Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Datenmanagement & -analyse

Datenbankabfragen

Prof. Dr. Christoph M. Flath
Lehrstuhl für WI & BA
Julius-Maximilians-Universität Würzburg

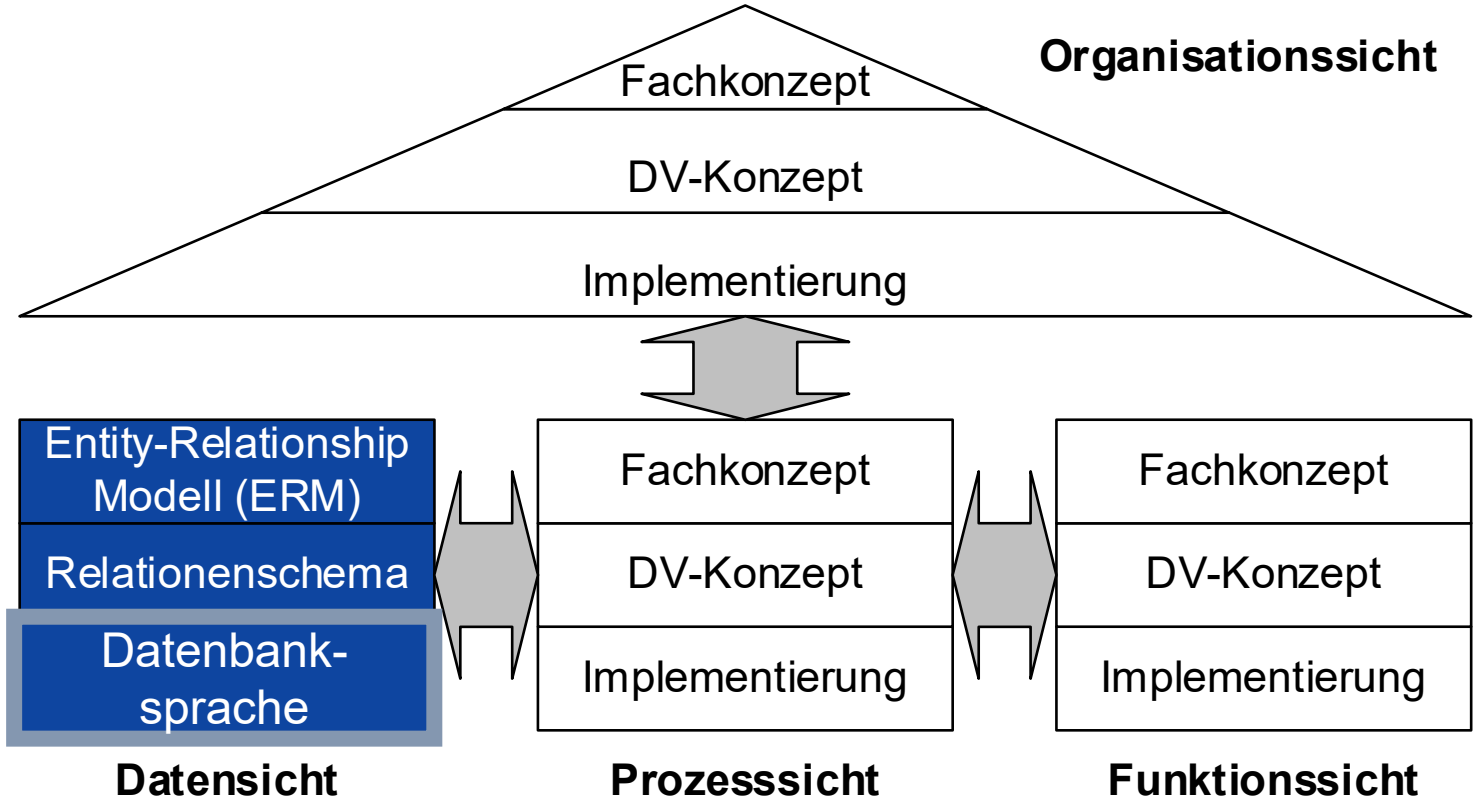
Sommersemester 2021

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

1. **Datenbank- und Abfragesprachen**
2. **Abfragen mit dplyr**
3. **SQL als Data Query Language**

Architektur Integrierter Informationssysteme



Scheer (1992)

Datenbanksprachen

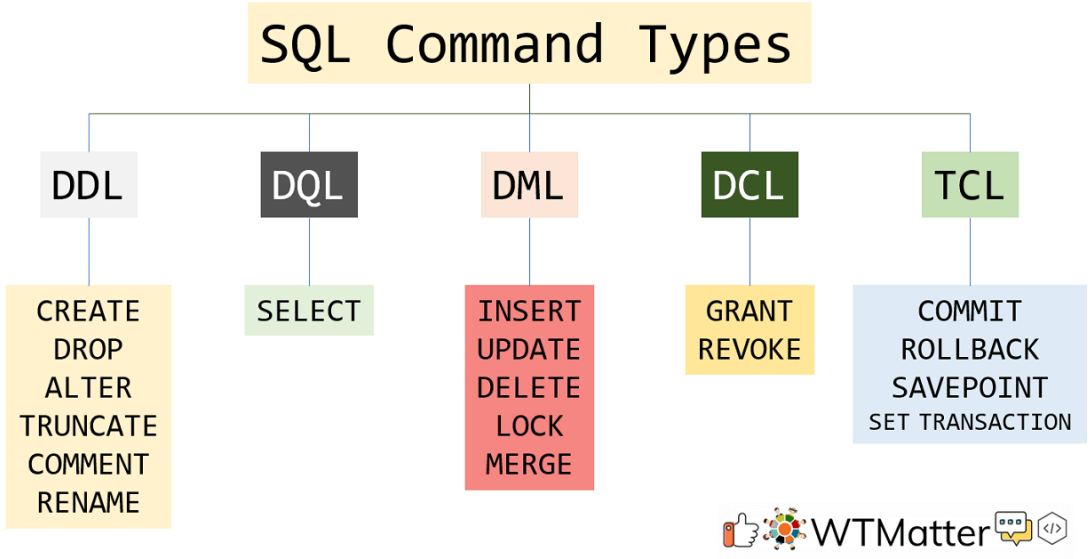
- Formale Sprachen, die für den Einsatz in Datenbanksystemen entwickelt wurden
- Die Datenbanksprache erlaubt Kommunikation mit der Datenbank
- Datenbanksprachen sind speziell auf die Anforderungen des Datenbankmanagements zugeschnitten
  - Datenbankerstellung
  - Datenbankpflege
  - Datenbankabfrage
- Es gibt eine Vielzahl von Datenbanksprachen, die oft auf bestimmte Datenbankmanagementsysteme zugeschnitten sind

Es gibt eine Vielzahl von Datenbanksprachen, die oft auf bestimmte Datenbankmanagementsysteme zugeschnitten sind

- Die Standardsprache für relationale Datenbanksysteme ist SQL (structured query language)
  - Obermenge vieler, proprietärer *SQL-Dialekte*
- In analysegetriebenen Data Science Szenarien werden häufig DataFrame Ansätze für ähnliche Aufgabenstellungen genutzt. Beispiele hierfür sind
  - pandas in Python
  - dplyr in R
  - DataFramesMeta in Julia

Befehlsklassen

- SQL als Data **Definition** Language (DDL)
  - Anlegen, Ändern, Löschen von Tabellen
- SQL als Data **Query** Language (DQL)
  - Ausführen von Abfragen auf Datenbestand
- SQL als Data **Manipulation** Language (DML)
  - Einfügen, Ändern, Löschen von Daten
- SQL als Data **Control** Language (DCL)
  - Benutzer- und Transaktionsverwaltung
- SQL als **Transaction** Control Language (TCL)
  - Verwaltung von Transaktionen



