https://matse.paddel.xyz/spicker

Datenbanken

Patrick Gustav Blaneck, Tim Wende

Letzte Änderung: 26. Januar 2022

Inhaltsverzeichnis

1 Anmerkungen zum Spicker

Information: Genutzte Datenbank

Um in diesem Spicker möglichst konsistene Beispiele zu erstellen, die sich von denen in der Vorlesung unterscheiden, haben wir eine Datenbank basierend auf den Pokémon-Spielen bis zur achten Spielgeneration erstellt.

Die Datenbank enthält die folgenden Tabellen:

Name	Beschreibung
`pokemon`	enthält alle[a] Pokémon, inkl. Größe, Gewicht und Typen
`entwicklung`	enthält Informationen über alle möglichen Entwicklungen
`typ`	enthält alle Pokémon-Typen
`effektivitaet`	beschreibt die Effektivität zwischen allen möglichen Typkombinationen
`attacke`	enthält alle Attacken, inkl. Informationen zu Stärke, Genauigkeit und Typ
`item`	enthält alle Items
`lernt`	beschreibt, welches Pokémon welche Attacke wann und wie erlernen kann
`version`	enthält alle bisher erschienenen Hauptreihen-Pokémon-Spiele mit der jeweiligen Spielgeneration
`generation`	beschreibt, welche Region zu welcher Generation gehört[b]
`attacke_tm`	beschreibt, welches Item welche Attacke in welchem Pokémon-Spiel beibringen kann
`arenaleiter`	enthält alle Arenaleiter jeder Generation

Die meisten Beispiele beziehen sich auf diese Datenbank, wobei es aber auch vereinzelt Ausnahmen gibt.

[a] nicht beschrieben sind regionale Varianten
[b] die Region, die bezeichnend für die jeweilige Generation ist

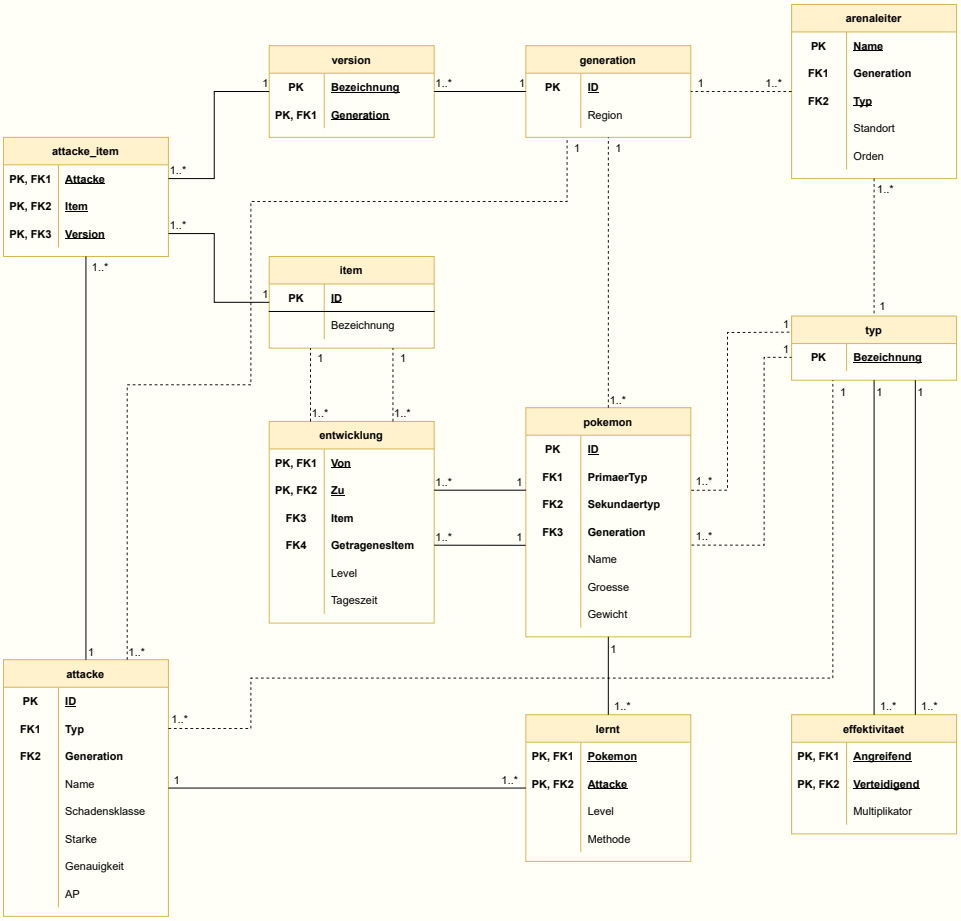
Information: Installation der Datenbank

Jede Abfrage in diesem Spicker wurde ausgeführt bzw. getestet.

Die Datenbank stellen wir zur Verfügung unter dem GitHub-Repository: matse-spicker-db

Dort findet sich auch eine Anleitung zur Installation.

Information: ER-Diagramm der Datenbank



2 Grundlagen

Definition: Anforderungen an Datenbanksysteme

- Widerspruchsfreie, dauerhafte, effiziente und schnelle Speicherung von Daten jeder Art
- Bedarfsgerechte und optimierte Bereitstellung von Daten
- Datensicherheit und Datenschutz
- Mehrbenutzungsbetrieb

Definition: Datenbanksystem

Ein *Datenbanksystem (DBS)* besteht aus dem Datenbankmanagementsystem (DBMS) und der Datenbank.

Definition: Datenbankmanagementsystem

Ein *Datenbankmanagementsystem (DBMS)* ist eine Software zur sicheren, konsistenten und persistenten Speicherung großer Datenmengen, mit dem Ziel mehreren Benutzern (gleichzeitig) effizienten, zuverlässigen, sicheren und bequemen Zugriff auf diese Daten zu ermöglichen.

Es besteht aus Komponenten zur Abfrage, Verwaltung und Administration der Daten bzw. Datenbank.

Bonus: DBMS Typen

Je nach Art der Daten und des Anwendungsfalls eignen sich unterschiedliche Typen von DBS, genauer DBMS, unterschiedlich gut. Da sind beispielsweise

- *relationale DBMS*, die Objekten gleicher Struktur (Attribute) in Tabellen verwalten.
  - meist genutzt
  - SQL als Abfragesprache
  - vergleichsweise einfache und effiziente Behandlung von Objekten gleichen Aufbaus in Tabellen
  - in der Praxis sind viele Daten strukturiert und typisiert
- *hierarchische oder objektorientierte DBMS*, die Objekte mit Eltern-Kind- oder Vererbungsbeziehungen besitzen und einzelne Strukturen gemeinsam haben.
- *dokumentenorientierte DBMS*, die Objekte ohne vergleichbare Strukturen verwalten.

