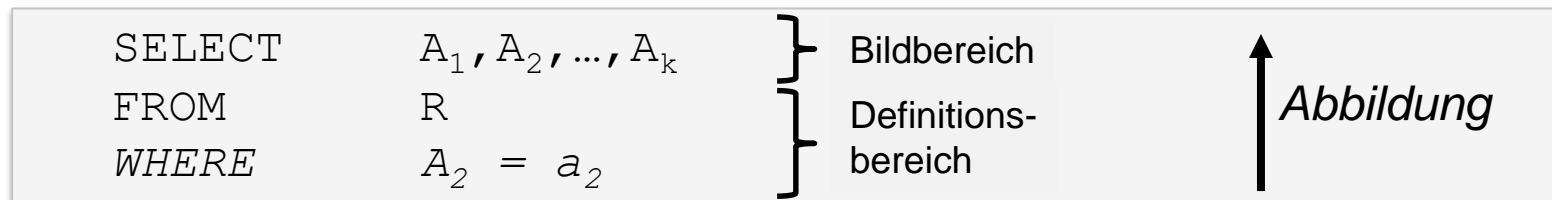


Abbildung: Eingaberelationen (**FROM**) werden unter Auswertung von Bedingungen (**WHERE**) in Attribute einer Ergebnistabelle (**SELECT**) abgebildet

- Allgemeines Format
 - <Spezifikation der Operation>
 - <Liste der referenzierten Tabellen>
 - [WHERE Boolescher Prädikatsausdruck]

ERWEITERUNGEN ZU EINER VOLLSTÄNDIGEN DB-SPRACHE



– Datenmanipulation

- Einfügen, Löschen und Ändern von individuellen Tupeln und von Mengen von Tupeln
- Zuweisung von ganzen Relationen

– Datendefinition

- Definition von Wertebereichen, Attributen und Relationen
- Definition von verschiedenen Sichten auf Relationen

– Datenkontrolle

- Spezifikation von Bedingungen zur Zugriffskontrolle
- Spezifikation von Zusicherungen (assertions) zur semantischen Integritätskontrolle

– Kopplung mit einer Wirtssprache

- deskriptive Auswahl von Mengen von Tupeln
- sukzessive Bereitstellung einzelner Tupeln

	Retrieval	Manipulation	Datendefinition	Datenkontrolle
Stand-Alone DB-Sprache	SQL RA	SQL	SQL	SQL
Eingebettete DB-Sprache	SQL	SQL	SQL	SQL

BEFEHLSÜBERSICHT (AUSWAHL)

Datenmanipulation (DML):

SELECT
INSERT
UPDATE
DELETE

MERGE

Aggregatfunktionen: COUNT, SUM, AVG, MAX, MIN

Datenkontrolle:

Constraint-Definitionen bei CREATE TABLE

CREATE TRIGGER
DROP TRIGGER
GRANT
REVOKE
COMMIT
ROLLBACK

Datendefinition (DDL):

CREATE SCHEMA
CREATE DOMAIN
CREATE TABLE
CREATE VIEW
ALTER TABLE
DROP SCHEMA
DROP DOMAIN
DROP TABLE
DROP VIEW


Eingebettetes SQL:

DECLARE CURSOR
FETCH
OPEN CURSOR
CLOSE CURSOR
SET CONSTRAINTS
SET TRANSACTION
CREATE TEMPORARY TABLE



SQL-TRAINING IN LOTS (<https://lots.uni-leipzig.de>)

- „freies Üben“ auf einer SQL-Datenbank (SELECT-Anweisungen)
 - Realisierung auf Basis von Postgres
- „aktives“ SQL-Tutorial



Leipzig Online-Test-System

UNIVERSITÄT LEIPZIG
Fakultät für Mathematik und Informatik
Institut für Informatik
Abteilung Datenbanken

- Logout
- News
- Training
- SQL-Training**
- XQuery
- Mein Profil
- Impressum

Tutorial

- 1 Einleitung
- 2 Datenbankmodellierung und Relationenmodell
- 3 SQL
- 4 Einfache SQL-Anfragen
 - 4.1 Vorbemerkungen
 - 4.2 Einfache Selektion und Projektion**
 - 4.3 Ausgabebearbeitung
- 5 Verbund-Anfragen
- 6 Unterabfragen
- 7 Aggregatfunktionen
- 8 Partitionierung in Gruppen und Auswahl
- 9 Suchbedingungen
- 10 Mengentheoretische Operationen
- 11 Datendefinition
- 12 Datenmanipulation auf

[Zurück](#) [Weiter](#) [Hoch](#) | [zurück zum SQL-Anfrageformular](#)

4.2 Einfache Selektion und Projektion

Die Projektion aus der [Relationenalgebra](#) stellt sich dar als Auflistung der Attribute nach dem Wort `SELECT`. Folgende SQL-Anfrage repräsentiert eine sehr einfache Projektion.

Beispiel:

BNF: [select-ausdruck](#) diese Anfrage ausführen

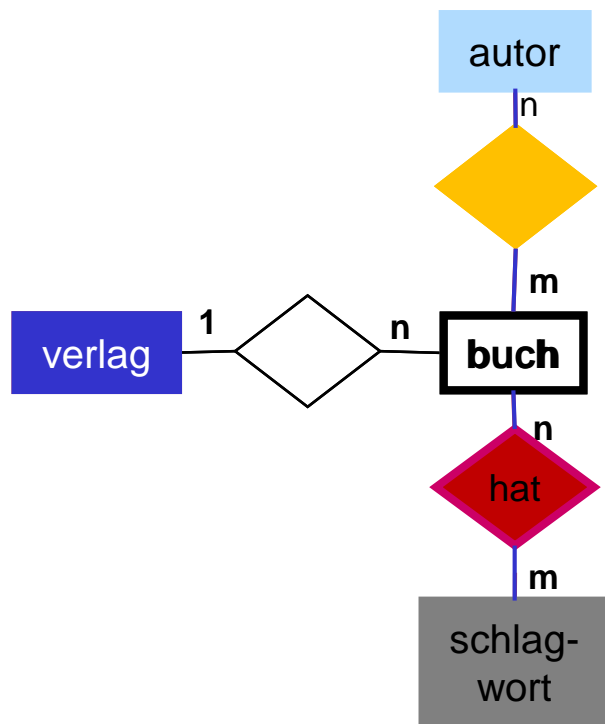
```
SELECT name
FROM verlag
```

Erklärung:

Es werden alle Attributwerte des Attributes "name" aus der Relation "verlag" ausgegeben. In der Reihenfolge der Zeilen ist keine Information codiert. Die Zeilen werden so ausgegeben, wie sie sich (zufällig) in der Relation befinden.