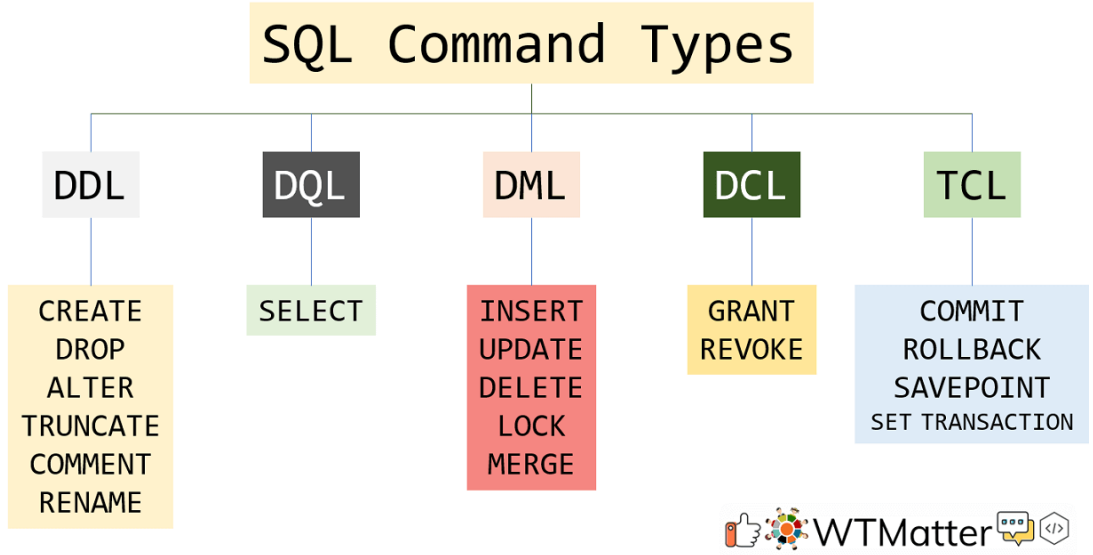
Historie der SQL-Standards [Wikipedia]

- etwa 1975: *SEQUEL = Structured English Query Language*, der Vorläufer von *SQL*, wird für das Projekt System R von IBM entwickelt.
- 1979: *SQL* gelangt mit *Oracle V2* erstmals durch *Relational Software Inc.* auf den Markt.
- 1986: *SQL1* wird von ANSI als Standard verabschiedet.
- 1987: *SQL1* wird von der Internationalen Organisation für Normung (ISO) als Standard verabschiedet und 1989 nochmals überarbeitet.
- 1992: Der Standard *SQL2* oder *SQL-92* wird von der ISO verabschiedet.
- 1999: *SQL3* oder *SQL:1999* wird verabschiedet. Im Rahmen dieser Überarbeitung werden weitere wichtige Features (wie etwa Trigger oder rekursive Abfragen) hinzugefügt.
- 2003: *SQL:2003*. Als neue Features werden aufgenommen SQL/XML, Window functions, Sequences.
- 2006: *SQL/XML:2006*. Erweiterungen für SQL/XML[2].
- 2008: *SQL:2008* bzw. ISO/IEC 9075:2008. Als neue Features werden aufgenommen INSTEAD OF-Trigger, TRUNCATE-Statement und FETCH Klausel.
- 2011: *SQL:2011* bzw. ISO/IEC 9075:2011. Als neue Features werden aufgenommen „Zeitbezogene Daten“ (PERIOD FOR). Es gibt Erweiterungen für Window functions und die FETCH Klausel.
- 2016: *SQL:2016* bzw. ISO/IEC 9075:2016. Als neue Features werden aufgenommen JSON und „row pattern matching“.
- 2019: *SQL/MDA:2019*. Erweiterungen für einen Datentyp „mehrdimensionales Feld“.

SQL wird uns so schnell nicht verlassen!

Befehlsklassen

- SQL als Data **Definition** Language (DDL)
  - Anlegen, Ändern, Löschen von Tabellen
- SQL als Data **Query** Language (DQL)
  - Ausführen von Abfragen auf Datenbestand
- SQL als Data **Manipulation** Language (DML)
  - Einfügen, Ändern, Löschen von Daten
- SQL als Data **Control** Language (DCL)
  - Benutzer- und Transaktionsverwaltung
- SQL als **Transaction** Control Language (TCL)
  - Verwaltung von Transaktionen



TYPISCHER DATA SCIENCE FOKUS

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

Den Menschen nicht aus dem Auge verlieren

86

The real first question is why are people more productive with DataFrame abstractions than pure SQL abstractions.

TLDR; SQL is not geared around the (human) development and debugging process, DataFrames are.

The main reason is that DataFrame abstractions allow you to construct SQL statements whilst avoiding verbose and illegible nesting. The pattern of writing nested routines, commenting them out to check them, and then uncommenting them is replaced by single lines of transformation. You can naturally run things line by line in a repl (even in Spark) and view the results.

Wir fangen heute mit einer High-Level Query language in R an (dplyr)

Wir nutzen R für den praktischen Einstieg in Data Science

- R ist nicht die einzige Sprache, die für die Datenanalyse verwendet werden kann.
- Warum R und nicht eine andere?
- Ein paar Ideen
  - Open Source
  - Interaktive Skriptsprache
  - Interne Datenstrukturen entsprechen der Logik der Datenanalyse
  - Gute Erweiterbarkeit durch Pakete und aktive Community
    - Erstklassige Visualisierungsmöglichkeiten (ggplot)
    - Effiziente Datenabfragesprache (dplyr)

http://www.burns-stat.com/documents/tutorials/why-use-the-r-language/

Das tidyverse

- *'A collection of R packages that share common philosophies and are designed to work together'*

→ *'Solve complex problems by combining **simple, uniform pieces**!'*

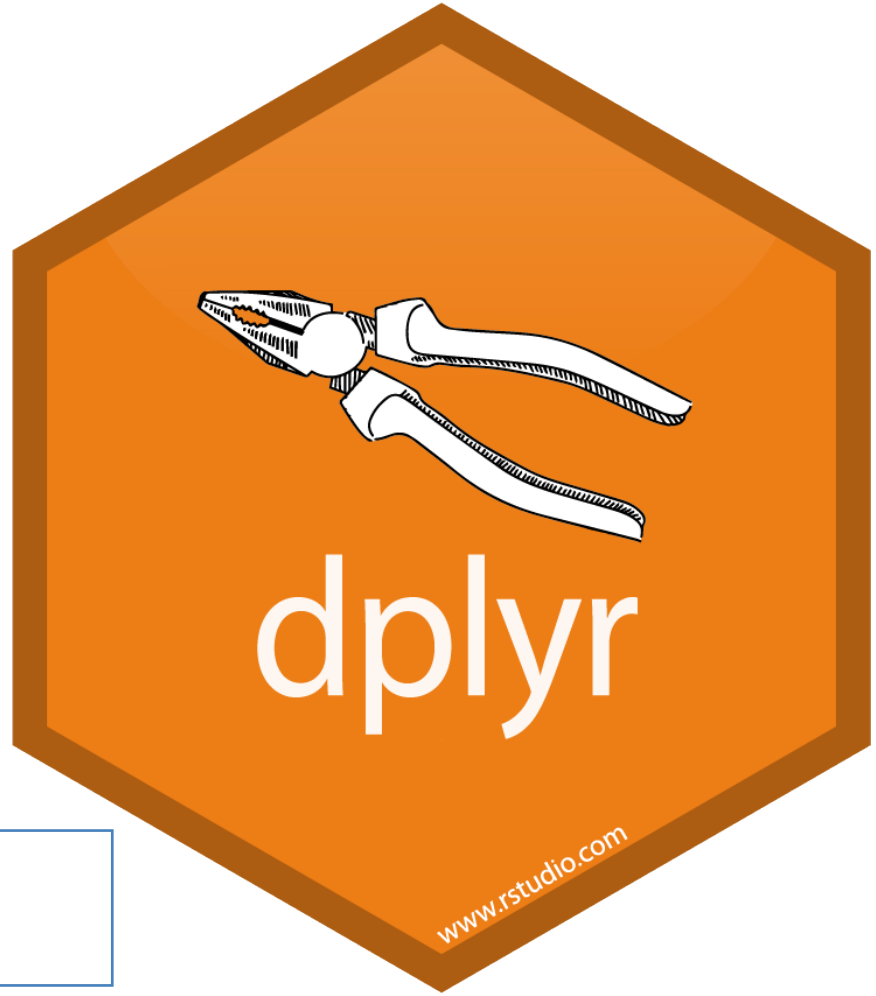
- **Vorhandene Datenstrukturen wiederverwenden.**
- **Einfache Funktionen mit der *Pipe* zusammenstellen.**

  "No matter how complex and polished the individual operations are, it is often the quality of the glue that most directly determines the power of the system."
- Paradigma der funktionalen Programmierung
- **Entwurf für Menschen.**

  "Programs must be written for people to read, and only incidentally for machines to execute."

