Datenbankbasierte Web-Anwendungen

Foreign Keys und Datenabfragen

Medieninformatik SoSe 2017

Renzo Kottmann

Zusammenfassung der vorherigen Vorlesung

- Eigene Datentypen
- Foreign Keys bei Aufzaehlungen (technisch)

Nicht besprochen:

- Umsetzung von ERM Beziehungen mittels Foreign Keys

Aktuelle Implementierung

- SQL file mit ersten Testdaten

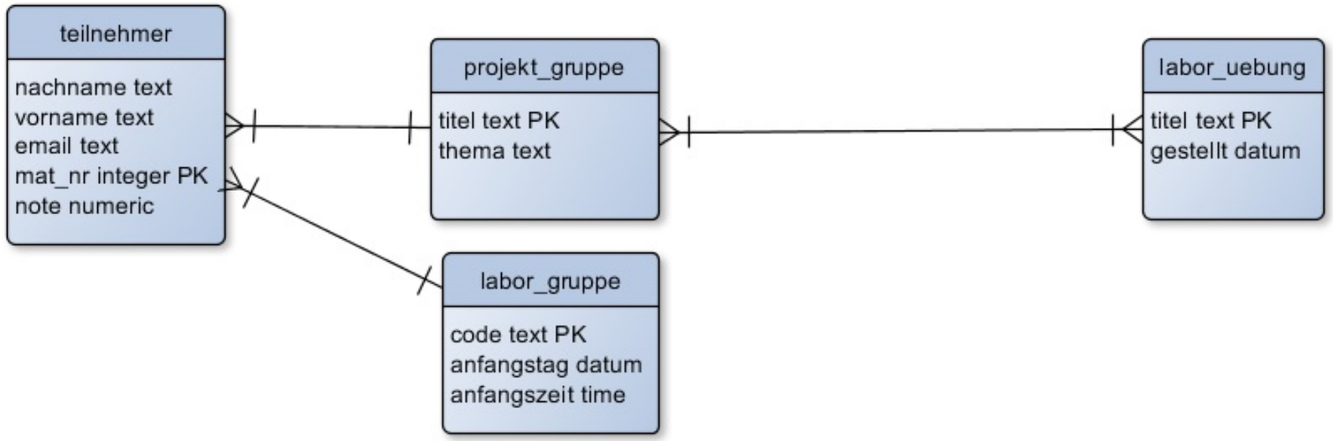
Relationships (Beziehungen) revisited

Verschiedene Entitäten können zueinander in Beziehung gesetzt werden.

- In jeder Beziehung haben Entitäten gewisse Rollen
- Beziehungen können Eigenschaften (Attribute) haben
- Beziehungen haben Kardinalitäten

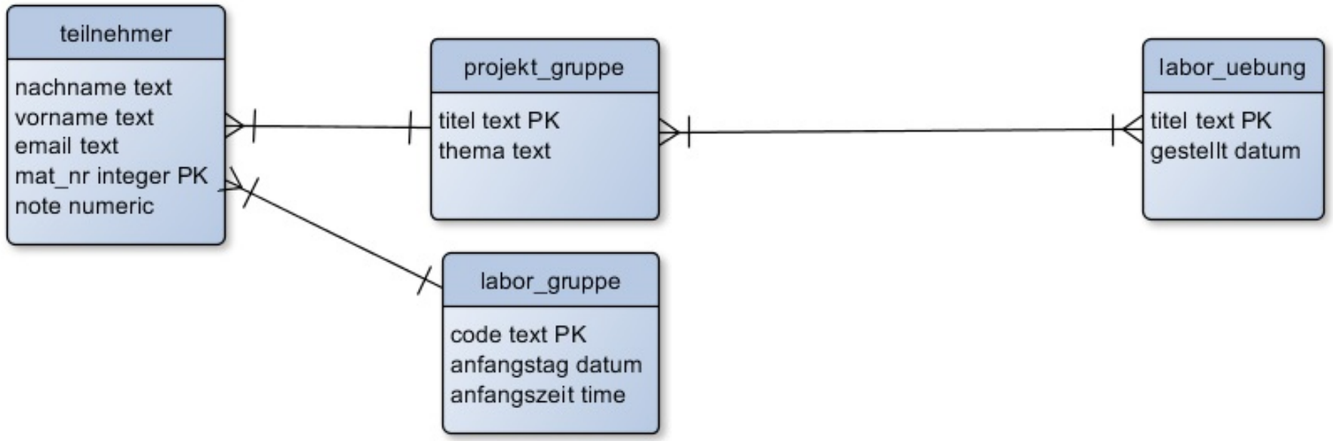
Beziehungen

- Teilenhmer ist in einer Projektgruppe und einer Laborgruppe
- Jede Projektgruppe bearbeitet mehere Laboruebungen