Bonus: Datum

Daten sind zunächst Folgen von Zeichen (Zahlen, Buchstaben, Symbole) oder auch strukturierte Objekte. Sie folgen einer bestimmten Syntax.

Bonus: Information

Aus Daten entstehen *Informationen*, wenn die Bedeutung bekannt ist.

Durch die Hinzunahme von Semantik, ggf. weiteren Daten, kann das ursprüngliche Datum interpretiert bzw. verstanden werden.

Bonus: Wissen

Wissen entsteht durch systematische Verknüpfung von Daten, Informationen und (subjektiver) Erfahrung (Pragmatik).

Definition: Redundanz

Der Begriff der *Redundanz* beschreibt diejenigen Informationen oder Daten, die in einer Informationsquelle (hier: Datenbank) mehrfach vorhanden sind. Eine Informationseinheit ist dann redundant, wenn sie ohne Informationsverlust weggelassen werden kann.

Redundante Daten ermöglichen inkonsistente Daten (z.B. unterschiedliche Groß- und Kleinschreibung) und belegen Speicher.

Definition: Konsistenz

Ein Datenbankzustand wird als *konsistent* bezeichnet, wenn die gespeicherten Daten alle Anforderungen der Datenintegrität, die sich aus der Anwendung ergeben, erfüllen. Auf fehlerhafte Daten kann mit einer Fehlermeldung und dem Zurückweisen dieser Daten reagiert werden oder, falls möglich, auch mit einer automatischen Korrektur.

Das Gegenteil einer konsistenten Datenbasis ist eine inkonsistente.

Definition: Datenintegrität

Datenintegrität ist ein Begriff für die Qualität und Zuverlässigkeit von Daten eines Datenbanksystems. Im weiteren Sinne zählt zur Integrität auch der Schutz der Datenbank vor unberechtigtem Zugriff (Vertraulichkeit) und Veränderungen.

Daten spiegeln Sachverhalte der realen Welt wider. Logischerweise wird verlangt, dass sie dies korrekt tun. Ein DBMS soll Unterstützung bieten bei der Aufgabe, nur korrekte und widerspruchsfreie (konsistente) Daten in die Datenbank gelangen zu lassen. Außerdem wird mit korrekten Transaktionen die Konsistenz auch während des Systembetriebs aufrechterhalten.

Definition: Transaktion

Eine *Transaktion* ist ein Bündel von Aktionen, die in der Datenbank durchgeführt werden, um diese von einem inkonsistenten Zustand wieder in einen konsistenten (widerspruchsfreien) Zustand zu überführen.

Dazwischen sind die Daten zum Teil zwangsläufig inkonsistent.

Eine Transaktion ist atomar, d.h. nicht weiter zerlegbar. Innerhalb einer Transaktion werden entweder alle Aktionen oder keine durchgeführt. Nur ein Teil der Aktionen würde zu einem inkonsistenten Datenbankzustand führen.

Definition: Relation

Im Zusammenhang mit relationalen Datenbanken ist es üblich, eine *Relation* durch eine Tabelle zu beschreiben. In Tabellenform entsprechen die Attribute den Spaltenköpfen, die Attributwerte den in den Spalten vorhandenen Einträgen. Ein Tupel entspricht einer Zeile einer Tabelle.

Definition: Mehrbenutzungsbetrieb

Ein Datenbanksystem erlaubt mehreren Benutzern gleichzeitig den Zugriff auf eine Datenbank. Die Beantwortung unterschiedlicher Fragestellung (der diversen Benutzer) mit den gleichen (Basis-) Daten ist ein zentraler Aspekt eines Informationssystems.

Eine solche Mehrfachnutzung erhöht auch die Wirtschaftlichkeit eines Systems.

Definition: Datenansicht

Typischerweise hat eine Datenbank mehrere Benutzer, und jeder benötigt, je nach Zugangsrechten und Bedürfnissen, eine individuelle Ansicht der Daten (Inhalt und Form). Eine solche *Datensicht* kann aus einer Teilmenge der gespeicherten Daten oder aus daraus abgeleiteten (nicht explizit gespeicherten) Daten bestehen.

Definition: Persistenz

Persistenz meint, dass in einem DBMS einzelne Daten solange aufbewahrt werden müssen, bis sie explizit gelöscht werden.

Die Lebensdauer von Daten muss also von den Benutzern direkt oder indirekt bestimmbar sein und darf nicht von irgendwelchen Systemgegebenheiten abhängen. Ebensowenig dürfen einmal in die Datenbank aufgenommene Daten verloren gehen.

Änderungen, die eine Transaktion in einer Datenbank vornimmt, sind dauerhaft. Wenn die Transaktion abgeschlossen ist, kann auch ein darauf folgender Systemabsturz die Daten nicht mehr gefährden.

Definition: Komponenten eines DBMS

- *Data Manipulation Language (DML)*:
  - Überführt Anfrage in ausführbare Form, primär zur Manipulation von Daten
- *Data Definition Language (DDL) Compiler*:
  - Wie DML, nur für Datenstrukturen
- *Anfragebearbeitung*:
  - Erstellen eines Ablaufplans für eine Abfrage
  - inkl. logischer und physischer Optimierung
- *Datenbankmanager*:
  - Kern des DBMS, Ausführung der Anfragen
- *Schemaverwaltung*:
  - Verwaltung der Metadaten
  - „Typsystem“ des DBMS
- *Mehrbenutzungsynchronisation, Fehlerbehandlung*:
  - Transaktionsverwaltung