Das dplyr package

> “A fast, consistent tool for working with data frame like objects, both in memory and out of memory.“



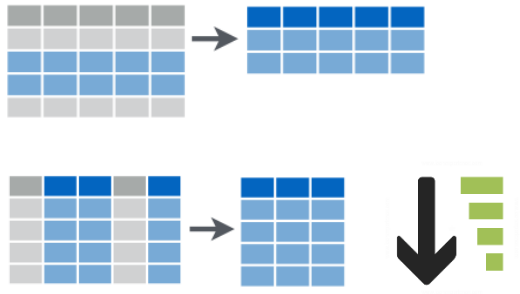
- dplyr bietet insbesondere
  - eine Reihe von einfachen Verben und Aktionen
  - den Split-Apply-Combine-Ansatz
  - Übersetzungsmöglichkeiten nach SQL mit dbplyr

Grundlage: "Data manipulation with dplyr" von Hadley Wickham

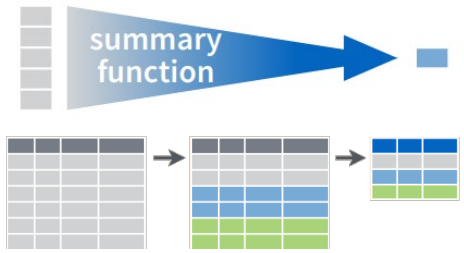
http://datascience.la/hadley-wickhams-dplyr-tutorial-at-user-2014-part-1/

Die vereinfachte Welt der Abfragen

Einfache Tabellenoperationen

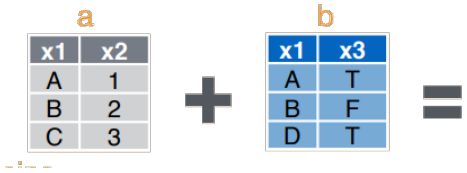


Gruppieren & Zusammenfassen



Fensteroperationen

Verbundoperationen



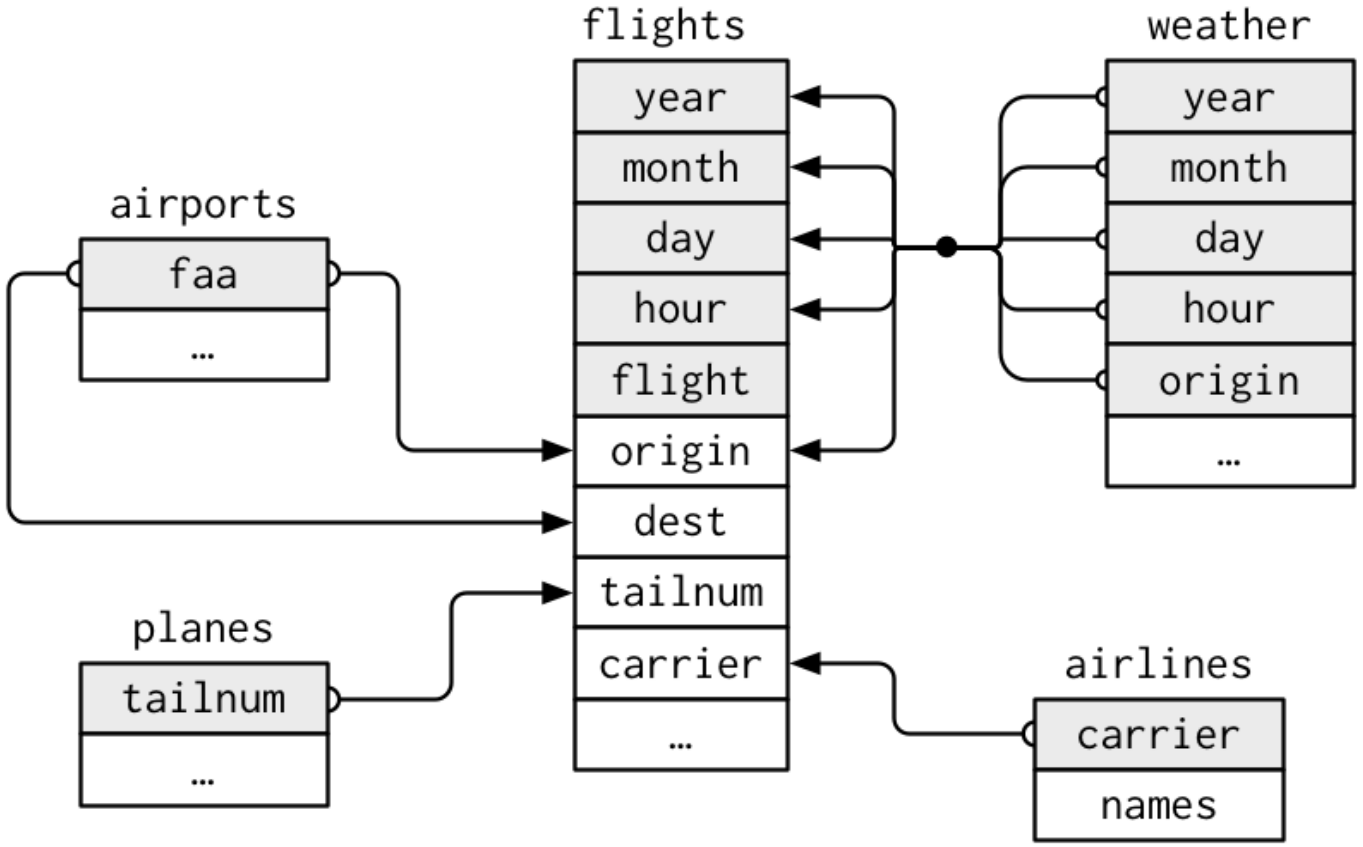
Funktionen auf einer Tabelle

Funktionen auf mehreren Tabellen

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

Die nycflights13 Datenbank



Die grundlegenden Verben von dplyr

- select: Spalten nach Namen auswählen
- filter: wählt Zeilen aus, die den Kriterien entsprechen
- arrange: Zeilen neu anordnen
- mutate: neue Variablen hinzufügen

Semantik

- Erstes Argument ist ein data.frame
- Nachfolgende Argumente sagen, was mit dem data.frame geschehen soll
- Gibt immer einen data.frame zurück, modifiziert niemals das ursprüngliche Objekt

df

color	value
blue	1
black	2
blue	3
blue	4
black	5

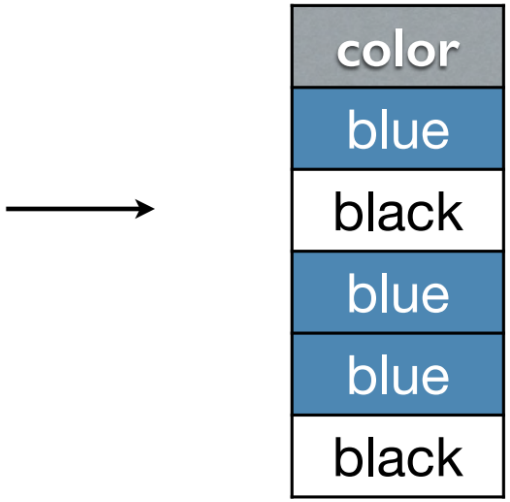
df2

color	value
4	1
1	2
5	3
3	4
2	5

select

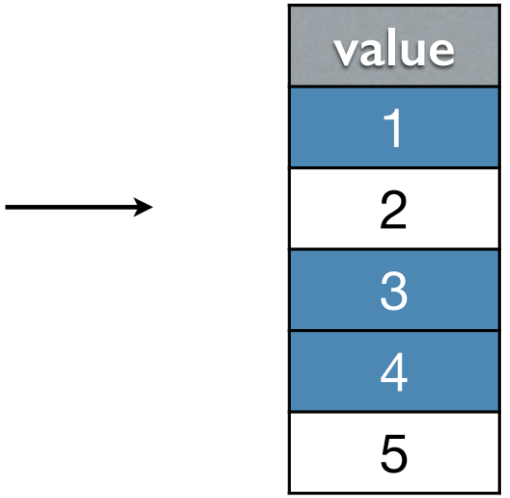
- Erlaubt das Festlegen der anzuzeigenden Spalten

```
select(df, color)
```



color
blue
black
blue
blue
black

```
select(df, -color)
```



value
1
2
3
4
5

filter

- Filtert einen data.frame, um Zeilen zu identifizieren, die bestimmte Spaltenkriterien erfüllen

```
filter(df, color == "blue")
```

- Weitere Filterkriterien können als zusätzliche Argumente übergeben werden

df

color	value
blue	1
black	2
blue	3
blue	4
black	5

color	value
blue	1
blue	3
blue	4

Filter (2)