Eine Selektion der [Relationenalgebra](#) ist eigentlich eine Auswahl von Zeilen in der Relation. In der `WHERE`-Klausel werden logische Bedingungen angegeben. Für jede Zeile der Tabelle wird ausgewertet, ob die Bedingung erfüllt ist. Bei positivem Ergebnis wird die Zeile der Auswahlmenge hinzugefügt, andernfalls ignoriert.

TEST-DATENBANK



Verlag	<u>verlagsid</u>	name	ort
--------	------------------	------	-----

Autor	<u>autorid</u>	nachname	initialen	vornamen	zusatz
-------	----------------	----------	-----------	----------	--------

Buch_Aut	<u>buchid</u>	<u>autorid</u>	rolle	rang
----------	---------------	----------------	-------	------

Buch	<u>buchid</u>	titel	isbn	auflage	jahr	preis	waehrung	signatur	verlagsid
------	---------------	-------	------	---------	------	-------	----------	----------	-----------

Buch_SW	<u>buchid</u>	<u>swid</u>
---------	---------------	-------------

Schlagwort	<u>swid</u>	schlagwort
------------	-------------	------------

Buch_Aut.rolle kann sein: Herausgebers (H), Verfasser (V), Übersetzer (U), Mitarbeiter(M)

Buch_Aut.rang: Position des Autors in der Autorenliste (z.B. 1 für Erstautor)

Autor.zusatz: Namenszusatz wie „von“ oder „van“

Buch.signatur entspricht der Signatur in der Ifl-Bibliothek (Stand 1998)

– Mengengerüst (ca. 18.000 Sätze)

- „Buch“ : 4873 Datensätze, „Verlag“ : 414 Datensätze
- „Autor“ : 5045 Datensätze, „Buch_Aut“ : 5860 Datensätze
- „Schlagwort“ : 843 Datensätze, „Buch_SW“ : 789 Datensätze



ANFRAGEMÖGLICHKEITEN IN SQL

select-expression ::=

SELECT [ALL | DISTINCT] select-item+|*

FROM table-ref-commalist

[**WHERE** cond-exp]

[**GROUP BY** column-ref-commalist]

[**HAVING** cond-exp]

[**ORDER BY** order-item+]

select-item ::= derived-column | [range-variable.] *

derived-column ::= scalar-exp [AS column]

order-item ::= column [ASC | DESC]

- **SELECT** –Klausel spezifiziert auszugebende Attribute
 - mit **SELECT** * werden alle Attribute der spezifizierten Relation(en) ausgegeben
- **FROM**-Klausel spezifiziert zu verarbeitende Objekte (Relationen, Sichten)
- **WHERE**-Klausel kann eine Sammlung von Suchprädikaten enthalten, die mit **NOT**, **AND** und **OR** verknüpft sein können
 - dabei sind Vergleichsprädikate der Form $A_i \theta a_i \in W(A_i)$ bzw. $A_i \theta A_j$ möglich mit $\theta \in \{=, <>, <, \leq, >, \geq\}$



EINFACHE SELEKTIONEN UND PROJEKTIONEN

Q1: Welche (Berliner) Verlage gibt es?

```
SELECT  
FROM  
WHERE
```

Q2: Welche Bücher erschienen vor 1980 in einer Neuauflage?
(Ausgabe: Titel, Jahr)

```
SELECT  
FROM  
WHERE
```

SQL Anfrage

```
SELECT Titel, Jahr
FROM Buch
WHERE jahr < 1980 and auflage > 1
```


[akt](#)
[Bibl](#)
[DB](#)
[Zwi](#)

zeige Datensätze 1 - 11 (11 insgesamt)

Zeige:

11

Datensätze, beginnend ab

1

titel	jahr
List Processing	1968
Moderne Logik : Abriß der formalen Logik	1972
Logic and logic design	1973
Wörterbuch der Datenverarbeitung : Englisch-Deutsch	1968
Introduction to switching theory and logical design	1974
Prädikatenkalkül der ersten Stufe	1975
The theory of error-correcting codes	1977
Computer data-base organisation	1977
Grundzüge der theoretischen Logik	1972
Matrizen und ihre technischen Anwendungen	1961
Introduction to switching theory and logical design	1974

Zeige:

11

Datensätze, beginnend ab

1



SORTIERTE AUSGABE

ORDER BY

- Klausel zur Sortierung von Ergebnissätzen
 - aufsteigende (ascending) oder absteigende (descending) Sortierung
 - Voreinstellung: ASCending
 - statt Attributnamen sind in ORDER BY-Klausel auch relative Positionen der Attribute aus Select-Klausel möglich
 - Sortierung nach mehreren Attributen möglich

```
SELECT A1,..., Ak FROM R
ORDER BY Ai ASC, ..., Am DESC
```

Q3: wie **Q2**, jedoch sortiert nach Jahr (absteigend), Titel (aufsteigend)

```
SELECT titel, jahr
WHERE jahr < 1980 and auflage > 1
FROM Buch
ORDER BY
```