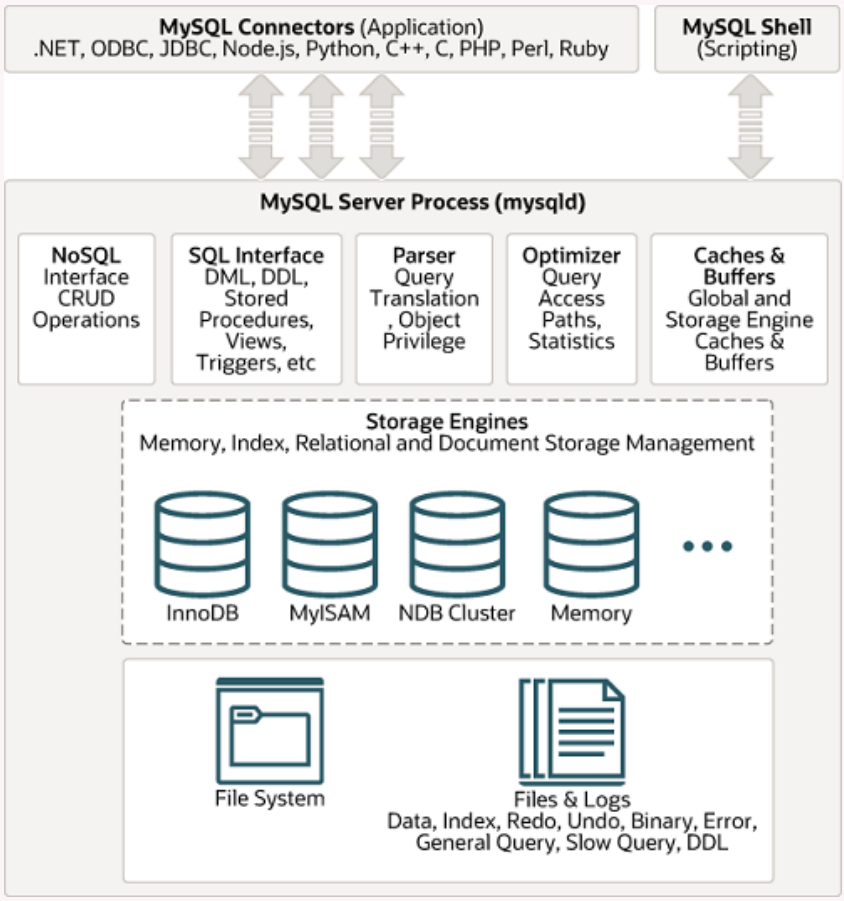
Definition: Storage Engine

Als *Storage Engine* eines Datenbankmanagementsystems wird jene Komponente bezeichnet, welche für das physische Abspeichern und Lesen der Daten zuständig ist.

Häufig stellen Datenbankmanagementsysteme mehrere austauschbare Storage Engines mit unterschiedlichen Eigenschaften zur Verfügung, z.B. transaktionssichere und nicht-transaktionssichere Storage Engines oder In-Memory Storage Engines. Das bringt den Vorteil, dass Applikationen von Fall zu Fall abgestimmt auf die Anforderungen der Anwendung die jeweiligen Stärken der einzelnen Storage Engines nutzen, dabei aber trotzdem das einheitliche Interface des Datenbankmanagementsystems verwenden können.

Beispiel: MySQL Architektur

Aus der MySQL-Dokumentation:



3 Modellierung

Definition: Modellierung

Modellierung übersetzt ein reales Problem (Szenario/Mini-Welt, z.B. die lesenswerten Artikel) in eine „bearbeitbare“ Version, das Modell, um final ein Ergebnis (Datenbankentwurf) systematisch zu entwickeln.

Ein Modell sollte

- vollständig,
- korrekt,
- minimal,
- verständlich und
- erweiterbar

sein.

Definition: ANSI/SPARC-Architektur

Die *ANSI-SPARC-Architektur* (auch Drei-Schema-Architektur, Drei-Ebenen-Architektur oder Drei-Ebenen-Schema-Architektur) beschreibt die grundlegende Trennung verschiedener Beschreibungsebenen für Datenbankschemata.

Die drei Ebenen sind:

- Die *externe Ebene*:
  - stellt den Benutzern und Anwendungen individuelle Benutzersichten bereit
  - z.B. Formulare, Masken-Layouts, Listen, Schnittstellen.
- Die *konzeptionelle Ebene*:
  - beschreibt, welche Daten in der Datenbank gespeichert sind und deren Beziehungen zueinander
  - Designziel ist vollständige und redundanzfreie Darstellung aller zu speichernden Informationen
  - hier findet Normalisierung des relationalen Datenbankschemas statt
- Die *interne Ebene* (auch physische Ebene):
  - stellt physische Sicht der Datenbank im Computer dar
  - beschreibt, wie und wo die Daten in der Datenbank gespeichert werden
  - Designziel ist hier effizienter Zugriff auf die gespeicherten Informationen
  - meistens nur durch eine bewusst in Kauf genommene Redundanz erreicht

Die Vorteile des Drei-Ebenen-Modells sind:

- *Physische Datenunabhängigkeit*:
  - interne Ebene ist von der konzeptionellen und externen Ebene getrennt
  - physische Änderungen, z. B. des Speichermediums oder des Datenbankprodukts, wirken sich nicht auf konzeptionelle oder externe Ebene aus
- *Logische Datenunabhängigkeit*:
  - konzeptionelle und die externe Ebene getrennt
  - bedeutet, dass Änderungen an der Datenbankstruktur (konzeptionelle Ebene) keine Auswirkungen auf externe Ebene, also die Masken-Layouts, Listen und Schnittstellen haben

3.1 Entity-Relationship-Modell

Definition: Entity-Relationship-Modell

Das *Entity-Relationship-Modell* bzw. *ER-Modell* dient dazu, im Rahmen der semantischen Datenmodellierung den in einem gegebenen Kontext (z. B. einem Projekt zur Erstellung eines Informationssystems) relevanten Ausschnitt der realen Welt zu bestimmen und darzustellen.

Grundlage der Entity-Relationship-Modelle ist die Typisierung von Objekten, ihrer Beziehungen untereinander und der über sie zu führenden Informationen („Attribute“).

3.1.1 Chen-Notation

Definition: Chen-Notation

Die *Chen-Notation* ist eine grafische Notation für Entity-Relationship-Modelle.

Angegeben in der grafischen Darstellung werden die:

- Entitätstypen bzw. Klassen
- Kardinalitäten
- Beziehungstypen

Im Folgenden beziehen wir uns bis auf weiteres auf die Chen-Notation.

Definition: Entität

Eine *Entität („Entity“) e* beschreibt ein individuell identifizierbares Objekt der Wirklichkeit. Sie ist damit vergleichbar mit einer Objektinstanz und entspricht einem einzelnen Datensatz in einer Datenbank, z.B. das Pokémon *Glumanda* oder die Attacke *Flammenwurf*.