- Wenn aus einer Weißliste zulässige Werte gewählt oder aus einer Schwarzliste unzulässige Werte ausgeschlossen werden sollen können wir %in% nutzen

df

color	value
blue	1
black	2
blue	3
blue	4
black	5

→

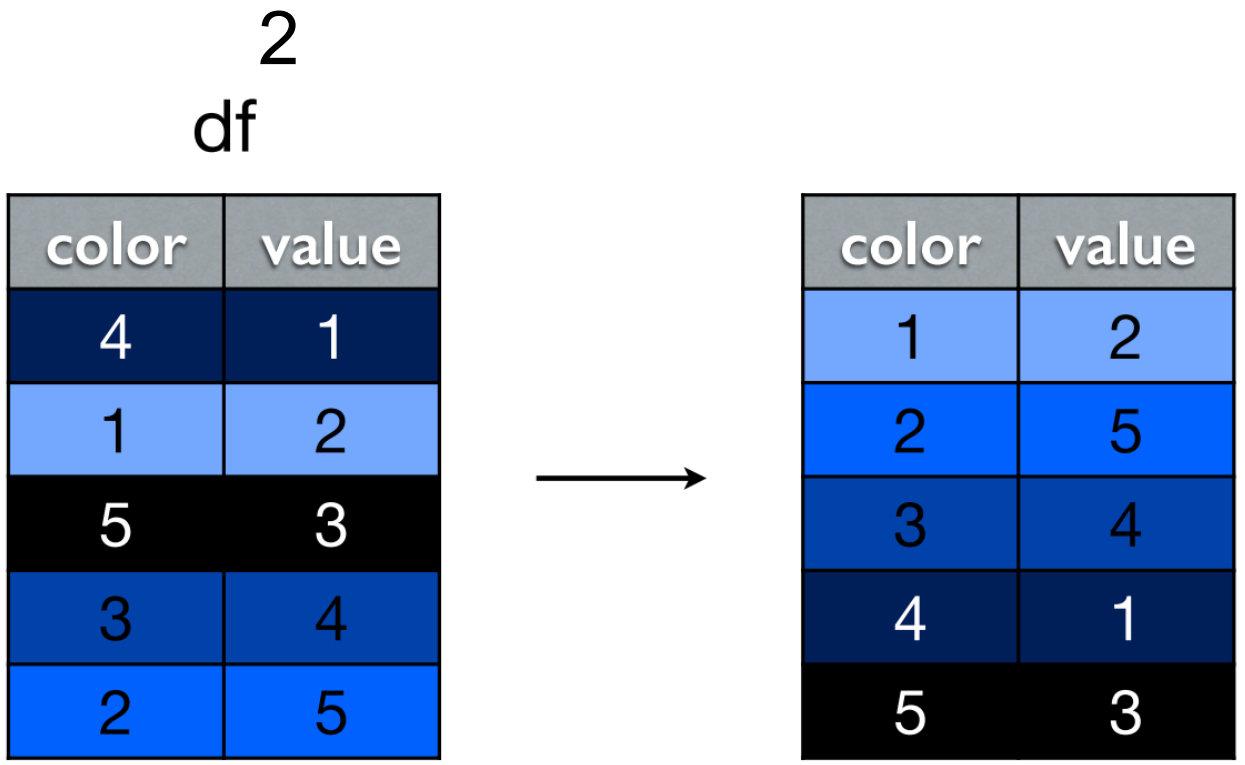
color	value
blue	1
blue	4

```
filter(df, value %in% c(1, 4))
```

arrange

Sortiert einen data.frame entsprechend der übergebenen Attribute

```
arrange(df2, color)
```



arrange (2)

```
arrange(df2, desc(color))
```

df2

color	value
4	1
1	2
5	3
3	4
2	5

color	value
5	3
4	1
3	4
2	5
1	2

Mutate [DML]

Mutate erzeugt neue Variablen im Datensatz (aus anderen Variablen):

```
mutate(df,
       double=2*value)
```

df

color	value
blue	1
black	2
blue	3
blue	4
black	5

→

color	value	double
blue	1	2
black	2	4
blue	3	6
blue	4	8
black	5	10

```
mutate(df,
       double=2 * value,
       quadruple = 4 * value)
```

df

color	value
blue	1
black	2
blue	3
blue	4
black	5

→

color	value	double	quadruple
blue	1	2	4
black	2	4	8
blue	3	6	12
blue	4	8	16
black	5	10	20

nycflights13 Beispielabfragen

- Welche Flüge hatten die größte Verspätung?
- Welche Flüge haben die meiste Verspätung während des Fluges abbauen können?

- Finde alle Flüge von
  - SFO oder OAK
  - im Januar
  - Mehr als eine Stunde Verspätung
  - Abflug zwischen Mitternacht und 5 Uhr morgens
  - Ankunftverspätung mehr als doppelt so hoch wie Abflugverspätung

Agenda

summarise

- Mehrere Variablen mit Hilfer einer summary-Funktion auf einen einzigen Wert reduzieren
- Typische Funktionen
  - `min(x), median(x), max(x),`
  - `quantile(x, p)`
  - `n(), n_distinct(), sum(x), mean(x)`
  - `sum(x > 10), mean(x > 10)`
  - `sd(x), var(x), iqr(x), mad(x)`

- Nicht sehr hilfreich bei ungruppierten Daten

```
summarise(df, total=sum(value))
```

df

color	value
blue	1
black	2
blue	3
blue	4
black	5

→

total
15

Was wäre wohl interessanter?

Grouping data based on a variable

- Erstellen Sie einen neuen data.frame, der die zugrundeliegende Gruppierung explizit hinterlegt hat

```
grouped_by_color <- group_by(df, color)
```

- Beim Zusammenfassen des gruppierten data.frame bleiben die Gruppierungen erhalten

```
summarise(grouped_by_color, total=sum(value))
```

df

color	value
blue	1
black	2
blue	3
blue	4
black	5

color	total
blue	8
black	7

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

Aufeinander aufbauende Abfragen führen zu repetitiven Code

- Ein typischer Workflow sieht folgendermaßen aus:
  - Filtern
  - Gruppieren
  - Zusammenfassen
  - Sortieren

- Da dplyr das Ursprungsobjekt nicht verändert müssen wir Zwischenergebnisse speichern

```
df.filtered <- filter (df, …)
df.filtered.grouped <- group_by(df.filtered, …)
df.summary <- summarise(df.filtered.grouped, …)
df.summary.arranged <- arrange(df.summary, …)
```

Pipelining nutzen die Struktur der dplyr Semantik

- Idee: `x %>% f(y)` ⇔ `f(x, y)`
  - Interpretation des Pipeline Operators `%>%`: links wird an rechts übergeben

- Einzeltabellen-Abfragen haben stets einen data.frame als erstes Argument und geben einen data.frame als Ergebnis zurück → perfekte Passung zur Pipeline-Logik

- Daher wird der Code aus der letzten Folie äquivalent durch die folgende Pipeline erfasst

```
df %>%
    filter(…) %>%
    group_by(…) %>%
    summarise(…) %>%
    arrange(…)
```

nycflights13 Beispielabfragen

Beantworten Sie die folgenden Fragen mit geeigneten Pipelines:

- Welche Destinationen haben die größten Verspätungen?
- Welche Flugverbindungen finden an jedem Tag statt? Wohin gehen diese?
- Welche Fluggesellschaft ist die pünktlichste?

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Fensterfunktionen

- Eine Fensterfunktion ist eine Variante einer Aggregationsfunktion
- Während eine Aggregationsfunktion, wie sum() und mean(), n Eingaben annimmt und einen einzigen Wert zurückgibt, gibt eine Fensterfunktion n Werte zurück
- Die Ausgabe einer Fensterfunktion kann von allen Eingabewerten abhängen
- Rangfolge- und Ordnungsfunktionen: row_number(), min_rank (RANK in SQL), dense_rank(), cume_dist(), percent_rank() und ntile().
- Diese Funktionen nehmen alle einen Vektor, nach dem sie sortieren, und geben verschiedene Arten von Rängen zurück
- Mit den Offsets lead() und lag() können Sie auf den vorherigen und den nächsten Wert in einem Vektor zugreifen, was die Berechnung von Differenzen und Trends erleichtert
- Kumulative Aggregate: cumsum(), cummin()

Rankingfunktionen

- Die Ranking-Funktionen sind Variationen eines Themas, die sich darin unterscheiden, wie sie mit Gleichheit umgehen:
- `x <- c(1,3, 2, 2, 1)`
- row_number(x) 1 5 3 4 2
- min_rank(x) 1 5 3 3 1
- dense_rank(x) 1 3 2 2 1
- Zwei weitere Ranking-Funktionen geben Zahlen zwischen 0 und 1 zurück.
  - percent_rank() liefert den prozentualen Anteil des Rangs
  - cume_dist() gibt den Anteil der Werte an, die kleiner oder gleich dem aktuellen Wert sind.
  - Diese sind nützlich, wenn Sie (z. B.) die obersten 10 % der Datensätze innerhalb jeder Gruppe auswählen wollen

Rangfolge- und Ordnungsfunktionen

- row_number(), min_rank (RANK in SQL), dense_rank(), cume_dist(), percent_rank() und ntile().
- Diese Funktionen nehmen alle einen Vektor, nach dem sie sortieren, und geben verschiedene Arten von Rängen zurück
- Mit den Offsets lead() und lag() können Sie auf den vorherigen und den nächsten Wert in einem Vektor zugreifen, was die Berechnung von Differenzen und Trends erleichtert
- Kumulative Aggregate: cumsum(), cummin()