DUPLIKATELIMINIERUNG

- SELECT **DISTINCT** erzwingt Duplikateliminierung
 - Default-mäßig werden Duplikate in den Attributwerten der Ausgabe nicht eliminiert (**ALL**)

Q4: Welche Verlagsorte gibt es?

```
SELECT  
    FROM Verlag
```



UMBENENNUNGEN

- Benennung von Ergebnis-Spalten
 - Umbenennung von Attributen (AS optional, DBMS abhängig)
 - Vergabe von Attributnamen für Texte und Ausdrücke

```
SELECT A [AS] B, (num_value/2) [AS] B FROM R
```

- Umbenennung von Tabellen (FROM-Klausel)
 - Einführung sogenannter Alias-Namen bzw. Korrelationsnamen
 - Entscheidend um gleichnamige Attribute aus unterschiedlichen Tabellen zu adressieren → Namenskollision

```
SELECT * FROM R [AS] NEW_R
```



SKALARE FUNKTIONEN: CASE

- Ausgabewert abhängig von verschiedenen Fällen, die durch boolesche Ausdrücke C1,...Cn dargestellt werden
- fehlender ELSE-Zweig: NULL-Wert für sonstige Fälle

```
SELECT CASE
    WHEN C1 THEN a1
    ...
    WHEN C2 THEN a2
    ELSE default END AS case_attr
FROM R
```

Beispiel: `SELECT titel, jahr, CASE`
 `WHEN jahr > 2008 THEN 'Aktuell'`
 `WHEN jahr > 1991 THEN 'Mittel'`
 `ELSE 'Veraltet' END AS aktualitaet`
`FROM Buch`



WEITERE SKALARE FUNKTIONEN (AUSWAHL)

String-Funktionen

- `||` (String-Konkatenation), `CHAR_LENGTH`, `BIT_LENGTH`
- `SUBSTRING` *Bsp.:* `SUBSTRING (Name FROM 1 FOR 20)`
- `POSITION`, `LOWER`, `UPPER`
- `TRIM` löscht White Spaces am Anfange und am Ende eines Strings
 `TRIM(character FROM Attribute)` löscht definiertes Zeichen
 (SQL Server)

Zeit/Datumsfunktionen

- `Now()`, `CURRENT_TIME`, `CURRENT_DATE`, `CURRENT_TIMESTAMP`
- `EXTRACT(field FROM time/date value)`

Typkonversionen

- `CAST(Expression AS datatype)`
- *Bsp.:* `CAST('2022-04-24' AS DATE)`, `CAST(20.7 AS INT)`



EINSATZ VON MENGEN-OPERATOREN

- Vereinigung (UNION), Durchschnitts- (INTERSECT) und Differenzbildung (EXCEPT) von Relationen

```
table-exp {UNION | EXCEPT | INTERSECT }[ALL] table-exp
```

- vor Ausführung werden Duplikate entfernt (außer für ALL)
- für die Operanden werden Vereinigungsverträglichkeit und gleicher/ähnlicher Datentyp gefordert
- **Q5:** Welche Schlagworte wurden nie verwendet ?

```
(SELECT swid FROM Schlagwort) EXCEPT  
(SELECT swid FROM Buch_SW)
```




JOIN-AUSDRÜCKE (1)

```
SELECT *  
FROM R, S  
WHERE R.A  $\theta$  S.A
```

- Angabe der beteiligten Relationen in `FROM`-Klausel
- `WHERE`-Klausel: Join-Bedingung sowie weitere Selektionsbedingungen
- Join-Bedingung erfordert bei gleichnamigen Attributen Hinzunahme der Relationen- oder Alias-Namen (Korrelationsnamen)
- Ohne Join-Bedingung in der `WHERE`-Klausel kartesisches Produkt



JOIN-AUSDRÜCKE (2)

- Join-Spezifikation direkt in FROM-Klausel (explizit)

```
join-table-exp ::= table-ref [NATURAL] [join-type] JOIN table-ref  
                [ON cond-exp | USING (column-commalist) ]  
                | table ref CROSS JOIN table-ref | (join-table-exp)  
table-ref ::= table | (table-exp) | join-table-exp  
join type ::= INNER | { LEFT | RIGHT | FULL } [OUTER] | UNION
```

- Join-Bedingung: ON cond-exp Ergebnisrelation führt Join Attribute nach Vorkommen der Attribute auf
 - Using einmalige Aufführung der Attribute
 - Natural Join zu vergleichende Attribute durch Gleichnamigkeit festgelegt und einmalige Aufführung
- Keine Angabe des join-type und nicht NATURAL → INNER
- table-ref: Relation, SELECT-Query oder bereits gejointe Relationen z.B.
$$\underbrace{(A \bowtie B)}_{\text{join-table-exp}} \bowtie C$$



JOIN-AUSDRÜCKE (3)

- Outer Joins: LEFT JOIN, RIGHT JOIN, FULL JOIN

```
SELECT schlagwort  
FROM Schlagwort LEFT OUTER JOIN Buch_SW USING (swid)  
WHERE buchid IS NULL
```

- kartesisches Produkt:

```
SELECT * FROM A CROSS JOIN B  $\Leftrightarrow$  SELECT * FROM A, B
```



JOIN-ANFRAGEN (1)

Q6: Welche Buchtitel wurden von Berliner Verlagen herausgebracht?

```
SELECT  *  
FROM    Buch B, Verlag V  
WHERE   B.verlagsid = V.verlagsid AND V.ort = 'Berlin'
```

```
SELECT  * FROM Buch NATURAL JOIN Verlag  
        WHERE ort = 'Berlin'
```

```
SELECT  * FROM Buch B JOIN Verlag V ON B.verlagsid = V.verlagsid  
WHERE   ort = 'Berlin'
```

Q7: Welche Bücher sind von Autor „Rahm“ vorhanden?

```
SELECT  titel  
FROM    Buch NATURAL JOIN Buch_Aut NATURAL JOIN Autor  
WHERE   nachname= 'Rahm'
```



JOIN-ANFRAGEN (2)

- hierarchische Beziehung auf einer Relation (PERS)
 - Beispielschema:
 - PERS (PNO int, Name, Salary, ..., MNO int, PRIMARY KEY (PNO), FOREIGN KEY (MNO) REFERENCES PERS)
- **Q8:** Finde die Angestellten, die mehr als ihre (direkten) Manager verdienen (Ausgabe: Name, Gehalt, Name des Managers)

```
SELECT  
FROM PERS  
WHERE
```

- Verwendung von Korrelationsnamen obligatorisch!