Beziehungen mit Foreign Keys

- Bestimmem die
  - Attribute welche die Beziehungen identifizieren
  - Kardinalitäten



Foreign Keys (one to many)

- Koennen nur eins zu viele Beziehungen umsetzen
- Werden immer von viele zu eins gesetzt

```
CREATE TABLE labor_gruppe (
  code text
    PRIMARY KEY
    CHECK ( code IN  ('w','x','y','z') ) ,
  -- Attribute fehlen
  anfangszeit time
    NOT NULL
);

CREATE TABLE teilnehmer (
  vorname text
    CHECK ( vorname != '' ),
  nachname text,
    CHECK ( vorname != '' ),
  matrikel_nr integer
    PRIMARY KEY,
  -- Attribute fehlen
  labor text
    REFERENCES labor_gruppe(code),
  geschlecht text
    REFERENCES geschlecht(name)
    ON UPDATE CASCADE
);
```

Foreign Keys (one to many)

Frage

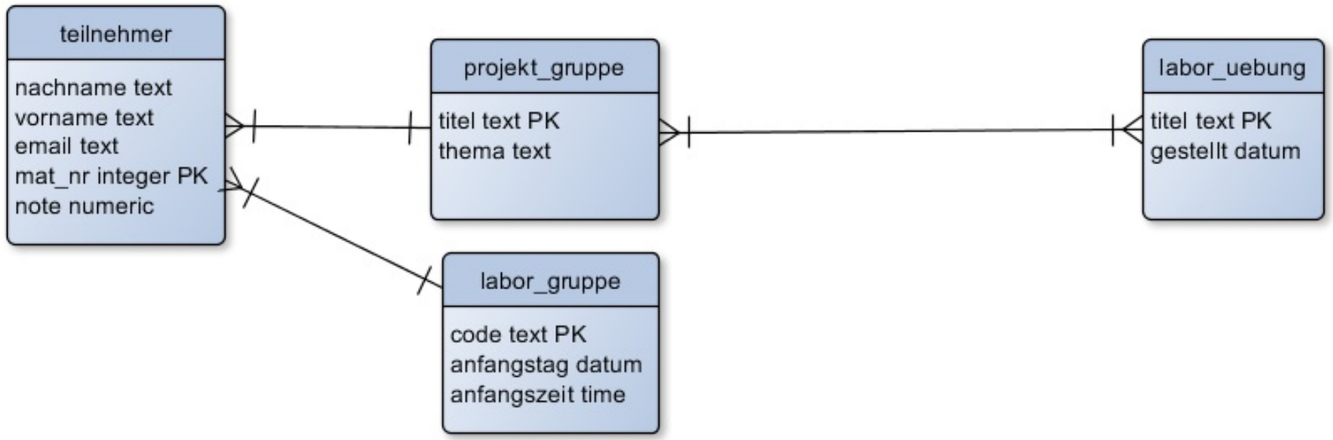
Wird hier `0..1 zu N`

oder

`1 zu N` umgesetzt?