Vorgänger und Nachfolgeroperationen

lead() und lag() erzeugen versetzte Versionen eines Eingangsvektors, die entweder vor oder hinter dem ursprünglichen Vektor liegen.

```
x <- c(1, 2, 3, 4, 5)
lead(x) #> [1] 2 3 4 5 NA
lag(x) #> [1] NA 1 2 3 4
```

Sie können sie verwenden, um:

- Gab es eine Änderung? `x != lag(x)`
- Absolute Änderung? `x - lag(x)`
- Prozentuale Änderung? `(x - lag(x)) / x`
- Verfielwachung? `x / lag(x)`
- Zuvor falsch, jetzt wahr? `!lag(x) & x`

nycflights13 Beispielabfragen

- Verwenden Sie die Ranking-Funktion, um die pünktlichsten Fluggesellschaften und die pünktlichsten Fahrten (Start-Ziel-Kombinationen) zu identifizieren

- Bestimmen Sie die monatliche Entwicklung der kumulierten Ankunftsverspätungen für Southwest Airlines ab

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

Normalisierte Daten liegen nicht in einer einzelnen Tabelle

Wenn wir uns auf relationale Datenbanken verlassen oder Daten aus verschiedenen Systemen verwenden, müssen wir Datensätze effizient kombinieren

Beispiele

- Wie können wir Verspätungen an Flughäfen auf einer Karte anzeigen?
- Hängt die Anzahl der Triebwerke mit der Geschwindigkeit zusammen?
- Sind Boeing-Flugzeuge pünktlicher als Airbus-Flugzeuge?

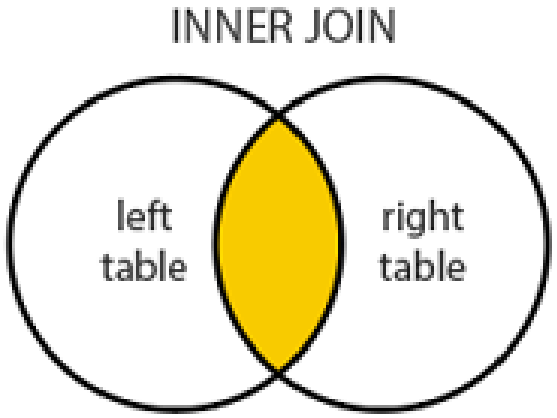
name	instrument
John	guitar
Paul	bass
George	guitar
Ringo	drums
Stuart	bass
Pete	drums

+

name	band
John	T
Paul	T
George	T
Ringo	T
Brian	F

= ?

Typische Verbundoperationen



x

name	instrument
John	guitar
Paul	bass
George	guitar
Ringo	drums
Stuart	bass
Pete	drums

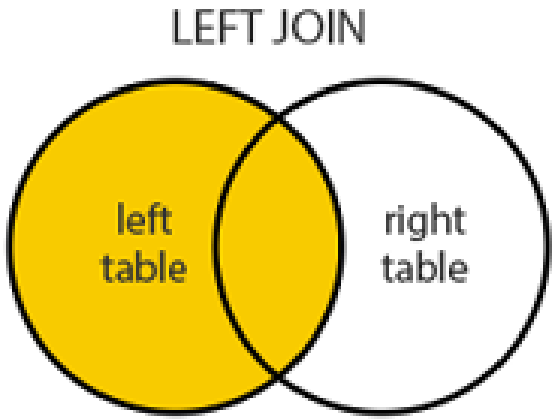
+ y

name	band
John	T
Paul	T
George	T
Ringo	T
Brian	F

=

name	instrument	band
John	guitar	T
Paul	bass	T
George	guitar	T
Ringo	drums	T

```
inner_join(x, y)
```



x

name	instrument
John	guitar
Paul	bass
George	guitar
Ringo	drums
Stuart	bass
Pete	drums

+ y

name	band
John	T
Paul	T
George	T
Ringo	T
Brian	F

=

name	instrument	band
John	guitar	T
Paul	bass	T
George	guitar	T
Ringo	drums	T
Stuart	bass	NA
Pete	drums	NA

```
left_join(x, y)
```

nycflights13 Beispielabfragen

- Welche Wetterbedingungen liegen bei Verspätungen in New York typischerweise vor?

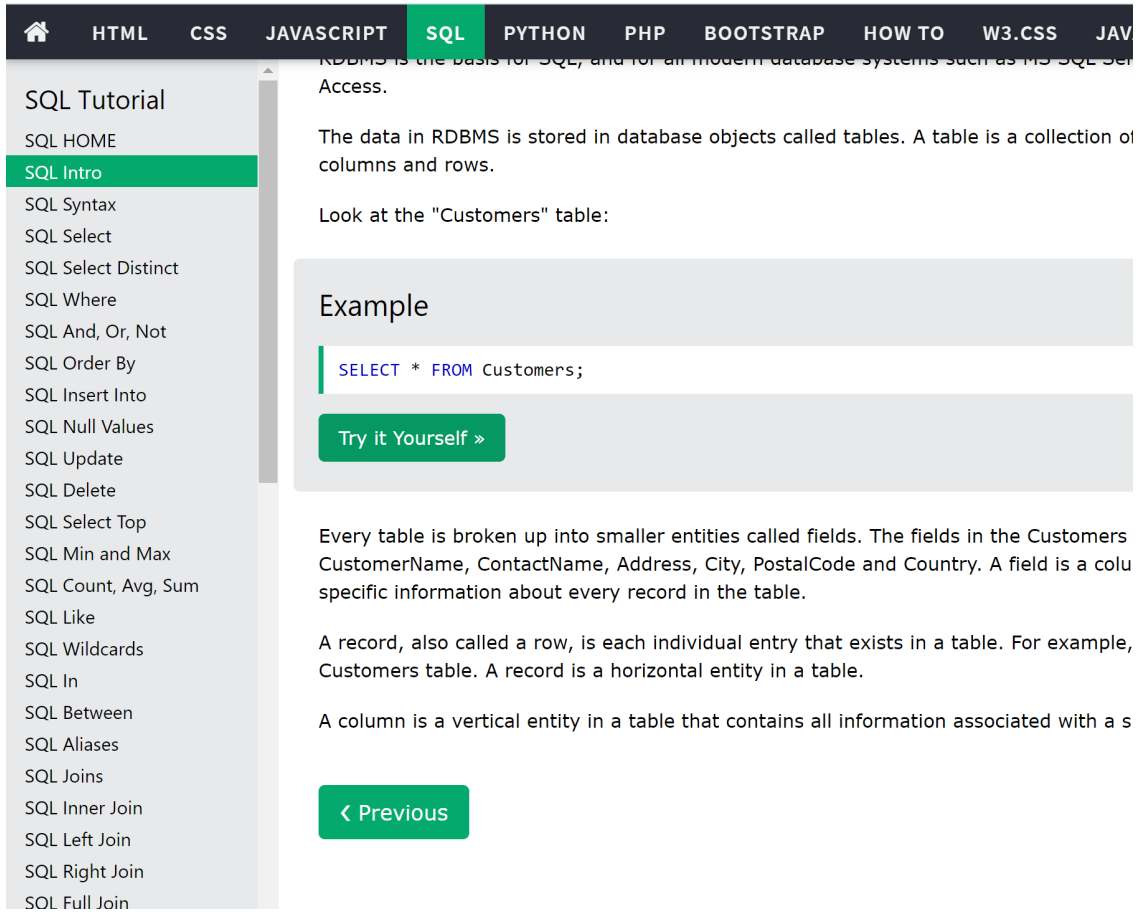
- Sind Boeing-Flugzeuge pünktlicher als Airbus-Flugzeuge?

Agenda

w3schools sind dein Freund

- Ein umfangreiches SQL Tutorial gibt es unter https://www.w3schools.com/sql/sql_intro.asp

- https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all bietet eine voll funktionsfähige Demo-Datenbank



Aufbau einer SQL-Abfrage

- Syntax

```
SELECT [ALL | DISTINCT] spaltenname1, ….
FROM tabelle1 [, tabelle2, …]
[WHERE bedingung];
```

- Beispiel
  - Zeige alle Kunden an.

```
SELECT * FROM Customers;
```

  - Zeige alle Artikel mit Preis von weniger als 50 an.

```
SELECT ProductName, Price
FROM Products
WHERE Price < 50;
```

Wie sieht die zugehörige dplyr Abfrage-Pipeline aus?

