Hochschule für Technik und Wirtschaft Berlin

University of Applied Sciences

Datenbanken

Vorlesungsskript für das 3. Semester

Studiengang Internationale Medieninformatik

3. SQL Aggregatfunktionen

Dozent: M. Sc. Burak Boyaci

Version: 28.10.2025

Wintersemester 25/26

4

Daten analysieren und zusammenfassen (GROUP BY)

Kapitelübersicht

4.1 Aggregatfunktionen

In Abschnitt 2.7 haben wir bereits einige Funktionen von SQL kennengelernt. Sie rechnen mit Spaltenwerten der einzelnen Datensätze. Wollen wir jedoch mit Werten *verschiedener* Datensätze rechnen, kommen wir damit nicht weiter. Wir benötigen dafür sogenannte Aggregatfunktionen.

Eine *Aggregatfunktion*, auch *Gruppenfunktion* genannt, ist eine Funktion in SQL, die als Argument einen Spaltennamen erhält und die Spaltenwerte mehrerer Datensätze zu einem einzigen Wert zusammenführt. Tabelle 4.1 gibt einen Überblick über gängige Aggregatfunktionen. Sie umfassen also Funktionen zum Zählen oder Summieren von Einträgen einer Spalte oder

COUNT(s), **COUNT**(*)	die Anzahl der ausgewählten Datensätze
COUNT(**DISTINCT** s)	Anzahl unterschiedlicher Werte für sder ausgewählten Datensätze
MAX(x)	der größte Eintrag in Spalte x
MIN(x)	der kleinste Eintrag in Spalte x
SUM(x)	die Summe aller Einträge von x
AVG(x)	Der Mittelwert aller Einträge von x
STDDEV(x)	die empirische Standardabweichung aller Einträge von x (D.h. die Einträge werden als Stichprobe einer Gesamtmenge mit unbekanntem Mittelwert betrachtet). Beachte: In MS Access lautet die Funktion **STDEV**!
VARIANCE(x)	die empirische Varianz aller Einträge von x

Tabelle 4.1: Gängige Aggregatfunktionen in SQL. Hier bezeichnet s einen beliebigen Spaltennamen, x den Namen einer Spalte eines numerischen Datentyps

zur Bestimmung des größten oder kleinsten Eintrags. Ebenso kann man statistische Größen wie Mittelwert oder Standardabweichung bestimmen.[1]

Beispiel 4.1. *(Zählen von Datensätzen)* Um die Anzahl von Datensätzen zu bestimmen, können wir die Aggregatfunktion **COUNT** verwenden. Mit

```
SELECT COUNT (*) FROM alben WHERE reihe= 'Asterix' ;
```

ermitteln wir die Anzahl aller Datensätze der Reihe Asterix. Die Ergebnismenge lautet:

COUNT (*)
4

Statt dem Sternchen * hätten wir in diesem Fall auch ein beliebiges Attribut der Tabelle alben einsetzen können, das Resultat bliebe dasselbe. Wichtig ist das Argument der COUNT-Funktion erst, wenn wir Fragen wie die folgende beantworten möchten: Wieviel *verschiedene* Reihen ungleich Asterix haben wir in unserem Albenbestand? Mit dem Argument reihe liefert die Abfrage

```
SELECT COUNT (reihe) FROM alben WHERE reihe <>'Asterix' ;
```

dann:

COUNT (reihe)
3

Das ist jedoch die Anzahl aller *Alben*, die nicht Teil der Asterix Reihe sind. Um die Anzahl der verschiedenen *Reihen* zu ermitteln, benötigen wir den zusätzlichen Parameter **DISTINCT**:

```
SELECT COUNT (DISTINCT reihe) FROM alben WHERE reihe <> 'Asterix' ;
```

ergibt

COUNT (DISTINCT reihe)
2

So erhalten wir die korrekte Antwort 2.