<u>PNO</u>	Name	Salary	MNO
34	Mey	32000	37
35	Schultz	42500	37
37	Abel	41000	-

<u>PNO</u>	Name	Salary	MNO
34	Mey	32000	37
35	Schultz	42500	37
37	Abel	41000	-



IN-PRÄDIKATE

```
row-constr [NOT] IN (table-exp) |  
scalar-exp [NOT] IN (scalar-exp-commalist)
```

- ein Prädikat in einer *WHERE*-Klausel kann ein Attribut oder ein Tupel von Attributen auf Zugehörigkeit zu einer Menge testen:
 - *explizite Mengendefinition*: $A_i \text{ IN } (a_1, a_j, a_k)$
 - *implizite Mengendefinition mittels Subquery*: $A_i \text{ IN } (\text{SELECT } \dots)$
- Semantik

$$x \text{ IN } (a, b, \dots, z) \Leftrightarrow x=a \text{ OR } x=b \dots \text{ OR } x=z$$
$$x \text{ NOT IN } (\text{table-exp}) \Leftrightarrow \text{NOT } (x \text{ IN } (\text{table-exp}))$$

Q9: Finde die Autoren mit Nachname Maier, Meier oder Müller

```
SELECT *  
FROM Autor  
WHERE nachname IN ('Maier', 'Meier', 'Müller')
```



GESCHACHTELTE ANFRAGEN (SUB-QUERIES)

- Auswahlbedingungen können sich auf das Ergebnis einer „inneren“ Anfrage (Sub-Query, Unteranfrage) beziehen

```
SELECT *  
FROM R  
WHERE A IN (  
    SELECT B  
    FROM S  
)
```

- innere und äußere Relationen können identisch sein
- eine geschachtelte Abbildung kann beliebig tief sein
- Join-Berechnung mit Sub-Queries
 - teilweise prozedurale Anfrageformulierung
 - weniger elegant als symmetrische Notation
 - schränkt Optimierungsmöglichkeiten des DBS ein



SUB-QUERIES (2)

– einfache Sub-Queries

- 1-malige Auswertung der Sub-Query
- Ergebnismenge der Sub-Query (Zwischenrelation) dient als Eingabe der äußeren Anfrage

```
SELECT *  
FROM R  
WHERE A IN  
      (SELECT B  
       FROM S)
```

– korrelierte Sub-Queries (verzahnt geschachtelte Anfragen)

- Sub-Query bezieht sich auf eine äußere Relation
- Sub-Query-Ausführung erfolgt für jedes Tupel der äußeren Relation
- Verwendung von Korrelationsnamen i.a. erforderlich
- „äußere“ Attribute können in der inneren Query verwendet werden

```
SELECT *  
FROM R r  
WHERE EXISTS  
      (SELECT *  
       FROM S s  
       WHERE r.A = s.B  
       )
```

besser: Join-Berechnung ohne Sub-Queries

- Sub-Queries sind nützlich zur Berechnung komplexer Vergleichswerte in WHERE-Klausel, z.B. durch Anwendung von Aggregatfunktionen in der Sub-Query



SUB-QUERIES -BEISPIEL (2)

- Q10: Welche Buchtitel wurden von Berliner Verlagen veröffentlicht?

einfache Sub-Query

```
SELECT B.titel
FROM buch B
WHERE B.verlagsid IN
      (SELECT V.verlagsid
       FROM verlag V
       WHERE V.ort = 'Berlin')
```

korrelierte Sub-Query

```
SELECT B.titel
FROM buch B
WHERE 'Berlin' IN
      (SELECT V.ort
       FROM verlag V
       WHERE V.verlagsid = B.verlagsid)
```



WEITERGEHENDE VERWENDUNG VON SUB-QUERIES

- 3 Arten von Sub-Queries
 - Table Sub-Queries (mengenwertige Ergebnisse)
 - Row Sub-Queries (Tupel-Ergebnis)
 - skalare Sub-Queries (atomarer Wert; Kardinalität 1, Grad 1)
- Table-Sub-Queries können überall stehen, wo ein Relationenname möglich ist, insbesondere in der FROM-Klausel.
 - Vergabe eines Alias für die Table-Expression

```
SELECT titel
FROM (SELECT * FROM Verlag WHERE Ort='Leipzig')
AS LVerlag NATURAL JOIN Buch
```

- skalare Sub-Queries können auch in SELECT-Klausel stehen

```
SELECT titel, (SELECT name FROM Verlag V
               WHERE V.verlagsid=B.verlagsid) AS Verlag
FROM Buch B
WHERE jahr =2001
```



BENUTZUNG VON AGGREGAT (BUILT-IN)- FUNKTIONEN

```
aggregate-function-ref ::= COUNT (*) | {AVG | MAX | MIN | SUM | COUNT}  
                        ([ALL | DISTINCT] scalar-exp)
```

- Standard-Funktionen: AVG, SUM, COUNT, MIN, MAX
 - Elimination von Duplikaten : DISTINCT
 - keine Elimination : ALL (Defaultwert)
 - Typverträglichkeit erforderlich
- Aggregatfunktionen können nicht direkt in WHERE-Klausel verwendet werden
 - Einsatz von Sub-Queries
- Auswertung
 - Count(select-item) zählt keine NULL-Werte
 - Built-in-Funktion (AVG) wird angewendet auf einstellige Ergebnisliste
 - Verwendung von arithmetischen Ausdrücken ist möglich

```
SELECT *  
FROM R  
WHERE k  $\theta$   
(SELECT aggregate-function-ref FROM S ...)
```