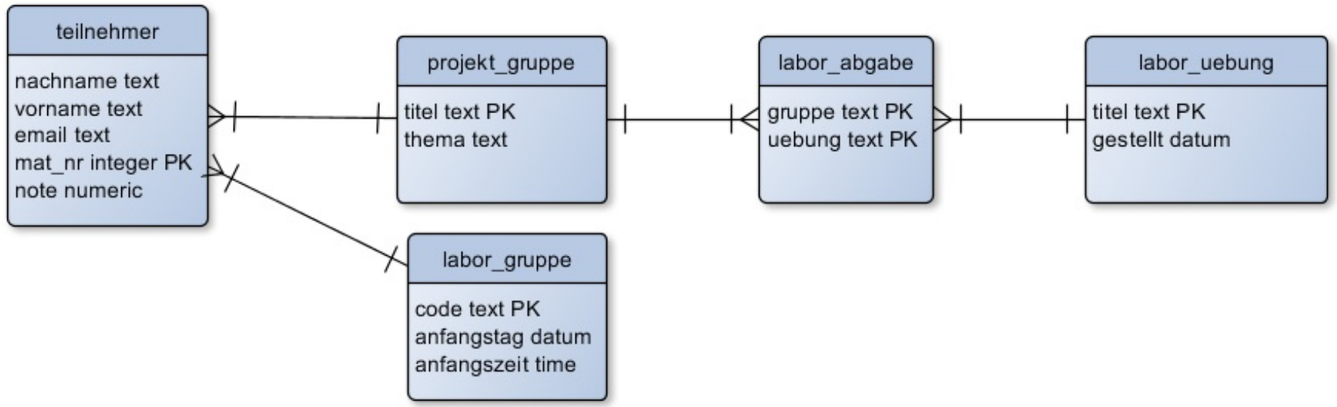
```
CREATE TABLE labor_gruppe (
  code text
    PRIMARY KEY
    CHECK ( code IN  ('w','x','y','z') ) ,
  -- Attribute fehlen
  anfangszeit time
    NOT NULL
);

CREATE TABLE teilnehmer (
  vorname text
    CHECK ( vorname != '' ),
  nachname text,
    CHECK ( vorname != '' ),
  matrikel_nr integer
    PRIMARY KEY,
  -- Attribute fehlen
  labor text
    REFERENCES labor_gruppe(code),
  geschlecht text
    REFERENCES geschlecht(name)
    ON UPDATE CASCADE
);
```

Foreign Keys (many to many)



Foreign Keys (many to many)

- Umsetzung durch neue "Beziehungs"-Relation



Foreign Keys (many to many)

- Neue Tabelle, die auf die beiden existierenden referenziert
  - Primary Key der neuen Tabelle ist Kombination der PKs der existierenden Tabellen

```
CREATE TABLE projekt_gruppe (
  titel text
    PRIMARY KEY
    CHECK ( titel != '' ) ,
  thema text
    NOT NULL
    DEFAULT ''
);

CREATE TABLE labor_uebung (
  titel text
    PRIMARY KEY,
  gestellt date
    NOT NULL
    UNIQUE
);

CREATE TABLE labor_abgabe (
  gruppe text
    REFERENCES projekt_gruppe(titel),
  uebung text
    REFERENCES labor_uebung(titel),

  PRIMARY KEY (gruppe,uebung)
);
```

Von Anfragen zu Abfragen

Anfragen

Z.b. Wieviel Teilnehmer sind in meinem Kurs?

Staerken relationaler Datenbanken:

1. Persistente, sichere und strukturierte Datenspeicherung
2. Effiziente **Abfragen** um Anfragen beantworten zu koennen!

Staerken relationaler Datenbanken:

1. Persistente, sichere und strukturierte Datenspeicherung
2. Effiziente **Abfragen** um Anfragen beantworten zu koennen!

- SQL hat nur einen Befehl dafuer:

Staerken relationaler Datenbanken:

1. Persistente, sichere und strukturierte Datenspeicherung
2. Effiziente **Abfragen** um Anfragen beantworten zu koennen!

- SQL hat nur einen Befehl dafuer:

SELECT

Anatomy von SELECT

```
SELECT *          -- welche Spalten sollen wie angezeigt werden
  FROM tabelle    -- Daten welcher Tabelle
 WHERE true       -- Selektionesbedingungen: nur Daten, die Kriterium entsprechen
```

Anatomy von SELECT

```
SELECT *         -- welche Spalten sollen wie angezeigt werden
  FROM tabelle   -- Daten welcher Tabelle
 WHERE true      -- Selektionesbedingungen: nur Daten, die Kriterium entsprechen
```

Kann gelesen werden als:

```
Zeige mir alle Spalten der Tabelle "tabelle" an und davon alle Zeilen.
```

Anatomy von SELECT

```
SELECT *         -- welche Spalten sollen wie angezeigt werden
  FROM tabelle   -- Daten welcher Tabelle
 WHERE true      -- Selektionesbedingungen: nur Daten, die Kriterium entsprechen
```

Kann gelesen werden als:

```
Zeige mir alle Spalten der Tabelle "tabelle" an und davon alle Zeilen.
```

Datenbank interpretiert das in der Reihenfolge `FROM`, `WHERE`, `'*'` (Spalten)

```
Hole aus der Tabelle "tabelle" alle Zeilen die der Bedingung 'true' entsprechen
und zeige davon alle Spalten an.
```

Konkretes SELECT

```
SELECT *         -- * (asterisk) heisst alle spalten, wie sie sind
  FROM "teilnehmer"; -- Daten der Tabelle mit dem Namen "order"
```

Boolsche `WHERE` Bedingung kann weggelassen werden, wenn man alle Zeilen will.