□

Beispiel 4.2. *(Statistische Auswertungen)* Um sich statistische Größen eines Datenbestandes berechnen zu lassen, können wir einige der Aggregatfunktionen verwenden, zum Beispiel für den Durchschnitt und die Standardabweichung der Albenpreise sowie deren Minimum und Maximum:

```
SELECT ROUND (AVG(preis), 3) AS mittelwert,
       ROUND(STDDEV(preis), 3) AS standardabweichung,
       MIN(preis),
       MAX(preis)
FROM alben ;
```

Hierbei wird für die berechneten Werte die Rundungsfunktion verwendet, um sie übersichtlich auszugeben:

mittelwert	standardabweichung	MIN(preis)	MAX(preis)
4.743	2.999	1.20	8.80

Bemerkung 4.3. Aggregatfunktionen sind deutlich zu unterscheiden von den in Abschnitt 2.7 aufgeführten Funktionen, die auf einzelnen Datensätzen wirken, aber nicht auf Gruppen von Datensätzen. Die Abfrage

```
SELECT SQRT (preis) FROM alben ;
```

zeigt so viele Werte an, wie es Datensätze gibt, während

```
SELECT AVG (preis) FROM alben ;
```

nur *einen* Wert anzeigt.

Bemerkung 4.4. NULL-Werte werden von Aggregatfunktionen nicht berücksichtigt, also einfach ignoriert. Für den Fall, dass NULL-Werte stattdessen durch einen Standardwert ersetzt werden sollen, können wir die Funktion **if null** (bzw. **Nz** bei MS Access oder **NVL** bei Oracle) verwenden.

Beispiel 4.5. *(Unterabfragen)* Um sich alle Titel anzuzeigen, deren Preis kleiner ist als der Durchschnittspreis aller Alben, muss man zunächst durch die Aggregatfunktion **AVG**(preis)den Durchschnitt berechnen und diesen als einzigen Wert der Ergebnismenge in einer WHERE-Klausel mit den Preisen der Alben filtern:

```
SELECT titel FROM alben WHERE preis < (SELECT AVG (preis) FROM alben) ;
```

Die in Klammern geschriebene Abfrage ist eine *Unterabfrage (Subquery)*, oft auch *verschachtelte Abfrage* genannt. Sie wird zuerst ausgeführt.

4.2 Gruppieren nach Spalten mit GROUP BY

Mit **GROUP BY** wird in einem SELECT eine Gruppenbildung durchgeführt. Dabei werden jeweils Teilmengen von Ergebniszeilen zu einer Gruppe zusammengefasst, wenn sie in der oder den ausgewählten Spalten – der oder den *Gruppenspalten* – die gleichen Werte besitzen. Gruppenbildungen werden meist vorgenommen, um Aggregatfunktionen wie Summe, Maximum o.ä. für die ganze Gruppe zu berechnen und dann für jede Gruppe separat in einer Zeile auszugeben.

Die Syntax lautet für n Gruppenspalten (n S 1):

```
SELECT   spalten aus gruppenspalten, [aggegatfunktionen] FROM tabelle
GROUP BY     gruppenspalte_1, ..., gruppenspalte_n ;
```

Hierbei wird nach den n Spalten gruppenspalte_1, . . . , gruppenspalte_ngruppiert und je Gruppe ggf. die Aggregatfunktion(en) je Gruppe berechnet. Dabei dürfen neben Aggregatfunktionen nur Spalten angezeigt werden, die auch Gruppenspalten sind, also nach **GROUP BY** stehen.

Beispiel 4.6. Betrachten wir als typisches Beispiel für eine Gruppenbildung die Frage nach dem Durchschnittspreis der einzelnen Reihen unserer Comic-Alben aus Beispiel 3.4 auf Seite 12 des vorherigen Skripts Datenbanken_2_SQL_Grundlagen.

```
SELECT reihe, AVG(preis) FROM alben GROUP BY reihe;
```

reihe	AVG(preis)
Gespenster Geschichten	1.2
Asterix	3.6
Franka	8.8

Die Gruppenspalte ist hier reihe und sollte auch als einzige Spalte der Tabelle im SELECT angezeigt werden. Ansonsten brechen die meisten Datenbanksysteme mit einer Fehlermeldung ab.

```
SELECT reihe,
        ROUND(AVG(preis), 3),
        ROUND(STDDEV(preis), 2) AS s,
        MIN(preis),
        MAX(preis)
FROM alben
GROUP BY reihe ;
```

reihe	AVG(preis)	S	MIN(preis)	MAX(preis)
Gespenster Geschichten	1.2	null	1.20	1.20
Asterix	3.6	1.05	2.80	5.00
Franka	8.8	0.0	8.80	8.80

Regel 1. *Wenn in der Spaltenauswahl einer **SELECT**-Anweisung Spalten* und *Aggregatfunktionen auftreten, müssen alle Spalten in der **GROUP BY**-Komponente aufgelistet werden.*

4.3 Gruppierungen filtern mit HAVING

Das Ergebnis einer **GROUP BY** Anweisung ist eine Tabelle. **HAVING** entspricht einer Filterung darauf.

```
... GROUP BY... HAVING <bedingung> ...;
```

Nachdem mit dem Ausdruck **GROUP BY** die Ergebnisdatensätze einer SELECT-Anweisung gruppiert wurden, können mit **HAVING** in diesen Gruppen die Datensätze gefiltert werden, die der Bedingungen genügen. Auch hier ist bei der Auswertung bei NULL-Werten die dreiwertige Logik zu beachten: Sowohl falsche als auch unbekannte Wahrheitswerte fallen heraus.