AGGREGATFUNKTIONEN – BEISPIELE (1)

Q11: Wie viele Verlage gibt es?

```
SELECT  
FROM Verlag
```

Q11b: Wie viele Verlage und wieviel davon in Berlin gibt es?

```
SELECT  
  
FROM Verlag
```

Q12: An wie vielen Orten gibt es Verlage?

```
SELECT  
FROM Verlag
```

Q13: Für wie viele Bücher ist der Verlag nicht bekannt?

```
SELECT  
FROM Verlag
```



AGGREGATFUNKTIONEN – BEISPIELE (2)

– Aggregatfunktion in WHERE-Klausel

Q14: Welches Buch (Titel, Jahr) ist am ältesten?

```
SELECT titel, jahr
FROM Buch
WHERE jahr =(SELECT MIN(jahr)
               FROM Buch)
```

Q15: An welchen Orten gibt es mehr als drei Verlage?

```
SELECT  DISTINCT V.ort
FROM    Verlag V
WHERE   3 < (SELECT COUNT(*)
              FROM Verlag V2
              WHERE V2.ort=V.ort)
```



PARTITIONIERUNG EINER RELATION IN GRUPPEN

```
SELECT A1,...,Ak [,aggrgeate_function_ref+] FROM ...  
[WHERE ...]  
[GROUP BY column-ref-commalist]
```

- Gruppenbildung auf Relationen: GROUP-BY-Klausel
 - Tupel mit übereinstimmenden Werten für Gruppierungsattribut(e) bilden je eine Gruppe
 - ausgegeben werden können nur:
Gruppierungsattribute, Konstante, Ergebnis von Aggregatfunktionen (→ 1 Satz pro Gruppe)
 - $\{A1, ... Ak\} \subseteq \text{column} - \text{ref} - \text{commalist}$
 - die Aggregatfunktion wird jeweils auf die Tupel einer Gruppe angewendet



GROUP BY – BEISPIEL (1)

Q16: Liste alle Verlage (verlagsid) mit der Anzahl ihrer Bücher auf

```
SELECT
  FROM Buch NATURAL JOIN Verlag
GROUP BY
```

Ausgabe von Verlagsname?

```
SELECT name
  FROM Buch NATURAL JOIN Verlag
GROUP BY
```



GROUP BY – BEISPIEL

Q17: Liste alle Abteilungen und das Durchschnitts- sowie Spitzengehalt ihrer Angestellten auf.

```
SELECT DNO, AVG(Salary) AS Avg_Salary, Max(Salary) AS Max_Salary
FROM PERS
GROUP BY DNO
```

PERS

<u>PNO</u>	Name	Age	Salary	DNO
235	Schmid	31	32500	K51
123	Schulz	32	43500	K51
237	Bauer	21	21000	K53
234	Meier	23	42000	K53
829	Müller	36	42000	K53
321	Klein	19	27000	K55
406	Abel	47	52000	K55
574	Schmid	28	36000	K55

DNO	Avg_Salary	Max_Salary
K51	38000	43500
K53	35000	42000
K55	38333	52000



AUSWAHL VON GRUPPEN (HAVING-KLAUSEL)

```
SELECT ... FROM ... [WHERE ...]  
[ GROUP BY column-ref-commalist ]  
[ HAVING cond-exp ]
```

- HAVING: Bedingungen nur bezüglich Sätzen einer Gruppe
 - meist Verwendung von Aggregatfunktionen
- Fragen werden in den folgenden Reihenfolge bearbeitet:
 1. Tupel werden ausgewählt durch die WHERE-Klausel.
 2. Gruppen werden gebildet durch die GROUP-BY-Klausel.
 3. Gruppen werden ausgewählt, wenn sie die HAVING-Klausel erfüllen



HAVING-KLAUSEL (2)

- **Q18:** Für welche Abteilungen in Leipzig ist das Durchschnittsalter kleiner als 30?

```
SELECT DNO
  FROM PERS NATURAL JOIN DEPT
 WHERE ort = 'Leipzig'
 GROUP BY DNO
 HAVING AVG(AGE) < 30
```

Q15': An welchen Orten gibt es mehr als drei Verlage?

„Profi-Version“ ohne korrelierte Sub-Query

```
SELECT ort
  FROM Verlag
 GROUP BY ort
 HAVING COUNT(*) > 3
```



SUCHBEDINGUNGEN

- Sammlung von Prädikaten
 - Verknüpfung mit AND, OR, NOT
 - Auswertungsreihenfolge ggf. durch Klammern
- nicht-quantifizierte Prädikate
 - Vergleichsprädikate
 - LIKE-, BETWEEN-, IN-Prädikate
 - Test auf Nullwert
 - UNIQUE-Prädikat: Test auf Eindeutigkeit
 - MATCH-Prädikat: Tupelvergleiche
 - OVERLAPS-Prädikat: Test auf zeitliches Überlappen von DATETIME-Werten
- quantifizierte Prädikate
 - ALL
 - ANY
 - EXISTS



VERGLEICHSPRÄDIKATE

```
comparison-cond ::= row-constructor  $\theta$  row-constructor  
row-constructor ::= scalar-exp | (scalar-exp-commalist) | (table-exp)
```

- skalarer Ausdruck (scalar-exp):
 - Attribut, Konstante bzw. Ausdrücke, die einfachen Wert liefern
- Tabellen-Ausdruck (table-exp) für Row Sub-Query
 - darf hier höchstens 1 Tupel als Ergebnis liefern (Kardinalität 1)
- Vergleiche zwischen Tupel-Konstruktoren (row constructor) mit mehreren Attributen