Sammeln von Anfragen

Transaktion

Eine Transkation ist eine Menge von Operationen die atomic, consistent, isoliert und dauerhaft sind (ACID).

ACID

- **A**lle Aenderungen mit
  Chirurgischer integritaet in einem
  Isoliertem Vorgang zusammengefasst und mit
  Dauerhaftem Ergebnis.

Die wichtigsten Eigenschaften im Einzelnen:

- Atomic

Eine Transaktion gruppiert mehrere Anweisungen in eine einzige atomare Operation in der "Alles oder Nichts” stattfindet.

- Consistent

Eine Transaktion bringt eine Datenbank von einem konsistenten Zustand in den nächsten.

- Isolated

Transaktion ist unabhängig, d.h. sie wird nicht durch konkurrierende Transaktionen beeinflusst.

- Durable

Bei der erfolgreichen Beendigung einer Transaktion sind alle Änderungen dauerhaft gespeichert.

Konsistenz bei sehr vielen gleichzeitigen Aenderungen

- Transaktionen garantieren, dass trotz gleichzeitiger Aenderungen von vielen verschiedenen Benuztern, alle Aenderungen konsistent sind.
- Tatsaechlich ist jede einzelne SQL-Anweisung bei den meisten Datenbank implizit eine einzelne Transaktion, d.h. eine Gruppe von Anweisungen mit nur einer Anweisung

Transaktion mit meheren Anweisungen

PostgreSQL Documentation fuer syntaktische Details und fortgeschrittene Eigenschaften

Ausgangslage: Datenbank ist in einem super gutem Zustand: Z1

```
-- Transaktion wird angefangen:
BEGIN;
SQL Anweisung 1;
SQL Anweisung 2;
...
SQL Anweisung N;
-- Bleibt offen bis:
COMMIT;
```

Wenn Transaktion **gut laeuft**, d.h. alle Anweisungen korrekt sind:

Datenbank ist in einem super gutem Zustand: Z2

Wenn Transaktion **fehlschlaegt**, d.h. nur eine der Anweisungen zum Fehler fuehrt:

Datenbank wird in den vorherigen Zustand zurueck gesetzt (rollback): Z1

PostgreSQL Documentation fuer syntaktische Details und fortgeschrittene

Gewollter Transaktionsabbruch

```
-- Transaktion wird angefangen:
BEGIN;
SQL Anweisung 1;
SQL Anweisung 2;
...
SQL Anweisung N;
-- Bleibt offen bis:
ROLLBACK;
```

Wenn Transaktion **fehlschlaegt** oder **erfolgreich** ist:

Datenbank wird in den vorherigen Zustand zurueck gesetzt (rollback): Z1

DDL Entwicklung

- PostgreSQL ist einer der wenigen DBMS, die auch DDL Anweisungen in Transaktionen ausfuehren kann

```
-- Transaktion wird angefangen:
BEGIN;
CREATE TABLE test (id integer PRIMARY KEY);
INSERT INTO test VALUES (1),(2),(3);
SELECT * from test;
UPDATE test SET id = id + 3;
SELECT * from test;
ROLLBACK;
```

- Sehr nuetzlich fuer iterative, evolutionaere Datenbankentwicklung

Danke für die Zusammenarbeit

Anhang

Abfragen ueber mehere Tabellen

Revisited: Anatomy von SELECT

```
SELECT *         -- welche Spalten sollen wie angezeigt werden
  FROM tabelle   -- Daten welcher Tabelle
 WHERE true      -- Selektionesbedingungen: nur Daten, die Kriterium entsprechen
```

Kann gelesen werden als:

```
Zeige mir alle Spalten der Tabelle "tabelle" an und davon alle Zeilen.
```

Revisited: Anatomy von SELECT

```
SELECT *         -- welche Spalten sollen wie angezeigt werden
  FROM tabelle   -- Daten welcher Tabelle
 WHERE true      -- Selektionesbedingungen: nur Daten, die Kriterium entsprechen
```