Definition: Entitätstyp

Ein *Entitätstyp („Entity-Set“)* beschreibt eine Menge E von Entitäten $e \in E$ mit gleichen Eigenschaften (Klassifikation), z.B. „Pokémon“. Hier liegt eine Betrachtung als Menge von Objekten zugrunde.

Ein Entitätstyp ist vergleichbar mit einer Klassenbeschreibung und entspricht damit häufig einer Tabelle in einer relationalen Datenbank, hierbei sind die Spalten die Attribute.

Definition: Attribut

Eine *Attribut („Attribute“)* definiert, was über eine Entität (im Kontext) von Interesse ist. Eine Entität e ist durch die Werte ihrer Attribute charakterisiert und ein Entitätstyp E durch die Attribute selber.

Jedem Attribut A ist ein Datentyp T und damit implizit oder auch explizit ein Wertebereich D („Domain“) zugeordnet, der festlegt, welche Attributwerte zulässig sind[a], z.B. *Größe* und *Gewicht* des Pokémon Glumanda vom Typ `float`, die *Stärke* und *Genauigkeit* der Attacke Flammenwurf vom Typ `int`.

Besonders ist der Nullwert `NULL`. Das ist ein spezieller Attributwert, dessen Bedeutung variiert, d.h. z.B. ist der Wert unbekannt oder noch nicht festgelegt oder nicht möglich. `NULL` kann, muss aber nicht, im Wertebereich liegen.

Wir notieren $E[A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n]$ für einen Entitätstyp E mit den Eigenschaften $A_1, \ldots, A_n$. Wertebereiche bzw. Typen werden, wenn nicht relevant, auch ausgelassen.

$W(D)$ bezeichnet alle real existierenden Werte der Domain D in einer Datenbank.

[a] Wir schreiben: $A \in D$ oder $A : D$ bzw. $A : T$.

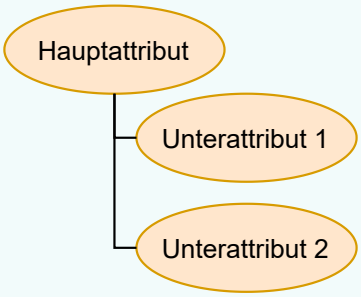
Definition: Zusammengesetztes Attribut

Zusammengesetzte Attribute haben selbst Attribute, z.B.

```
Typ: [Primärtyp: char(30), Sekundärtyp: char(30)]
```

Die Domain für ein zusammengesetztes Attribut A mit Unterattributen $A_1, \ldots, A_n$ ist dann gegeben mit:

$$A : [A_1 : D_2, \ldots, A_n : D_n] : D_1 \times \ldots \times D_n$$



Definition: Mehrwertiges Attribut

Ein *mehrwertiges Attribut* kann mehrere Ausprägungen bzw. Werte haben, z.B.


```
Attacke: {char(255)}
```

Die Domain für ein mehrwertiges Attribut ist dann gegeben mit

$$E : \{A\}$$

Definition: Schlüsselkandidat

Ein *Schlüsselkandidat*, oder kurz Schlüssel („key“), ist ein einwertiges Attribut oder eine Attributkombination, die jede Entität eindeutig identifiziert.

Es ist möglich, dass mehrere Schlüsselkandidaten existieren.

Der sogenannte *Primärschlüssel* ist ein ausgewählter Schlüsselkandidat. Seine Primärschlüssel-attribute werden im ER-Diagramm durch Unterstreichung gekennzeichnet.

Formal wird ein Primärschlüssel wie folgt dargestellt:

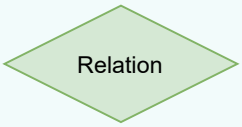
```
E: <[Attribute], {Primärschlüssel}>
```

z.B.:

```
Pokemon: <[ID, Name, Typ: [Primärtyp, Sekundärtyp], {Attacke: Name}], {ID}>
```

Definition: Relation

Eine *Relation („Relationship“)* erzeugt eine Verknüpfung bzw. einen Zusammenhang zwischen zwei oder mehreren Entitäten.



Eine Relationship-Menge R entspricht einer mathematischer Relation zwischen n Entity-Mengen E_i:

$$R \subseteq E_1 \times \ldots \times E_n \qquad (\text{häufig } n \in \{2,3\})$$

Eine Relation gibt an, ob eine Entität $e_1 \in E_1$ zu einer anderen Entität $e_2 \in E_2$ in Beziehung steht oder nicht, d.h.

$$(e_1, e_2) \in R \quad \text{bzw.} \quad (e_1, e_2) \notin R$$

z.B. gilt also:

$$\text{Pokémon Pikachu } \textit{lernt} \text{ die Attacke Donner} \iff (\text{Pikachu, Donner}) \in \text{lernt}$$

$$\text{Pokémon Pikachu } \textit{lernt nicht} \text{ die Attacke Glut} \iff (\text{Pikachu, Glut}) \notin \text{lernt}$$

Relationen können genau wie Entitäten Attribute besitzen.

Sei $A = [A_1 : D_1, \ldots, A_m : D_m]$ ein Tupel mit m Attributen $A = A_1, \ldots, A_m$ und $E_1, \ldots, E_n$ eine Menge von Entitätstypen. Dann ist

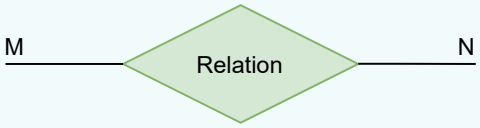
$$R \subseteq E_1 \times \ldots \times E_n \times D_1 \times \ldots \times D_m$$

eine mit A attributierte Relation.

Relationen können genauso gut auch reflexiv sein. Das kann im Diagramm mit Pfeilen verdeutlicht werden.

Definition: Kardinalität

Die *Kardinalität* gibt an, zu wie vielen Elementen ein Element in Beziehung steht.



Es gibt grundsätzlich folgende Abbildungsbeziehungen zwischen Entity-Mengen in der Chen-Notation:[a]

- *1:1* (One-to-One)
- *1:n* (One-to-Many) bzw. *n:1* (Many-to-One)
- *m:n* (Many-to-Many)

Die Kardinalität ergibt sich im Wesentlichen aus den Anforderungen und bestimmt ganz maßgeblich den Datenbankentwurf.