Beispiel 4.7. Eine typische Fragestellung, die mit der **HAVING**-Klausel gelöst werden kann, ist für unser Comic-Datenbank die folgende: Welche Reihen haben als Durchschnittspreis ihrer Bände einen Wert kleiner 5 €?

```
SELECT reihe, AVG(preis) FROM alben GROUP BY reihe HAVING AVG (preis) <= 5;
```

reihe	AVG(preis)
Gespenster Geschichten	1.2
Asterix	3.6

Die HAVING-Klausel unterscheidet sich von der WHERE-Klausel, dass die erstere auf die einzelnen Gruppen angewendet wird, letztere dagegen auf jeden einzelnen Datensatz. Auch ist die Abarbeitungsreihenfolge anders: Die WHERE-Klausel wird zunächst die jeden einzelnen Datensatz geprüft, bevor weitere Anweisungen der Abfrage ausgeführt werden. Bei HAVING werden *alle* Datensätze gruppiert, bevor für sie die Bedingung der HAVING-Klausel ausgewertet wird. Für große Datenmengen kann das zu längeren Laufzeiten führen. Auf der anderen Seite können in einer HAVING-Klausel aber auch Werte von Aggregatfunktionen berechnet werden. Das macht in einer WHERE-Bedingung keinen Sinn. Dagegen kann in einer HAVING-Klausel kein Alias verwendet werden.

Regel 2. *Die HAVING-Klausel wird als letzter Teil des SELECT-Befehls ausgeführt. Einschränkungen sollten daher möglichst vorher als Teil von WHERE-Bedingungen gefiltert werden. Nur wenn, wie bei Aggregatfunktionen, diese Einschränkungen erst am Schluss geprüft werden können, muss HAVING benutzt werden.*

Beispiel 4.8. Mit der folgenden Abfrage können wir uns die Reihen ungleich Asterix mit deren Anzahl Alben anzeigen lassen.

```
SELECT reihe, COUNT(*) FROM alben
GROUP BY reihe
HAVING reihe <> 'Asterix' ;
```

Sie liefert uns das Ergebnis:

reihe	COUNT(*)
Gespenster Geschichten	1
Franka	2

Die Abarbeitungsreihenfolge ist hier allerdings ungünstig, zuerst werden alle Datensätze gruppiert und erst danach diejenigen ungleich Asterix gefiltert. Dasselbe Resultat erhalten wir durch die folgende Abfrage:

```
SELECT reihe, COUNT(*) FROM alben
WHERE reihe <> 'Asterix'
GROUP BY reihe ;
```

Hier wird zuerst gefiltert und dann erst gruppiert. Das ist effizienter. Hätten wir eine Datenbank mit sehr vielen Datensätzen, so wäre die erste Abfrage merklich langsamer.

Beispiel 4.9. Wieviel Bände haben die Reihen in unserer Sammlung, die nur aus einem einzigen Wort bestehen? Zunächst ist hier die Bedingung („aus nur einem einzigen Wort“) zu überlegen. Aber mit dem Ansatz, dass eine Reihe aus mindestens zwei Wörtern ja mindestens ein Leerzeichen enthalten muss, kann mit Hilfe des **LIKE**-Operators und der Wildcard die Filterbedingung bilden:

```
reihe NOT LIKE '%%' ;
```

Da wir nach Reihen gruppieren möchten, können wir diese Bedingung als **HAVING**-Klausel verwenden:

```
SELECT reihe, COUNT(band) FROM alben GROUP BY reihe HAVING reihe NOT LIKE '%%' ;
```

Genauso gut wäre aber auch möglich, sie entsprechend in eine **WHERE**-Klausel zu packen: oder mit **WHERE**:

```
SELECT reihe, COUNT(band) FROM alben WHERE reihe NOT LIKE '%%' GROUP BY reihe ;
```

Beachten Sie auch hier die unterschiedliche Position von **HAVING** und **WHERE**.