Kann gelesen werden als:

```
Zeige mir alle Spalten der Tabelle "tabelle" an und davon alle Zeilen.
```

Es wird immer eine und nur eine Tabelle durch `SELECT` erzeugt, daher ist das technisch praeziser:

```
Erzeuge und zeig mir eine virtuelle Tabelle, die folgender Anweisung enpricht:
Zeige alle Spalten der Tabelle "tabelle" an und davon alle Zeilen.
```

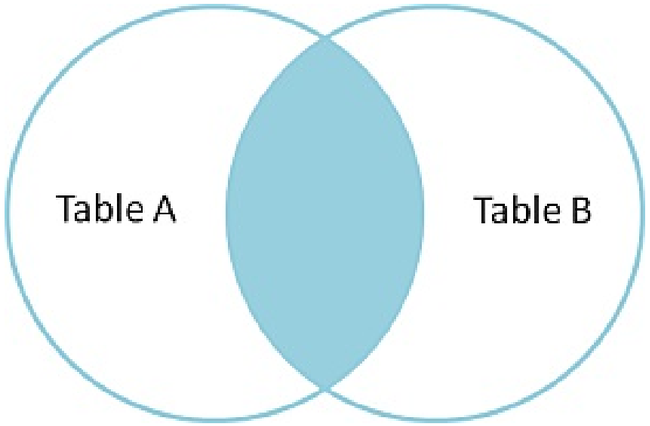
Beispiel-Tabellen

```
TabelleA      TabelleB
id name       id  name
-- ----       --  ----
1  Pirate     1   Rutabaga
2  Monkey     2   Pirate
3  Ninja      3   Darth Vader
4  Spaghetti  4   Ninja
```

INNER JOIN

```
SELECT *
  FROM TabelleA AS a
       INNER JOIN
       TabelleB as b
    ON a.name = b.name
```

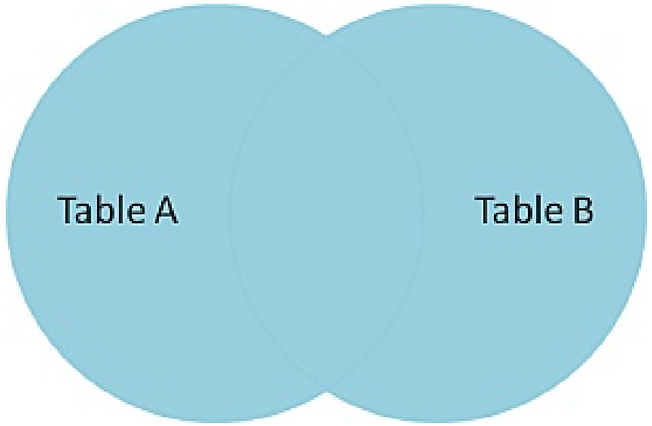
```
id  name       id   name
--  ----       --   ----
1   Pirate     2    Pirate
3   Ninja      4    Ninja
```



FULL OUTER JOIN

```
SELECT *
  FROM TabelleA as a
       FULL OUTER JOIN
       TabelleB as b
    ON a.name = b.name
```

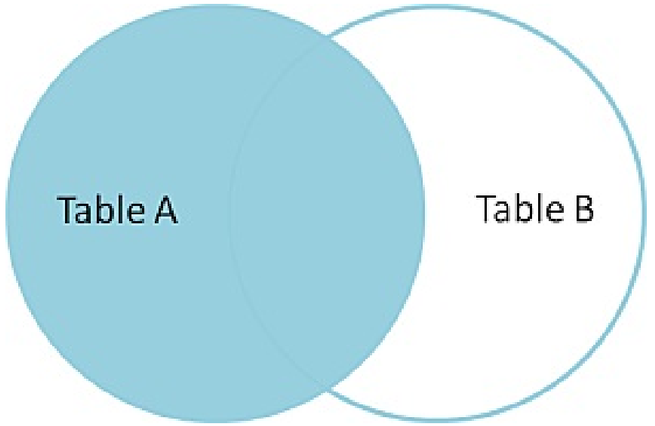
```
id    name       id    name
--    ----       --    ----
1     Pirate     2     Pirate
2     Monkey     null  null
3     Ninja      4     Ninja
4     Spaghetti  null  null
null  null       1     Rutabaga
null  null       3     Darth Vader
```