Weitere einfache Abfragen

- Rechenoperationen

```
SELECT Artikelname, Einzelpreis * 1.1149 AS Dollarpreis
FROM Artikel
WHERE Artikelname = 'Superbio Apfelsaft 1L';
```

- Kombination von WHERE-Klauseln

```
SELECT Artikelname, Einzelpreis
FROM Artikel
WHERE (Lagerbestand < 10 OR Lagerbestand > 100) AND Einzelpreis
< 1.99;
```

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

BETWEEN

```
SELECT ProductName, Price
FROM Products
WHERE Price BETWEEN 50 AND 100;
```

Alternativ

```
SELECT ProductName, Price
FROM Products
WHERE Price>=50 AND Price <= 100;
```

IN

```
SELECT *
FROM Products
WHERE Price IN (12, 81, 97);
```

- Alternativ

```
SELECT *
FROM Products
WHERE Price = 12 OR Price = 81 OR Price = 97;
```

LIKE

- Ähnliche Zeichenketten vergleichen
- Wildcard-Symbole: „`%`“ sowie „`_`“
- `%`: beliebig viele Zeichen in Zeichenkette
  - Beispiel: `'A%BC'` liefert `'ADEFGHBC'` oder `'ABC'`, …
- `_`: genau ein Zeichen in Zeichenkette
  - Beispiel: `'A_BC'` liefert `'ADBC'`, …

In dplyr nicht direkt modelliert aber über die Stringoperationen möglich

LIKE

- Beispiele

```
SELECT ProductID, ProductName, SupplierID
FROM Products
WHERE ProductName LIKE '%Chef%';
```

- Mögliche Ausgabe

ProductID	ProductName	SupplierID
4	Chef Anton's Cajun Seasoning	2
5	Chef Anton's Gumbo Mix	2

Rechenfunktionen

- **ROUND([, D])**

```
SELECT Artikelname, ROUND(Einzelpreis)
FROM Artikel
WHERE ROUND(Einzelpreis, -1) >= 20;
```

- **YEAR()**

```
SELECT BestellNr, YEAR(Bestelldatum)
FROM Bestellungen
WHERE YEAR(Bestelldatum) = 2018;
```

- Analog
  MONTH(), **DAY()**, **HOUR()** usw.

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

Aggregation

- Die Aggregation erlaubt die **Gruppierung gleicher Tupel** auf die eine **Aggregatfunktion** angewendet wird, so dass für die gesamte Gruppe ein Wert bestimmt wird
- Typische Aggregatfunktionen sind:
  - **count, sum, min, max, avg, std**

R		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_1	b_2	c_2

$$\gamma_{A;count(*)}(R)$$

$\gamma_{A;count(*)}(R)$	
A	count(*)
a_1	2
a_2	1

Aggregation in SQL: COUNT

- **COUNT()**
  - Ermittelt **Anzahl** an Zeilen, die eine bestimmte Suchbedingung erfüllen

- Beispiel
  - Bestimme die Anzahl der Artikel

```
SELECT COUNT(*) FROM Artikel;
```

SELECT COUNT(*) FROM Products;

Aggregation in SQL: MIN / MAX

- **`MIN() / MAX()`**
  - Gibt den **minimalen** bzw. **maximalen** Wert einer Spalte zurück
  - Eine Spalte als Input
  - Spalte muss nicht numerischen Datentyp haben
  - Datumsangaben und Zeichenketten auch als Input möglich
  - Zeichenkette: lexikografischer Vergleich

- Beispiel
  - Bestimme den maximalen Lagerbestand aller Artikel

```
SELECT MAX(Lagerbestand)
FROM Artikel;
```

  - Analog: Bestimme den minimalen Lagerbestand aller Artikel

```
SELECT MIN(Lagerbestand)
FROM Artikel;
```

SELECT MAX(Quantity) FROM OrderDetails
SELECT MIN(Quantity) FROM OrderDetails

Aggregation in SQL: SUM

- **`SUM()`**
  - Ermittelt **Summe** aller Werte einer Spalte
  - Eine Spalte als Input
  - Nur numerische Datentypen erlaubt
  - **Ergebnis muss im Bereich der darstellbaren Zahlen des Input-Datentyps** liegen
  - Behandlung von Overflows je nach DBMS
  - ggf. expliziter Cast sinnvoll
    - …`SUM(CAST(Menge) AS INT)`…

- Beispiel
  - Bestimme die Summe aller Lagerbestände aller Artikel.

```
SELECT SUM(Lagerbestand)
FROM Artikel;
```

SELECT SUM(Quantity) FROM OrderDetails

Aggregation in SQL: AVG

- **`AVG()`**
  - Ermittelt **Durchschnitt** aller Werte einer Spalte
  - Eine Spalte als Input
  - Nur numerische Datentypen erlaubt
  - Auf **`NULL`**-Werte achten

l	r
150	150
200	200
350	350
NULL	0

- Beispiel

```
SELECT AVG(l) …        = 233,3333
SELECT AVG(r) …        = 175
```

SELECT AVG(Quantity) FROM OrderDetails

Aggregation in SQL: STD

- **`STD()`**
  - Ermittelt **Standardabweichung** aller Werte einer Spalte
  - Eine Spalte als Input
  - Nur numerische Datentypen erlaubt
  - Auf **`NULL`**-Werte achten

- Beispiel
  - Bestimme die Standardabweichung der Lagerbestände aller Artikel.

```
SELECT STD(Lagerbestand)
FROM Artikel;
```

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

INNER JOIN in SQL

- **SELECT** *
  FROM tabellenname$_1$
  INNER JOIN tabellenname$_2$
  ON ausdruck;

- **SELECT** *
  FROM R
  INNER JOIN L
  ON R.C = L.D;