Man kann bei einem **GROUP BY** grundsätzlich immer da **HAVING** verwenden, wo auch ein **WHERE** möglich ist. Aber nicht umgekehrt, wie wir gleich mit Regel 4 sehen werden.

4.4 WHERE oder HAVING? Bedingungen mit Aggregatfunktionen

Eine typische Fragestellung beinhaltet oft das Problem, Datensätze unter einer Bedingung zu filtern, die eine Aggregatfunktion enthält. Beispielsweise sind das Fragen wie: Welche Artikel

wurden mehr als eine Million Mal verkauft? Welche Prüflinge haben einen besseren Notendurchschnitt als eine 3? Welche Aktien sind sehr riskant, haben also Kurse mit einer Standardabweichung größer als die Hälfte ihres Mittelwerts?
Zur Lösung solcher Fragestellungen müssen wir auf jeden Fall die Datensätze mit **GROUPBY** geeignet zusammenfassen, also beispielsweise **GROUP BY** artikel, pruefling oder aktie. Allerdings wollen wir ja gar nicht alle Datensätze sehen, d.h. wir müssen filtern. Erster Ansatz: Verwenden wir doch die **WHERE**-Klausel! Leider wird das nicht funktionieren. Denn die Filterbedingung hängt hier jeweils von (mindestens) einer Aggregatfunktion ab, nämlich **COUNT**(artikel), **AVG**(note) oder **STDDEV**(kurs). In einer **WHERE**-Klausel dürfen aber nur Spaltennamen auftreten. Wir *müssen* daher die **HAVING**-Klausel verwenden:

```
SELECT artikel FROM verkäufe
GROUP BY artikel
HAVING COUNT (artikel) > 1e6
```

oder

```
SELECT pruefling FROM pruefungen
GROUP BY pruefling
HAVING AVG (note) > 3
```

oder

```
SELECT aktie FROM portfolio
GROUP BY aktie
HAVING STDDEV (kurs) >= 0.5 * AVG(kurs)
```

Beispiel 4.10. Um herauszufinden, welche Reihen unserer Comics mehr als 3 Bände enthalten, können wir die folgende Anweisung verwenden:

```
SELECT reihe
FROM alben
GROUP BY reihe
HAVING count (band) > 3
```

Zur Überprüfung vergleichen wir die folgenden Anweisungen ohne und mit Filterbedingung:

```
SELECT reihe, COUNT(band) FROM
alben GROUP BY reihe ;
```

reihe	COUNT(band)
Gespenster Geschichten	1
Asterix	4
Franka	2

```
SELECT reihe, COUNT(band) FROM alben
GROUP BY reihe HAVING count (band) > 3 ;
```

reihe	COUNT(band)
Asterix	4

Wir erkennen daraus, dass die Aggregatfunktion nicht in der Spaltenauswahl auftreten muss, wenn wir sie nicht angezeigt haben wollen.

Halten wir zum Schluss die zusammenfassend die folgende Regel fest, die die Grenze der Anwendbarkeit von **WHERE** zu **HAVING** beschreibt.

Regel 3. *Filterbedingungen, die Aggregatfunktionen enthalten, müssen in die **HAVING**-Klausel. Eine **WHERE**-Klausel darf nur Spalten enthalten.*

4.5 Der OVER-PARTITION-Ausdruck

Der **OVER**-**PARTITION**-Ausdruck ist eine Möglichkeit, in einer Abfrage Aggregatfunktionen auf einzelne Spalten anzuwenden, ohne dabei die Datensätze in der Ausgabe zusammenzufassen. Sie funktioniert praktisch wie ein „spaltenweises“ GROUP-BY, das die berechneten Ergebnisse allerdings nicht gruppiert. Wir sprechen hierbei nicht von Gruppen, sondern von sogenannten *Partitionen* oder *Fenstern*. Bei einem **OVER**-Ausdruck handelt sich also um eine *Aggregationüber Partitionen*. Die Syntax lautet:

```
SELECT [spaltenliste ...],
        aggregatfunktion (spalte) OVER (PARTITION BY     spalte_x, ..., spalte_y) [...]
FROM tabelle ... ;
```