LEFT OUTER JOIN

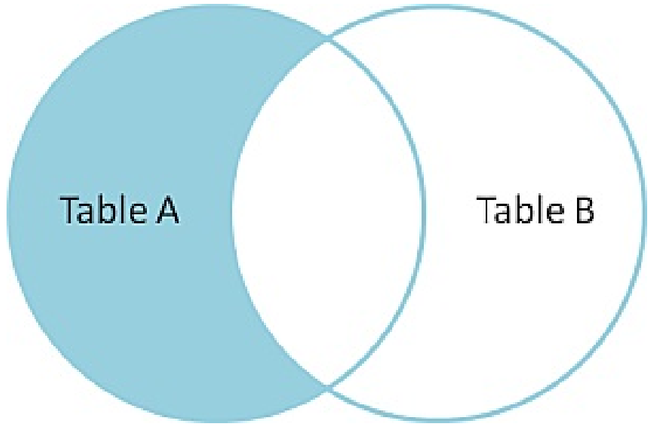
```
SELECT *
  FROM TabelleA AS a
       LEFT OUTER JOIN
       TabelleB AS b
    ON a.name = b.name
```

```
id  name       id    name
--  ----       --    ----
1   Pirate     2     Pirate
2   Monkey     null  null
3   Ninja      4     Ninja
4   Spaghetti  null  null
```



LEFT OUTER JOIN: nur Zeilen von TabelleA

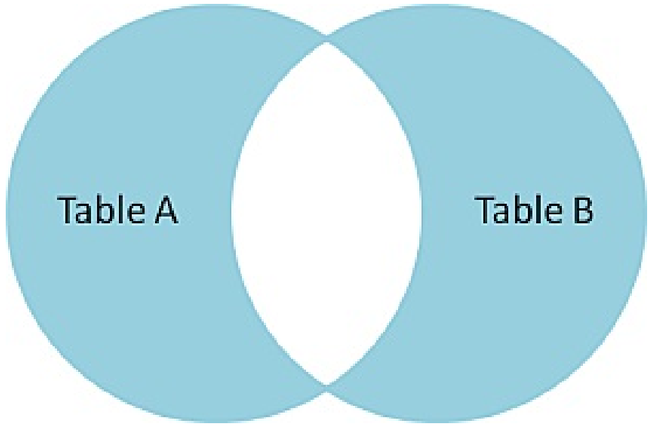
```
SELECT *
  FROM TabelleA AS a
       LEFT OUTER JOIN
       TabelleB AS b
    ON a.name = b.name
 WHERE b.id IS null
```




```
id  name       id    name
--  ----       --    ----
2   Monkey     null  null
4   Spaghetti  null  null
```

FULL OUTER JOIN: nur exklusive Zeilen

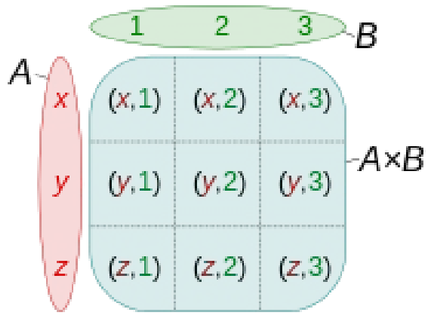
```
SELECT *
  FROM TabelleA AS a
       FULL OUTER JOIN
       TabelleB AS b
    ON a.name = b.name
 WHERE a.id IS null
       OR
       b.id IS null
```



id	name	id	name
2	Monkey	null	null
4	Spaghetti	null	null
null	null	1	Rutabaga
null	null	3	Darth Vader

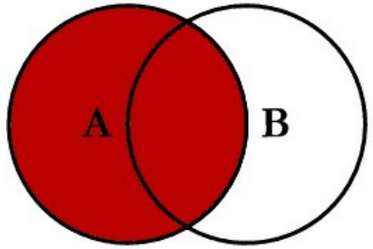
CROSS JOIN: Erzeugt kartesiches Produkt

```
SELECT *
  FROM TabelleA
       CROSS JOIN
       TabelleB
```

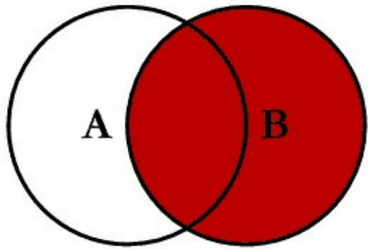


In einfachen Worten: Verbindet jede Zeile der TabelleA mit jeder Zeile der TabelleB