- $(a_1, a_2, \dots, a_n) = (b_1, b_2, \dots, b_n) \Leftrightarrow a_1 = b_1 \text{ AND } a_2 = b_2 \dots \text{ AND } a_n = b_n$
- $(a_1, a_2, \dots, a_n) < (b_1, b_2, \dots, b_n) \Leftrightarrow (a_1 < b_1) \text{ OR } ((a_1 = b_1) \text{ AND } (a_2 < b_2)) \text{ OR } (\dots)$

```
SELECT ...  
WHERE ('Leipzig', 2000) = (Select Ort, Gründungsjahr FROM Verein  
... )
```

LIKE-PRÄDIKATE

```
char-string-exp [ NOT ] LIKE char-string-exp  
                [ ESCAPE char-string-exp ]
```

- Suche nach Strings, von denen nur Teile bekannt sind (pattern matching)
- LIKE-Prädikat vergleicht einen Datenwert mit einem „Muster“ („Maske“)
- Aufbau einer Maske mit Hilfe zweier spezieller Symbole
 - % bedeutet „0 oder mehr beliebige Zeichen“
 - _ bedeutet „genau ein beliebiges Zeichen“
 - das LIKE-Prädikat ist TRUE, wenn der entsprechende Datenwert der aufgebauten Maske mit zulässigen Substitutionen von Zeichen für '%' und '_' entspricht
 - Suche nach '%' und '_' selbst durch Voranstellen eines Escape-Zeichens möglich.



LIKE-PRÄDIKATE - BEISPIEL

LIKE-Klausel	Wird z.B. erfüllt von
NAME LIKE '%SCHMI%'	'H. SCHMIDT' 'SCHMIED'
ANR LIKE '_7%'	'17' 'K75'
NAME NOT LIKE '%-%'	
STRING LIKE '%__%' ESCAPE '\'	'DBS_1' 'X_Y_Z'



BETWEEN-PRÄDIKATE

```
row-constr [ NOT] BETWEEN row-constr AND row-constr
```

- Überprüfung des Enthaltenseins eines Elements y in einem Intervall $[x, z]$
- Ordnung des Intervalls datentypspezifisch

Semantik

$y \text{ BETWEEN } x \text{ AND } z \quad \Leftrightarrow \quad x \leq y \text{ AND } y \leq z$

$y \text{ NOT BETWEEN } x \text{ AND } z \quad \Leftrightarrow \quad \text{NOT } (y \text{ BETWEEN } x \text{ AND } z)$

- Beispiel

```
SELECT DNO, count(*)  
FROM PERS  
WHERE DNO NOT BETWEEN 'K50' AND 'K54'  
      GROUP BY DNO  
      HAVING AVG (Age) BETWEEN 20 AND 35
```



NULL-WERTE

- pro Attribut kann Zulässigkeit von Nullwerten festgelegt werden
 - unterschiedliche Bedeutungen: Datenwert ist momentan nicht bekannt
 - Attributwert existiert nicht für ein Tupel, z.B. akademischer Titel
- Behandlung von Nullwerten
 - das Ergebnis einer arithmetischen Operation (+, -, *, /) mit einem NULL-Wert ist der NULL-Wert
 - Tupel mit NULL-Werten im Verbundattribut nehmen am Verbund nicht teil
 - Auswertung eines NULL-Wertes in einem Vergleichsprädikat mit irgendeinem Wert ist UNKNOWN (?)
- bei Auswertung von Booleschen Ausdrücken wird 3-wertige Logik eingesetzt
 - das Ergebnis ? bei der Auswertung einer WHERE-Klausel wird wie FALSE behandelt.

NOT	
T	F
F	T
?	?

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?



NULL-WERTE: PROBLEMFÄLLE

- 3-wertige Logik führt zu unerwarteten Ergebnissen
Bsp.: PERS (Age \leq 50) vereinigt mit PERS (Age $>$ 50)
ergibt nicht notwendigerweise Gesamtrelation PERS
- Nullwerte werden bei SUM, AVG, MIN, MAX nicht berücksichtigt, während COUNT(*) alle Tupel (inkl. Null-Tupel, Duplikate) zählt.

SELECT AVG (Salary) FROM PERS → Ergebnis X

SELECT SUM (Salary) FROM PERS → Ergebnis Y

SELECT COUNT (*) FROM PERS → Ergebnis Z

es gilt nicht notwendigerweise, dass $X = Y / Z$

(-> ggf. Verfälschung statistischer Berechnungen)

- spezielles SQL-Prädikat zum Test auf NULL-Werte:

```
row-constr IS [NOT] NULL
```

```
SELECT PNO, PNAME  
FROM PERS  
WHERE SALARY IS NULL
```



QUANTIFIZIERTE PRÄDIKATE

– All-or-Any-Prädikat

```
row-constr  $\theta$  { ALL | ANY | SOME } (table-exp)
```

- θ **ALL**: Prädikat wird zu `true` ausgewertet, wenn der θ -Vergleich für alle Ergebniswerte von `table-exp` `true` ist.
- θ **ANY**/ θ **SOME**: analog, wenn der θ -Vergleich für mindestens einen Ergebniswert `true` ist.

```
SELECT *  
  FROM R  
 WHERE A | (A1, ..., AK)  $\theta$  ALL  
      (SELECT B | (B1, ..., BK)  
      FROM S)
```



EXISTENZTESTS

```
[NOT] EXISTS (table-exp)
```

- Prädikat wird `false`, wenn `table-exp` auf die leere Menge führt, sonst `true`
- im `EXISTS`-Kontext kann `table-exp` mit `(SELECT * ...)` spezifiziert werden (Normalfall)
- Semantik

$$x \in \mathbf{ANY}(\text{SELECT } y \text{ FROM } T \text{ WHERE } p) \Leftrightarrow \text{EXISTS}(\text{SELECT } * \text{ FROM } T \text{ WHERE } (p) \text{ AND } (x \in T.y))$$
$$x \in \mathbf{ALL}(\text{SELECT } y \text{ FROM } T \text{ WHERE } p) \Leftrightarrow \text{NOT EXISTS}(\text{SELECT } * \text{ FROM } T \text{ WHERE } (p) \text{ AND NOT } (x \in T.y))$$