Die Funktion vor dem reservierten Wort **OVER** heißt *Fensterfunktion* und muss eine Aggregatfunktion sein. Der Ausdruck **OVER** definiert hier mit Hilfe der Anweisung **PARTITION BY**ein Fenster, d.h. eine benutzerdefinierte Gruppe von Datensätzen der Ergebnismenge, auf die die Fensterfunktion angewendet wird. Für ein Fenster mit **PARTITION BY** bestimmt dabei die Spaltenliste das Partitionierungskriterium. Lässt man die Klammer hinter **OVER** leer, so wird über den gesamten Datenbestand aggregiert. Eine Partition kann in der Klammer mit **ORDER BY** sortiert und so die logische Reihenfolge der Berechnungen durch die Fensterfunktion vorgegeben werden. Die Fensterfunktion wird dabei für jeden auszugebenden Datensatz separat ausgeführt, ganz im Gegensatz zu **GROUP BY**. Bei sehr großen Datenbeständen kann das zu erheblichen Laufzeiten führen.

An dieser Stelle sollten wir die auf den ersten Blick etwas komplizierte Syntax und Funktionsweise dieses Ausdrucks anhand eines Beispiels näher betrachten.

Beispiel 4.11. Wollen wir jeden einzelnen Titel mit der Anzahl der insgesamt in seiner Reihe erschienen Titel herausfinden, ohne sie dabei zu gruppieren, so können wir die folgende Anweisung anwenden:

```
SELECT titel, reihe, count(*) OVER (PARTITION  BY reihe) FROM alben;
```

Damit erhalten wir die Ausgabe:

titel	reihe	COUNT(*) OVER (PARTITION BY reihe)
Der große Graben	Asterix	5
Asterix, der Gallier	Asterix	5
Asterix und Kleopatra	Asterix	5
Asterix als Legionär	Asterix	5
Die Trabantenstadt	Asterix	5
Das Kriminalmuseum	Franka	2
Das Meisterwerk	Franka	2
Gespenster Geschichten	Gespenster Geschichten	1

Möchten wir die Titel mit verschiedenen Aggregationsgrößen sehen, beispielsweise die Anzahl der im selben Jahr derselben Reihe erschienen Titel und den mittleren Preis pro Reihe insgesamt, so können wir das mit der folgenden Abfrage erreichen:

```
SELECT titel, reihe, jahr,
        count(*) OVER (PARTITION BY reihe, jahr) AS 'Anzahl(reihe,jahr)' ,
        AVG(preis) OVER (PARTITION BY reihe) AS 'mittlerer Preis(reihe)'
FROM alben ;
```

Damit erhalten wir dann:

titel	reihe	jahr	Anzahl(reihe,jahr)	mittlerer Preis(reihe)
Asterix als Legionär	Asterix	NULL	1	3.440000
Asterix und Kleopatra	Asterix	1968	2	3.440000
Asterix, der Gallier	Asterix	1968	2	3.440000
Die Trabantenstadt	Asterix	1974	1	3.440000
Der große Graben	Asterix	1980	1	3.440000
Das Kriminalmuseum	Franka	1985	1	8.800000
Das Meisterwerk	Franka	1986	1	8.800000
Gespenster Geschichten	Gespenster Geschichten	1974	1	1.200000

Wollen wir abschließend uns für jeden Titel die Preisabweichung vom mittleren Preis der Reihe anzeigen lassen, so programmieren wir:

```
SELECT titel, preis,
preis AVG(preis) OVER (PARTITION BY reihe) AS 'Preisdifferenz' FROM alben
;
```

Damit erhalten wir dann:

titel	preis	Preisdifferenz
Asterix und Kleopatra	2.80	-0.640000
Asterix, der Gallier	2.80	-0.640000
Der große Graben	5.00	1.560000
Die Trabantenstadt	3.80	0.360000
Asterix als Legionär	2.80	-0.640000
Das Kriminalmuseum	8.80	0.000000
Das Meisterwerk	8.80	0.000000
Gespenster Geschichten	1.20	0.000000

Eine solche Abfrage wäre mit anderen Ausdrücken wie **GROUP BY**, verschachtelten Abfragen oder Kombinationen daraus nicht möglich. Der **OVER**-**PARTITION**-Ausdruck ist damit eine echte Spracherweiterung von SQL, die den WHILE-Schleifen imperativer Programmiersprachen entspricht. Wenn die gewünschte Ergebnismenge dagegen auch mit einem **GROUP BY** bzw. verschachtelten Abfragen erreicht werden kann, sollte **OVER**-**PARTITION** nicht verwendet werden, da ein **OVER**-**PARTITION**-Ausdruck sehr rechenaufwändig ist.