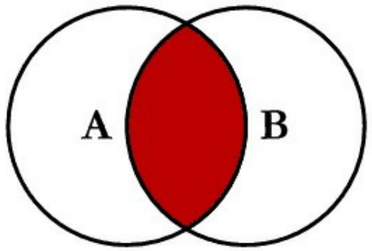
SQL JOINS



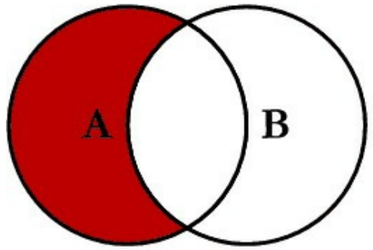

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



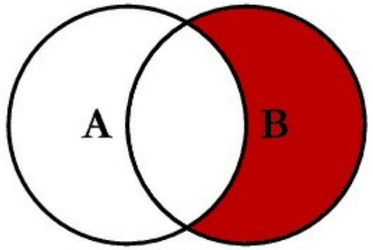

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



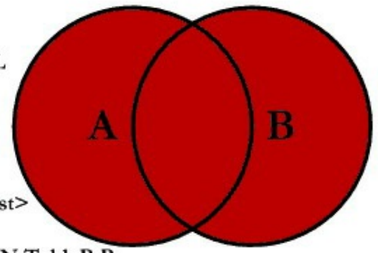

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



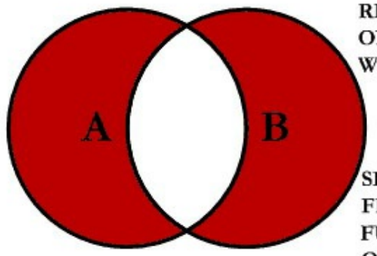

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Diskussion zu Visualisierung von SQL-joins

- Original Artikel mit Venn-Overview
- Diskussion und alternative Darstellung
- Kategorisierung nach JOIN-Typen

Normalisierung

Meine Vorgehensweise

- Alles was, gemaess den Anforderungen, eigenstaendig und unabhängig von exisitereneden Entitaeten gemanaged werden muss, ist eine neue eigenstaendige Entitaet.
  - Managen heisst: Alles was zu unterschiedlichen Zeiten erst ins System kommt und eigene insert, update, und delete Regeln hat.

Appendix

Mother Celko's Thirteen Normalization Heuristics

1. Does the table model either a set of one and only one kind of entity or one and only one relationship. This is what I call disallowing a “Automobiles, Squids and Lady GaGa” table. Following this rule will prevent ‘Multi-valued dependencies’ (MVD) problems and it is the basis for the other heuristics.

2. Does the entity table make sense? Can you express the idea of the table in a simple collective or plural noun? To be is to be something in particular; to be everything in general or nothing in particular is to be nothing at all (this is known as the Law of Identity in Greek logic). This is why EAV does not work – it is everything and anything.

3. Do you have all the attributes that describe the thing in the table? In each row? The most important leg on a three-legged stool is the leg that is missing.

1. Are all the columns scalar? Or is a column serving more than one purpose? Did you actually put hat size and shoe size in one column? Or store a CSV list in it?

2. Do not store computed values, such as (unit_price * order_qty). You can compute these things in VIEWs or computed columns.

3. Does the relationship table make sense? Can you express the idea of the table in a simple sentence, or even better, a name for the relationship? The relationship is “marriage” and not “person_person_legal_thing”

4. Did you check to see if the relationship is 1:1, 1:m or n:m? Does the relationship have attributes of its own? A marriage has a date and a license number that does not belong to either of the people involved. This is why we don't mind tables that model 1:1 relationships.

1. Does the entity or relationship have a natural key? If it does, then you absolutely have to model it as the PRIMARY KEY or a UNIQUE constraint. Is there a standard industry identifier for it? Let someone else do all that work for you.

2. If you have a lot of NULL-able columns, the table is probably not normalized.

3. The NULLs could be non-related entities or relationships.

4. Do the NULLs have one and only one meaning in each column?

5. If you have to change more than one row to update, insert or delete a simple fact, then the table is not normalized.

6. Did you confuse attributes, entities and values? Would you split the Personnel table into “Male_Personnel” and ”Female_Personnel” by splitting out the sex code? No, sex is an attribute and not an entity. Would you have a column for each shoe size? No, a shoe size is a value and not an attribute.