[a] In der Chen-Notation können 1, n und m auch 0 meinen!

Beispiel: 1:1 Relation (Chen-Notation)

In Pokémon verleiht im Laufe des Spiels jede *Arena* genau einen *Orden*.

Damit gilt:

Beispiel: 1:n Relation (Chen-Notation)

In Pokémon hat jede *Arena* mindestens einen, manchmal aber auch mehrere *Arenaleiter*.

Damit gilt:

Beispiel: n:1 Relation (Chen-Notation)

In Pokémon ist jede *Attacke* von genau einem *Typ*.

Natürlich können aber mehrere *Attacken* auch vom gleichen *Typ* sein.

Damit gilt:

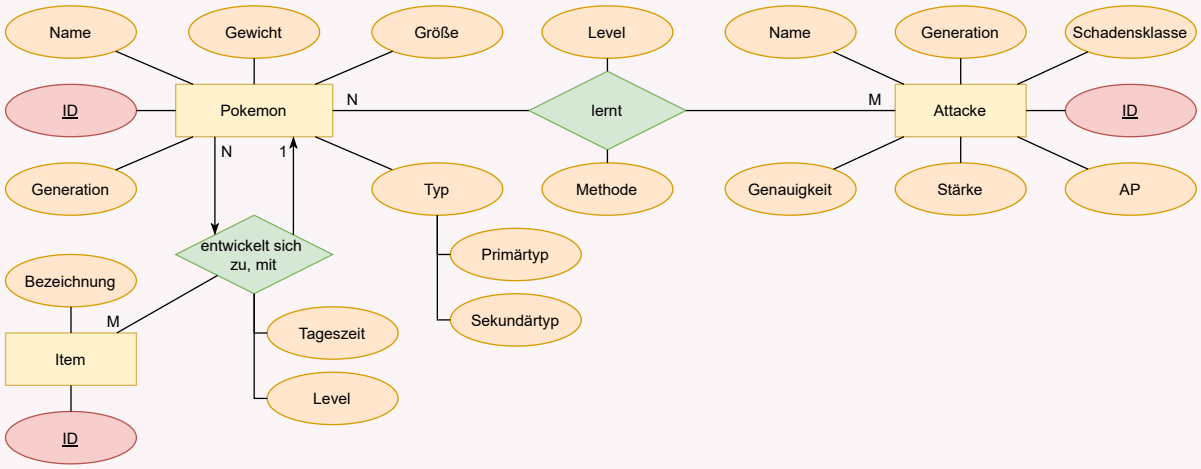
Beispiel: m:n Relation (Chen-Notation)

In Pokémon kann jedes *Pokémon* einige *Attacken* lernen.

Die selbe *Attacke* kann aber gleichzeitig von mehreren *Pokémon* erlernt werden.

Damit gilt:

Beispiel: Entity-Relationship-Modell (Chen-Notation)



Definition: Existenzabhängiger Entitätstyp

Die Grundidee von *existenzabhängigen Entitätstypen* ist, dass sie nur in Verbindung mit anderen Entitäten existieren[a].

Ein existenzabhängiger Entitätstyp hat keinen eigenen Schlüsselkandidaten, sondern wird über die Beziehung zur übergeordneten Entität identifiziert, z.B. über eine Kombination von Attributen.

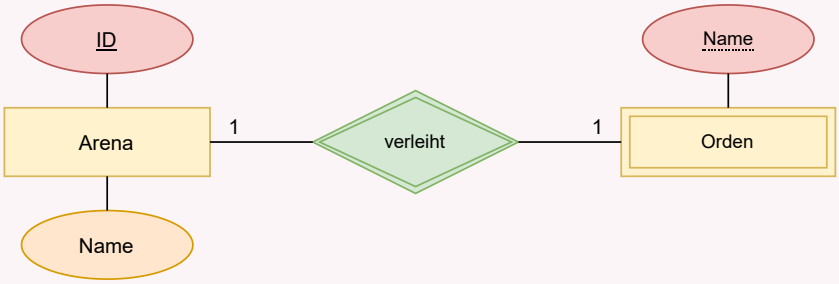
[a] vergleichbar mit der UML-Komposition

Beispiel: Existenzabhängiger Entitätstyp

In Pokémon verleiht im Laufe des Spiels jede *Arena* genau einen *Orden*.

Wir können das bisherige Beispiel insoweit verbessern, dass wir erkennen, dass ein spezieller Orden nicht ohne die jeweilige Arena existieren kann, bzw. wenn eine Arena aus der Datenbank gelöscht wird, der Orden ebenfalls gelöscht wird.

Damit gilt:

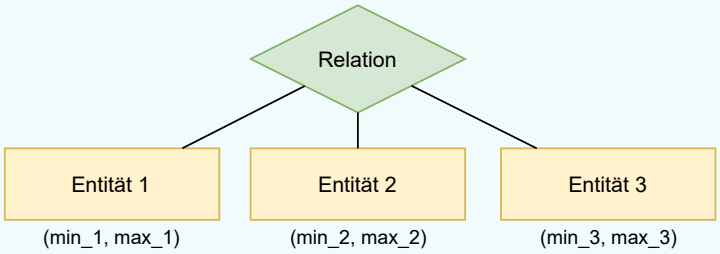


3.1.2 Min-Max-Notation

Definition: Min-Max-Notation

Ein Nachteil der Chen-Notation ist, dass die Angaben der Kardinalität sehr ungenau sind.

Als Verbesserung wird die *Min-Max-Notation* vorgeschlagen. In der Min-Max-Notation wird angegeben, wieviele Relationen minimal bzw. maximal möglich sein sollen.



Für jeden an einer Relation beteiligten Entitätstypen E_k wird angegeben, wie oft eine Entität minimal und maximal in der Relation enthalten sein darf.[a]

Sei $R \subseteq E_1 \times \ldots \times E_n$ und die Projektion auf das k-te Element bzw. die Anzahl eines Wertes an k-ter Stelle definiert durch

$$\text{proj}_k(e_1, \ldots, e_k, \ldots, e_n) = e_k$$

$$\text{count}_k(e) = |\{r \in R \mid \text{proj}_k(r) = e\}|$$

dann muss für alle k gelten:

$$\forall e_k \in E_k : \min_k \leq \text{count}_k(e_k) \leq \max_k$$

[a] Die Min-Max-Notation unterscheidet sich grundlegend von den anderen Notationsformen im Hinblick auf die Bestimmung der Kardinalität und die Position, an der die Häufigkeitsangabe im ER-Diagramm vorgenommen wird.