QUANTIFIZIERTE PRÄDIKATE - BEISPIELE

- Q19: Finde die Manager, die mehr verdienen als alle ihre Angestellten

```
SELECT M.PNO
FROM   PERS  M
WHERE  M.Salary > ALL(SELECT P.Salary
                      FROM PERS P
                      WHERE P.MNO=M.PNO)
```

- Q19' (EXISTS)

```
SELECT M.PNO
FROM PERS M
WHERE NOT EXISTS (SELECT *
                  FROM PERS P
                  WHERE P.MNO=M.PNO AND
                        NOT (M.Salary > P.Salary)
                  )
```



QUANTIFIZIERTE PRÄDIKATE – BEISPIELE (2)

Q20: Finde die Schlagworte, die für mindestens ein (... **kein**) Buch vergeben wurden

```
SELECT S.*  
FROM schlagwort S  
WHERE NOT EXISTS (SELECT *  
                  FROM Buch_SW B WHERE B.swid=S.swid)
```



DIVISION

- Keine explizite Realisierung in SQL
- abgeleiteter Operator basierend auf kartesischen Produkt und Differenz
- Sei $R(E1, E2, C1, C2, C3)$ und $S(C1, C2, C3)$ Relationen, wobei $C1, C2, C3$ gemeinsame Attribute und $E1, E2$ exklusiv für R
 - $R \div S = \pi_{E1, E2}(R) - \pi_{E1, E2}[(\pi_{E1, E2}(R) \times S) - R]$

E1	E2	C1	C2	C3
1	1	x	y	z
1	1	a	b	c
2	3	x	y	z

C1	C2	C3
x	y	z
a	b	c

```
SELECT R1.E1, R1.E2
FROM R R1
WHERE (E1, E2) NOT IN (
    SELECT E1, E2
    FROM ((SELECT R.E1, R.E2 FROM R) R2 CROSS JOIN S) cp WHERE
    (cp.E1, cp.E2, cp.C1, cp.C2, cp.C3) NOT IN (SELECT * FROM R)
)
```

DIVISION



– Korrelierte Subquery

```
SELECT DISTINCT R1.E1, R1.E2
FROM R R1
WHERE NOT EXISTS
```

```
(SELECT *
FROM S WHERE (S.C1, S.C2, S.C3) NOT IN
(SELECT R2.C1, R.C2, R.C3
FROM R R2
WHERE R1.E1=R2.E1 AND R1.E2=R2.E2)
)
```

für ein Ergebnistupel $e=(E1,E2)$ darf so ein gemeinsames Tupel aus S nicht existieren

Tupel aus S, die nicht in R sind bzgl. der gemeinsamen Tupel (C1,C2, C3) für ein bestimmtes exklusiv Tupel $e=(E1,E2)$

E1	E2	C1	C2	C3
1	1	x	y	z
1	1	a	b	c
2	3	x	y	z

C1	C2	C3
x	y	z
a	b	c

– GROUP BY und HAVING COUNT

```
SELECT DISTINCT R.E1, R.E2
FROM R, S
WHERE R.C1 = S.C1 AND R.C2 = S.C2 AND R.C3 = S.C3
GROUP BY R.E1, R.E2
HAVING COUNT(*) = (SELECT COUNT(DISTINCT S.C1, S.C2, S.C3) FROM S)
```

Tupel in R die Verbundpartner in S haben



DIVISION: BEISPIEL

- Q21: Welche Bücher umfassen alle Schlagworte, die das Buch 3545 enthält?
- $\pi_{buchid,swid}(Buch_{sw}) \div \pi_{swid}(\sigma_{buchid=3545}(Buch_{sw}))$
- Abwandlung mithilfe von NOT IN muss nicht Vereinigungsverträglichkeit beachtet werden
- $\pi_{buchid}(Buch) - \pi_{buchid}[(\pi_{buchid}(Buch) \times$

```
SELECT DISTINCT b.titel
FROM Buch b WHERE b.buchid NOT IN (
    SELECT b_cross.buchid
    FROM (SELECT buchid FROM BUCH) b_cross CROSS JOIN
    (SELECT swid FROM buch_sw WHERE buchid = 3545) buch_sel_cross
    WHERE (b_cross.buchid, buch_sel_cross.swid) NOT IN
        (SELECT buchid, swid FROM buch_sw)
)
```




DIVISION: BEISPIEL

– Q21': korreliert

```
SELECT DISTINCT b.titel
FROM Buch b WHERE NOT EXISTS (
    SELECT *
    FROM buch_sw
    WHERE buchid = 3545 AND swid NOT IN
    (SELECT swid FROM buch_sw WHERE b.buchid = buch_sw.buchid)
)
```

– Q21'': GROUP BY und COUNT

```
SELECT DISTINCT b.titel
FROM Buch b NATURAL JOIN buch_sw NATURAL JOIN
(SELECT swid FROM buch_sw WHERE buchid = 3545) b_sel
GROUP BY buch_id HAVING(COUNT(swid))=
    (SELECT COUNT(DISTINCT SWID)
     FROM buch_sw WHERE buchid = 3545)
```



EINFÜGEN VON TUPELN (INSERT)

```
INSERT INTO table[(column-commalist)]  
    {VALUES row-constr-commalist|  
    table-exp|  
    DEFAULT VALUES}
```

- satzweises Einfügen (direkte Angabe der Attributwerte)
- Bsp.: Füge Autor Garfield mit Autorid 4711 ein

```
INSERT INTO Autor(autorid, vorname)  
    VALUES (4711, 'Garfield')
```