- **Alternativ**
  SELECT *
  FROM R, L
  WHERE R.C = L.D

R		
A	B	C
1	9	1
2	8	2
3	7	2

L	
D	E
1	3
2	3

$R \bowtie_{C=D} L$				
A	B	C	D	E
1	9	1	1	3
2	8	2	2	3
3	7	2	2	3

```
SELECT *
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

Left (Outer) Join

- Beim **Left (Outer) Join** werden auch diejenigen Tupel der **linken Tabelle** mit in die Ereignisrelation aufgenommen, die **keinen** Join-Partner gefunden haben

R		
A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_3	b_2	c_3

L	
C	D
c_1	d_1
c_4	d_2

R⋈L			
A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_2	NULL
a_3	b_2	c_3	NULL

- `SELECT * FROM R LEFT OUTER JOIN L ON R.C=L.C;`

Full (Outer) Join

- Beim **Full (Outer) Join** werden auch alle Tupel mit in die Ereignisrelation aufgenommen, die **keinen** Join-Partner gefunden haben

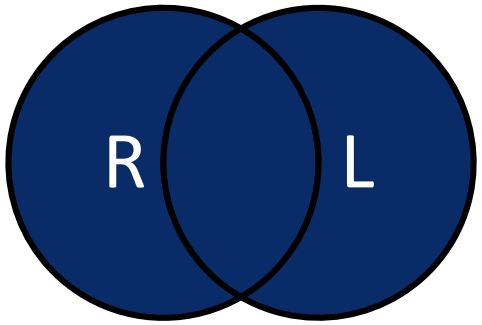
R		
A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_3	b_2	c_3

L	
C	D
c_1	d_1
c_4	d_2

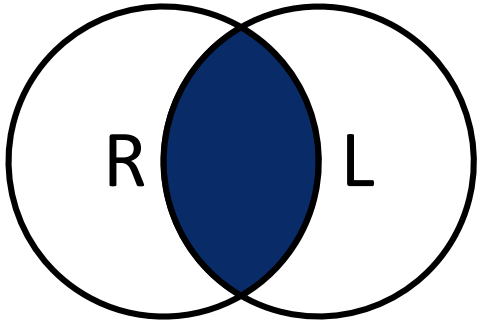
R⋈L			
A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_2	NULL
a_3	b_2	c_3	NULL
NULL	NULL	c_4	d_2

- `SELECT * FROM R FULL OUTER JOIN L ON R.C=L.C;`

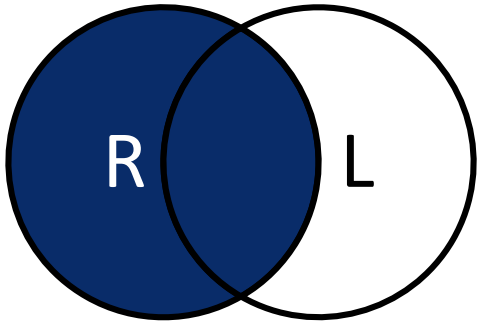
Verbundoperationen



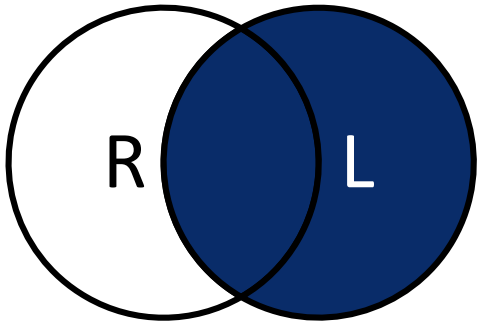
Full Outer Join



Inner (Equi) Join

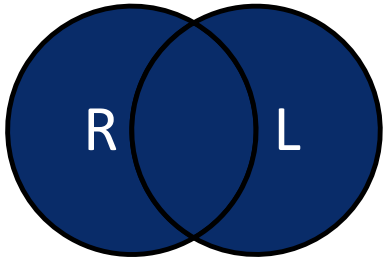


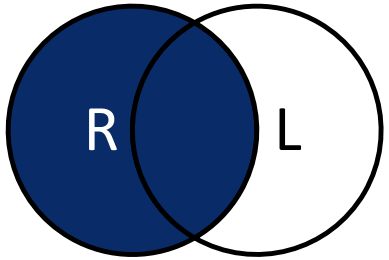
Left Join

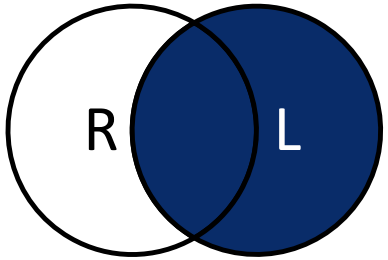


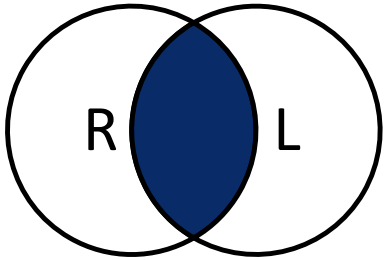
Right Join

Noch mehr Verbundoperationen

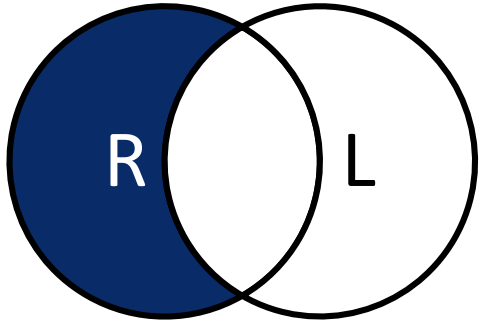


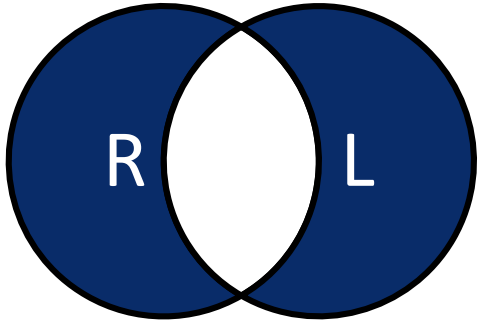






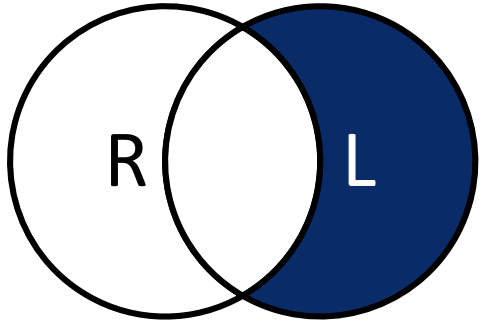

```
SELECT *
FROM R
LEFT JOIN L
ON R.C = L.C
WHERE L.C IS NULL;
```






```
SELECT *
FROM R
FULL OUTER JOIN L
ON R.C = L.C
WHERE L.C IS NULL
OR R.C IS NULL;
```

```
SELECT *
FROM R
RIGHT JOIN L
ON R.C = L.C
WHERE R.C IS NULL;
```



Agenda

Gruppierung: `GROUP BY`

- **Gruppierung** von Zeilen anhand der **Zeilenwerte**
  - Spalten, nach denen gruppiert wird, enthalten **keine doppelten** Zeileneinträge mehr
- Nach **`GROUP BY`** erfolgt Spezifikation aller Spalten, über die gruppiert wird

- Wenn über mehrere Spalten gruppiert wird, dann Untergruppen bilden
- Bei Gruppierung
  - im **`SELECT`**–Statement nur Spalten, über die gruppiert wird,
  - oder Ausdrücke, die genau einen Wert pro Gruppe zurückgeben (Aggregatfunktionen)

Gruppenbedingungen: `HAVING`

- **Optionale Auswahl** bestimmter Gruppen
- Wird entsprechend auf die **`GROUP BY`**–Klausel angewendet
- Nur Gruppen angezeigt, die **`HAVING`**-Bedingung erfüllen

- Unterschied zu **`WHERE`**-Bedingung
  - **`HAVING`** wird nicht auf Zeilen, sondern auf **Gruppen von Zeilen** angewendet
  - Zeilen, die **`WHERE`**-Bedingung nicht erfüllen, werden gar **nicht** erst in Ergebnismenge **aufgenommen**
  - Zeilen, die **`HAVING`**-Bedingung nicht erfüllen, werden in Ergebnismenge aufgenommen, aber **ausgeblendet**

- Aggregatfunktionen in **`HAVING`**-Bedingungen

Beispiel Gruppenbedingungen

- Beispiel
  - Ermittle Warengruppen, die mehr als drei Artikel enthalten

```
SELECT WarengruppenNr, COUNT(ArtikelNr) AS Anzahl
FROM Artikel
GROUP BY WarengruppenNr
HAVING Anzahl > 3;
```

- Kombination mehrerer **HAVING**-Bedingungen mit **AND** bzw. **OR**

```
SELECT CategoryID, COUNT(ProductID) AS Anzahl
FROM Products
GROUP BY CategoryID
HAVING Anzahl > 3
```

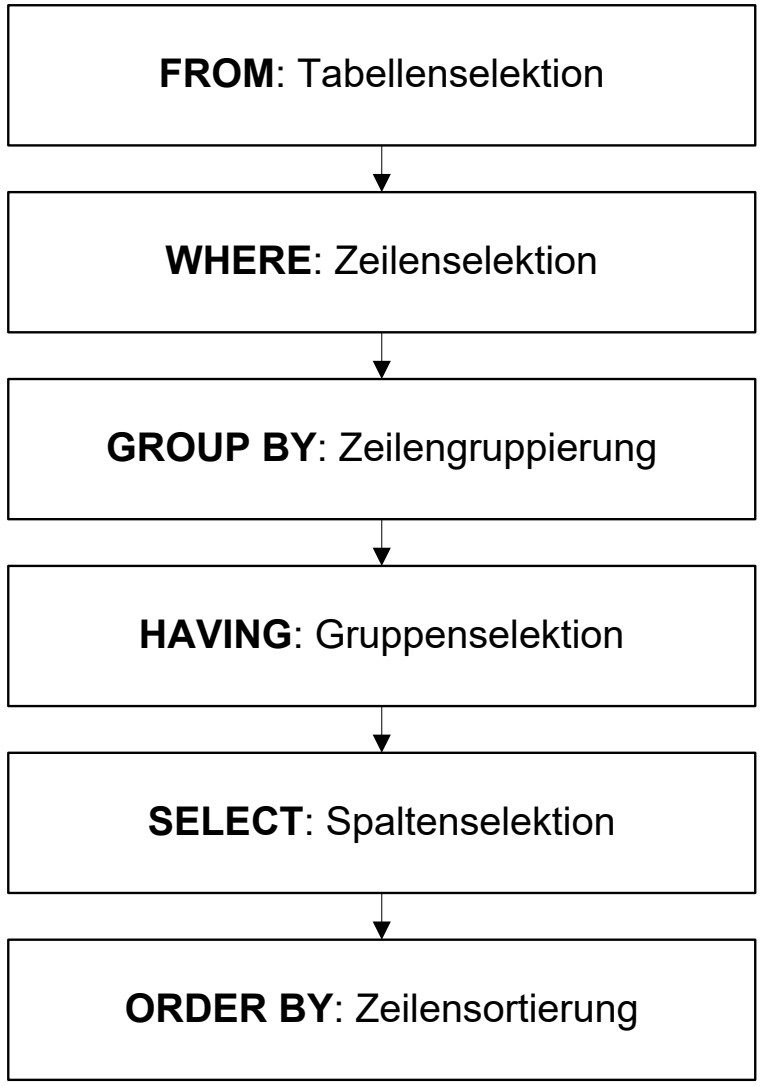
Sortieren der Ergebnismenge: `ORDER BY`

- Die Ergebnismenge kann optional nach einer oder mehreren Spalten mit **`ORDER BY`** **sortiert** werden
- Für jede Spalte kann Sortierung **aufsteigend** **`ASC`** oder **absteigend** **`DESC`** erfolgen
- Wenn keine Angabe der Sortierungsreihenfolge, dann **Default** **`ASC`**
- Beispiel