Bei allen anderen Notationen wird die Kardinalität eines Beziehungstyps dadurch bestimmt, dass für eine Entität des einen Entitätstyps nach der Anzahl der möglichen beteiligten Entitäten des anderen Entitätstyps gefragt wird.

Bei der Min-Max-Notation hingegen ist die Kardinalität anders definiert. Hierbei wird für jeden der an einem Beziehungstyp beteiligten Entitätstyp nach der kleinst- und größtmöglichen Anzahl der Beziehungen gefragt, an denen eine Entität des jeweiligen Entitätstyps beteiligt ist.

Das jeweilige Min-Max-Ergebnis wird bei dem Entitätstyp notiert, für den die Frage gestellt wurde.

Beispiel: 1:1 Relation (Min-Max-Notation)

In Pokémon verleiht im Laufe des Spiels jede *Arena* genau einen *Orden*.

Damit gilt:

Beispiel: 1:n Relation (Min-Max-Notation)

In Pokémon hat jede *Arena* mindestens einen, manchmal aber auch mehrere *Arenaleiter*.

Damit gilt:

Beispiel: n:1 Relation (Min-Max-Notation)

In Pokémon ist jede *Attacke* von genau einem *Typ*.

Natürlich können aber mehrere *Attacken* auch vom gleichen *Typ* sein.

Damit gilt:

Beispiel: m:n Relation (Min-Max-Notation)

In Pokémon kann jedes *Pokémon* einige *Attacken* lernen.

Die selbe *Attacke* kann aber gleichzeitig von mehreren *Pokémon* erlernt werden.

Damit gilt:

3.1.3 UML-Notation

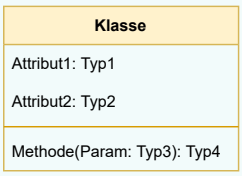
Definition: UML-Notation

Die *UML-Notation* ist eine Notation zur objektorientierten Modellierung.

Bei den Kardinalitäten wird sich bezüglich der Reihenfolge bei der Chen-, bezüglich der Angaben bei der Min-Max-Notation bedient.

Definition: Entität (UML-Notation)

Eine *Entität* wird in der UML-Notation als Klasse dargestellt. Diese Klasse ist ein Rechteck mit drei Sektionen für



- den Klassennamen,
- Attribute (optional) und
- Methoden (optional).

Hierbei werden häufig nur die relevanten Details gezeigt.

Die Notation ist wie folgt:

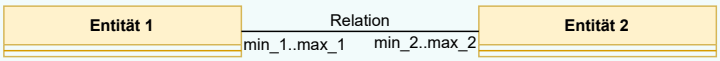
```
Sichtbarkeit: Name(Parameter): Rückgabetyp Eigenschaften
```

Dabei gilt:

- Sichtbarkeiten:
  - +: public
  - -: private
  - #: protected
- Unterstrichene Attribute/Operationen stehen für Klassenattribute/Operationen
- Eigenschaften können Bedingungen (constraints) sein
- Schlüsselattribute z.B. als Kommentar / Eigenschaft

Definition: Relation (UML-Notation)

Für jeden an einer Relation beteiligten Entitätstyp E_k wird angegeben, wie oft eine Entität minimal und maximal in E_k enthalten sein darf, wenn der andere Eintrag ($n = 2$) bzw. die anderen Einträge ($n > 2$) fest gewählt ist, bzw. sind.



Wir haben hier die gleiche Sichtweise wie bei der Chen-Notation, nur mit Angabe der minimalen und maximalen Elemente in der Entitätenmenge.

Dadurch ist eine Angabe der Fälle

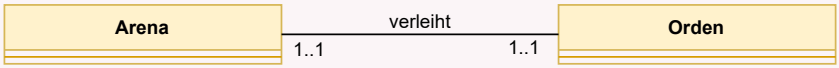
- *optional*, wenn Kardinalität $= 0$, bzw.
- *verpflichtend*, sonst.

Attributierte Relationen werden in der UML-Notation in einer assoziierten Klasse zusammengefasst.

Beispiel: 1:1 Relation (UML-Notation)

In Pokémon verleiht im Laufe des Spiels jede *Arena* genau einen *Orden*.

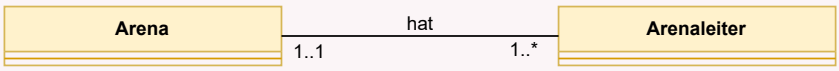
Damit gilt:



Beispiel: 1:n Relation (UML-Notation)

In Pokémon hat jede *Arena* mindestens einen, manchmal aber auch mehrere *Arenaleiter*.

Damit gilt:

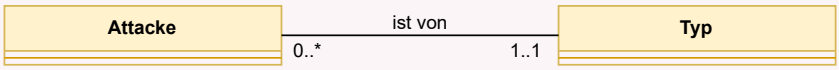


Beispiel: n:1 Relation (UML-Notation)

In Pokémon ist jede *Attacke* von genau einem *Typ*.

Natürlich können aber mehrere *Attacken* auch vom gleichen *Typ* sein.

Damit gilt:

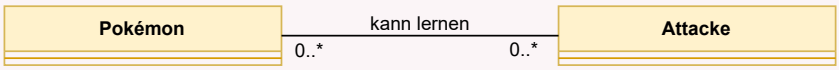


Beispiel: m:n Relation (UML-Notation)

In Pokémon kann jedes *Pokémon* einige *Attacken* lernen.

Die selbe *Attacke* kann aber gleichzeitig von mehreren *Pokémon* erlernt werden.

Damit gilt:

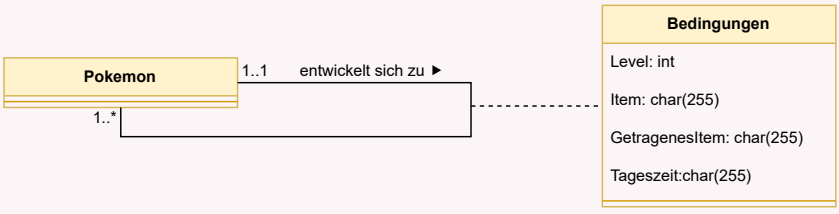


Beispiel: UML-Relation mit Attributen

In Pokémon kann sich ein *Pokémon* in teils mehrere *Pokémon* entwickeln.