- alle nicht angesprochenen Attribute erhalten Nullwerte
- falls alle Werte in der richtigen Reihenfolge versorgt werden, kann Attributliste entfallen (NICHT zu empfehlen)
- Integritätsbedingungen müssen erfüllt werden



INSERT (2)

- mengenorientiertes Einfügen: einzufügende Tupeln werden mit Hilfe einer SELECT-Anweisung ausgewählt
- *Bsp.: Füge die Verlage aus Leipzig in Relation TEMP ein*

```
INSERT INTO TEMP
      SELECT *
      FROM Verlag
      WHERE ort='Leipzig'
```

- (leere) Relation TEMP mit kompatiblen Attributen sei vorhanden
 - Reihenfolge der Wertebereiche der selektierten Attributwerte muss kompatibel mit Wertebereichen der Attribute der Relation sein
- spezifizierte Tupelmenge wird ausgewählt und in Zielrelation kopiert
- die eingefügten Tupel sind unabhängig von denen, von denen sie abgeleitet/kopiert wurden
- Einfügen von Werten für bestimmter Attribute mittels table-expression erfordert Spezifikation der Attribute in die einzufügende Relation

ÄNDERN VON TUPELN (UPDATE)

```
searched-update ::=      UPDATE table
                        SET update-assignment-commalist
                        [WHERE cond-exp]
update-assignment ::= column = {scalar-exp|DEFAULT|NULL}
```

Gib den Mitarbeitern der Abteilung „K56“ eine Gehaltserhöhung von 2%

```
UPDATE PERS  SET Salary=Salary * 1.02
WHERE DNO='K56'
```

Stelle die DM-Preise für die nach 1997 erschienenen Bücher auf Euro um (Verhältnis 1:1,95)

```
UPDATE Buch
SET preis=preis/1.95, SET waehrung='Euro'
WHERE jahr > 1997 AND waehrung='DM' AND
      preis IS NOT NULL
```

LÖSCHEN VON TUPELN (DELETE)



```
searched-delete ::= DELETE FROM table [WHERE cond-exp]
```

- Aufbau der WHERE-Klausel entspricht dem der SELECT-Anweisung.
 - Lösche den Autor mit der Autorid 4711

```
DELETE FROM Autor  
WHERE autorid=4711
```

- Lösche alle Schlagworte, die nicht verwendet werden.

```
DELETE FROM Schlagwort  
WHERE swid NOT IN (SELECT swid FROM Buch_SW)
```



MERGE (SEIT SQL:2003)

- Kombination von Insert/Update/Delete beim Mischen von Tabellen

```
MERGE INTO table1 USING table2 | table-exp ON cond-expr
WHEN MATCHED THEN
    UPDATE SET update-assignment-commalist [WHERE cond-expr]
    [ DELETE WHERE cond-expr ]
WHEN NOT MATCHED THEN
    INSERT (column-commalist) VALUES row-constr-commalist
    [WHERE cond-expr]
```

- Unterscheidung zwischen bereits vorkommenden Werten und neuen
- Ändern oder Löschen der existierenden Werte table1 mit Werten aus table2 (Einschränkung der zu ändernden/löschenden Tupel durch WHERE-Bedingung)
- Bedingtes (WHERE cond-expr) Einfügen der neuen Tupel aus table2



MERGE - BEISPIEL

Course	Task	Stud	Credits
C1	1	S1	4
C1	2	S1	5
C1	1	S2	3
C1	2	S2	5

Course	Task	Stud	Credits
C1	1	S1	5
C1	2	S1	5
C1	1	S2	3
C1	2	S2	6
C1	3	S2	3

```
MERGE INTO CREDITS AS C USING NEW_CREDITS as N
  ON C.Course = N.Course AND C.Task = N.Task AND
  C.Stud = N.Stud
WHEN MATCHED THEN UPDATE SET Credits = N.Credits
WHEN NOT MATCHED THEN INSERT VALUES(N.Course, N.Task,
N.stud, N.Credits)
```



RELATIONENALGEBRA VS. SQL (RETRIEVAL)

Relationenalgebra	SQL
$\pi_{A1, A2, \dots, Ak} (R)$	SELECT DISTINCT A1, A2, ... Ak FROM R
$\sigma_P (R)$	SELECT * FROM R WHERE P
$R \times S$	SELECT * FROM R,S bzw. FROM R CROSS JOIN S
$R \bowtie_{R.A \theta S.B} S$	SELECT * FROM R,S WHERE R.A θ S.B bzw. SELECT * FROM R JOIN S ON (R.A θ S.B)
$R \bowtie S$	SELECT * FROM R NATURAL JOIN S
$R \Join S$	SELECT * FROM R LEFT JOIN S
$R \Join S$	SELECT * FROM R FULL JOIN S
$R \ltimes S$	SELECT R.* FROM R NATURAL JOIN S
$R \cup S$	SELECT * FROM R UNION SELECT * FROM S
$R \cap S$	SELECT * FROM R INTERSECT SELECT * FROM S
$R - S$	SELECT * FROM R EXCEPT SELECT * FROM S
$R \div S$	SELECT ... FROM R WHERE NOT EXISTS (SELECT * ...)



ZUSAMMENFASSUNG

- SQL wesentlich mächtiger als Relationenalgebra
- Hauptanweisung: SELECT
 - Projektion, Selektion, Joins
 - Aggregatfunktionen
 - Gruppenbildung (Partitionierungen)
 - quantifizierte Anfragen
 - Unteranfragen (einfache und korrelierte Sub-Queries)
 - allgemeine Mengenoperationen UNION, INTERSECT, EXCEPT
- Datenänderungen: INSERT, UPDATE, DELETE
- hohe Sprachredundanz
- SQL-Implementierungen weichen teilweise erheblich von Standard ab (Beschränkungen / Erweiterungen)