```
SELECT PLZ, Firma
FROM Kunde
ORDER BY PLZ DESC, Firma;
```

```
SELECT PostalCode, CustomerName
FROM Customers
ORDER BY PostalCode DESC, CustomerName;
```

Auswertungsreihenfolge




```
FROM tabellenname1,
  tabellename2, …
WHERE suchbedingung
GROUP BY
  spaltendefinition1,
  spaltendefinition2, …
HAVING gruppenbedingung
SELECT
  spaltendefinition1,
  spaltendefinition2, …,
  aggregatfunktion(
  spaltendefinitionx)
ORDER BY
  spaltendefinitioni
  [ASC|DESC];
```

Agenda

Unterabfrage (Subquery)

- **Strukturierter Aufbau** komplexer Abfragen
- Weitere `SELECT`…`FROM`…`WHERE`-Abfragen innerhalb eines `SELECT`-Befehls

- Innere Abfrage = Unterabfrage (Subquery)
- Äußere Abfrage = Hauptabfrage
- Weitere Verschachtelungen möglich

- Vier Arten von Unterabfragen
  - **Skalar-**, **Spalten-**, **Zeilen-** und **Tabellen**unterabfrage
  - Unterscheidung nach Ergebnistyp

Beispiel

- Beispiel
  - Ermittle Artikelnummer und Lagerbestand des Artikels mit dem maximalen Lagerbestand

Artikel
ArtikelNr
Lagerbestand

```
SELECT ArtikelNr, Lagerbestand
FROM Artikel
WHERE Lagerbestand = (
    SELECT MAX(Lagerbestand)
    FROM Artikel);
```

```
SELECT OrderID, Quantity
FROM OrderDetails
WHERE Quantity = (SELECT MAX(Quantity)
FROM OrderDetails);
```

```
SELECT OrderID, Maxanzahl FROM Orders
NATURAL JOIN (SELECT OrderID,
MAX(Quantity) AS Maxanzahl FROM OrderDetails)
```

Skalarunterabfragen

- Unterabfrage liefert genau **einen Wert** zurück (eine Spalte und eine Zeile)
- Verwendung, wo einzelne Werte zulässig sind
- Kann mit <, =, >, <= und >= verglichen werden

- Bestimme zu allen Artikeln ihre Artikelnummern und deren Abweichung vom Durchschnittspreis

```
SELECT ArtikelNr, Einkaufspreis –
      (SELECT AVG(Einkaufspreis)
       FROM Liefernachweis
       ) AS 'Abweichung vom Durchschnittspreis'
FROM Liefernachweis;
```

Liefernachweis
#ArtikelNr
Einkaufspreis

SELECT ProductID, Price - (SELECT AVG(Price) FROM Products) AS 'Abweichung vom Durchschnittspreis' FROM Products;

Zeilenunterabfragen

Artikel
ArtikelNr
Einzelpreis
Lagerbestand

- Liefert mehrere **Spalten**, aber genau eine **Zeile**
- Verwendung bei **zeilenbasierten Vergleichen**
- Können mit **<**, **=**, **>**, **<=** und **>=** verglichen werden

- Bestimme alle Artikelnummern, die den maximalen Einzelpreis und minimalen Lagerbestand haben

```
SELECT ArtikelNr
FROM Artikel
WHERE (Einzelpreis, Lagerbestand) =
    (SELECT MAX(Einzelpreis), MIN(Lagerbestand)
     FROM Artikel);
```

Zeilenunterabfragen

Artikel
ArtikelNr
Einzelpreis
Lagerbestand

- Liefert mehrere **Spalten**, aber genau eine **Zeile**
- Verwendung bei **zeilenbasierten Vergleichen**
- Können mit <, =, >, <= und >= verglichen werden
- Bestimme alle Artikelnummern, die den maximalen Einzelpreis und minimalen Lagerbestand haben

```
SELECT ArtikelNr
FROM Artikel
WHERE (Einzelpreis, Lagerbestand) =
    (SELECT MAX(Einzelpreis), MIN(Lagerbestand)
     FROM Artikel);
```

Äquivalent zu:

```
SELECT ArtikelNr
FROM Artikel
WHERE Einzelpreis =
    (SELECT MAX(Einzelpreis) FROM Artikel)
AND Lagerbestand =
    (SELECT MIN(Lagerbestand) FROM Artikel);
```

Spaltenunterabfrage

- Liefert genau **eine Spalte**, aber **mehrere Zeilen**
- Verwendung vor allem bei **Vergleichen** mit Listen, z.B. bei IN-Operator
- Können nur mit **`ANY`**, **`ALL`**, **`[NOT] IN`**, **`EXISTS`**, **`UNION`** verglichen werden

- Bestimme alle Artikelnamen, die einen Einkaufspreis von weniger als 10 haben

```
SELECT Name
FROM Artikel
WHERE ArtikelNr IN(
    SELECT ArtikelNr
    FROM Liefernachweis
    WHERE Einkaufspreis < 10);
```

Artikel
ArtikelNr
Name

Liefernachweis
#ArtikelNr
Einkaufspreis

Tabellenunterabfragen (1)

- Liefert **mehr als** eine Spalte und Zeile
- Verwendung bei **zeilenbasierten Vergleich** mit einer Liste von Zeilen (z.B. IN-Operator) oder an Stelle von Tabellen
- Können nur mit **`ANY`**, **`ALL`**, **`[NOT] IN`**, **`EXISTS`**, **`UNION`** verglichen werden

- Bestimme alle Bestellpositionen, die einen Einkaufspreis von weniger als 10 haben

```
SELECT *
FROM Bestellposition
WHERE (ArtikelNr, LieferantenNr) IN
    (SELECT ArtikelNr, LieferantenNr
     FROM Liefernachweis
     WHERE Einkaufspreis < 10);
```

Bestellposition
#ArtikelNr
#LieferantenNr

Liefernachweis
#ArtikelNr
#LieferantenNr
Einkaufspreis

Tabellenunterabfragen (2)

- Behandlung der Unterabfragen im **FROM**-Teil wie normale Tabellen
- Zuweisung eines **Alias** notwendig

Artikel
ArtikelNr
#KategorieNr
Lagerbestand

- Bestimme den Durchschnitt der Lagerbestände pro Kategorie und die Anzahl der Kategor

```
SELECT AVG(Summe), COUNT(KategorieNr)
FROM
    (SELECT SUM(Lagerbestand) AS Summe,
     KategorieNr
    FROM Artikel
    GROUP BY KategorieNr) AS Summentabelle;
```

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Agenda

Views

- **Sicht** auf Datenbank
- **Logische Relation**, die im DBMS über eine gespeicherte Abfrage vorgehalten werden
- View als **Alias für** eine **Abfrage**
- Einbinden **externer** Schemata

- Syntax

```
CREATE VIEW name_der_view AS
abfragespezifikation;
```

- Beispiel

```
CREATE VIEW
KundenOhneBestellungen AS
SELECT DISTINCT Name
FROM Kunde LEFT JOIN
Bestellung USING (KundenID)
WHERE Bestellzeit IS NULL;
```

- Zugriff auf View

```
SELECT Name FROM
KundenOhneBestellungen;
```

CREATE VIEW TeureProdukte AS SELECT * FROM [Products] WHERE PRICE > 100

Views

- Vorteile
  - **Vereinfachter Zugriff** auf Datenbankschema
    - Zugriff **ohne komplexes Wissen über Schema**
    - **keine Aufweichung** der Normalisierung
  - **Kein zusätzlicher Aufwand** zur **Vorbereitung** der **Abfrage**
    - View vom Parser bereits syntaktisch zerlegt
    - View vom Anfrageoptimierer bereits vereinfacht
- Nachteile
  - **Komplexität** der zugrundeliegenden Abfrage unterschätzt
  - Unbedachter Einsatz kann zu **Performanceproblemen** führen
- Materialisierte Sichten
  - Auch **Indexed View** (Microsoft) oder **Automatic Summary Tables** (IBM) genannt
  - Speichert Ergebnismenge einer Sicht in Tabelle ab
  - Cache-Funktion
- **Aktualisierung** materialisierter Sichten
  - Konzepte
    - Inkrementelle Updates (logbasiert)
    - Komplette Neuerstellung (einfach, aber extrem teuer)
  - Zeitpunkte
    - Transaktionsbasiert bei Update der Basistabellen
    - Zeitpunktbezogen