Dabei sind verschiedene *Bedingungen* zu erfüllen um diese Entwicklung zu erreichen.

Damit gilt:



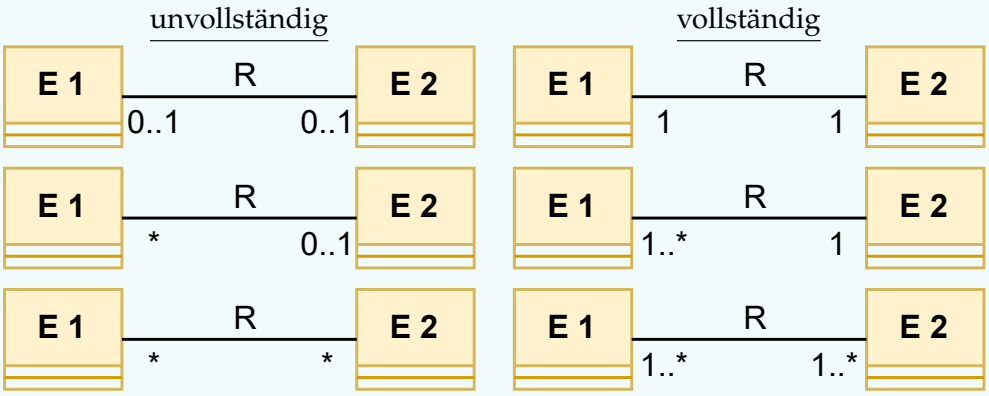
Definition: Vollständige und unvollständige UML-Relation

Eine UML-Relation heißt *vollständig*, wenn alle Entitäten teilnehmen, *unvollständig* sonst.

In diesem Kontext werden oft Kurzformen genutzt:

- 0..* → *
- 1..1 → 1
- 1 als Default

Damit gilt:



3.2 Generalisierung, Spezialisierung

Definition: Generalisierungstyp

Der *Generalisierungstyp* enthält die Attribute (Spalten), die alle spezialisierten Entitäten gemeinsam haben.

Jede Entitätsmenge einer Generalisation verlangt eine eigenständige Tabelle, wobei der Primärschlüssel der übergeordneten Tabelle auch Primärschlüssel der untergeordneten Tabelle wird.

Definition: Spezialisierungstyp

Die *Spezialisierungstypen* enthalten die speziell für sie zutreffenden Attribute, wobei

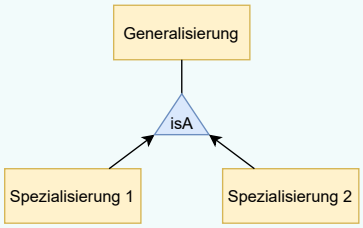
- die Spezialisierung disjunkt oder überlappend sein kann und
- die Eigenschaften der Generalisierung erbt.

In einer spezialisierten Tabelle existieren keine Tupel, welche in der generalisierten Tabelle nicht vorkommen. Eine Spezialisierung wird durch Vererbung realisiert.

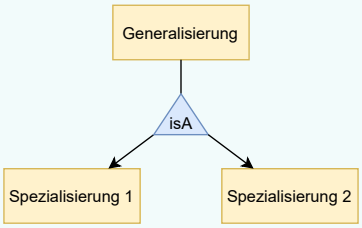
Man beachte bei den Diagrammen:

- zeigt der Pfeil der Spezialisierung in Richtung *der Generalisierung*, so ist sie *nicht disjunkt*.
  Eselsbrücke: Pfeile treffen sich → es existiert Schnittmenge → nicht disjunkt.
- zeigt der Pfeil der Spezialisierung in Richtung *der Spezialisierung*, so ist sie *disjunkt*.
  Eselsbrücke: Pfeile treffen sich nie → es existiert keine Schnittmenge → disjunkt.

nicht disjunkt



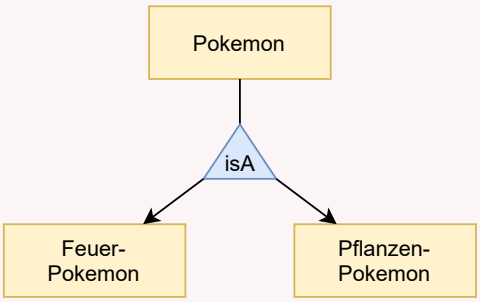
disjunkt

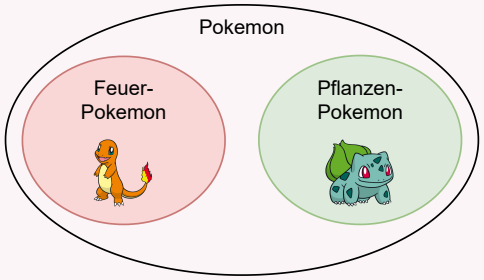


Beispiel: Spezialisierung

In Pokémon existieren *keine* Pokémon mit der Typkombination Feuer und Pflanze.

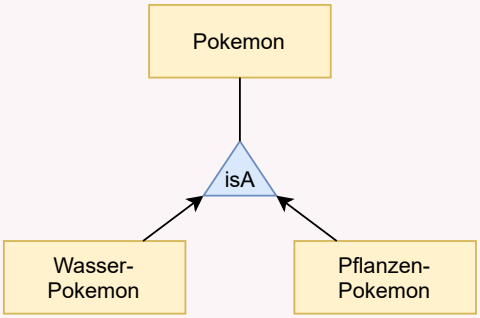
Damit gilt:

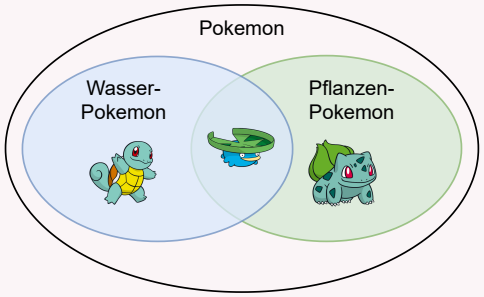




Allerdings existieren Pokémon mit der Typkombination Wasser und Pflanze.

Damit gilt:





3.3 Umsetzung von Relationen in RDBMS

Bonus: 1:1 Relation (RDBMS)

Eine Möglichkeit der Umsetzung ist über eine Fremdschlüsselbeziehung in einer der beiden Tabellen.

Eine andere wäre über eine eigene Relationen-Tabelle. Hier darf aber dann jede Entität nur einmal vorkommen.

Bonus: n:1 Relation (RDBMS)

Eine Möglichkeit der Umsetzung sind zwei Tabellen für die Entitätstypen und die Realisierung der Relation als Fremdschlüssel in der Tabelle mit der n-Kardinalität.

Eine andere Möglichkeit der Umsetzung sind zwei Tabellen für die Entitätstypen und eine eigene Tabelle für die Realisierung der Relation mit zwei Fremdschlüsseln.

Bonus: m:n Relation (RDBMS)

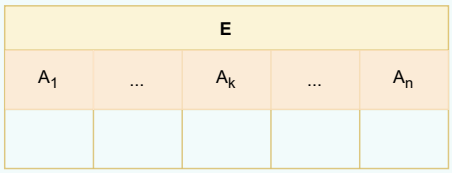
Die einzige Möglichkeit der Umsetzung sind zwei Tabellen für die Entitätstypen und eine eigene Tabelle für die Realisierung der Relation mit zwei Fremdschlüsseln.

3.4 Relationales Modell

Definition: Entitätstyp (Relationales Modell)

Ein wie bisher definierter Entitätstyp E mit

$$E : \{A_1 : D_1, \ldots, A_n : D_n\}$$



wird im relationalen Modell zu einer *Tabelle* E mit:

- *atomaren*[a] Attributen A_k als *Spalten* mit Datentyp D_k
- Schlüsselattributen als *primary keys*

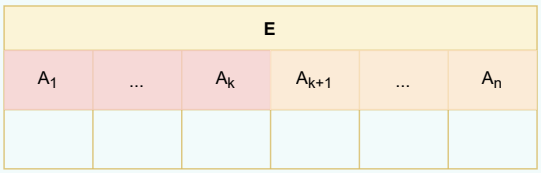
Die Reihenfolge der Zeilen und Spalten ist nicht relevant.

Enthaltene Informationen werden nur durch Datenwerte ausgedrückt.

[a] weder zusammengesetzt noch mehrwertig

Definition: Schlüsselattribut (Relationales Modell)

Jede Zeile bzw. Entität in der Tabelle E beschreibt ein eindeutiges Objekt.



Technisch sind Tabellen ohne Schlüsselattribut und Einträge, die in allen Werten identisch sind, denkbar, aber erst die Verwendung eindeutiger *Schlüsselattribute* garantiert die Identifikation unterschiedlicher Objekte, selbst bei gleichen Werten der restlichen Attribute.

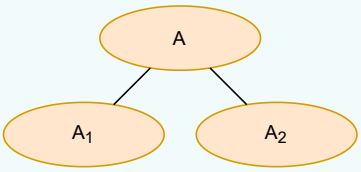
Schlüsselattribute werden im relationalen Modell zu *primary keys*.

Definition: Zusammengesetztes Attribut (Relationales Modell)

Bei einem zusammengesetzten Attribut A mit

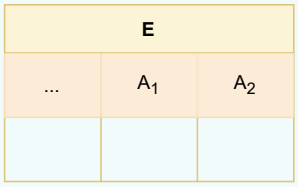
$$A : [A_1 : D_1, \ldots, A_n : D_n] : D_1 \times \ldots \times D_n$$

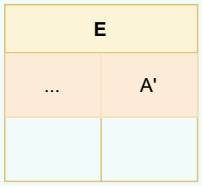
liegen die Informationen in den A_k, d.h. A an sich enthält selbst keine Informationen.



Deswegen kann das Modell in eine der zwei Möglichkeiten mit nur atomaren Attributen überführt werden:

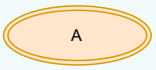
1. „Umhängen“ der Attribute A_k direkt an den Entitätstypen
2. Zusammenfassen der A_k zu einem neuen gemeinsamen Attribut A' (z.B. durch Stringkonkatenation)





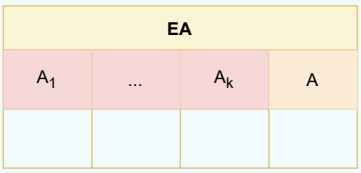
Definition: Mehrwertige Attribute (Relationales Modell)

Bei einem mehrwertigen Attribut A werden die verschiedenen Ausprägungen von A üblicher-weise in eine $1 : n$-Relation mit einem weiteren Entitätstyp E_A abgebildet.



Eine allgemeinere Rolle von E_A und eine entsprechende $m : n$-Relation ist auch denkbar.

Schlüsselattribute von E können so auch zu Schlüsselattributen von E_A werden. Das hängt von der genauen Umsetzung der Relation ab.

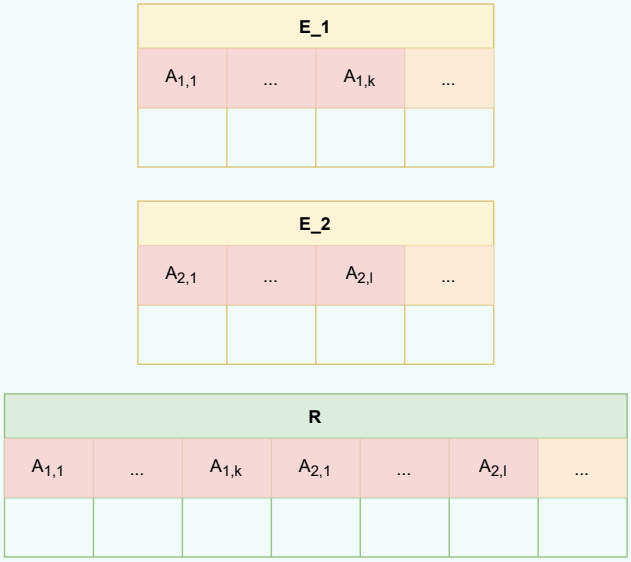


Definition: m:n Relation (Relationales Modell)

$n : m$-Relationen werden im relationalen Modell immer in einer eigenen Tabelle R realisiert.

Diese Tabelle R enthält alle Schlüsselattribute der E_k, die vereint Schlüsselattribute von R sind („Fremdschlüssel“). Es ist aber auch möglich, ein künstliches Schlüsselattribut (meist `ID`) zu verwenden.

Sollte R attributiert sein, dann finden sich diese Attribute ebenfalls in R.



Definition: 1:n Relation (Relationales Modell)

Da wir bei einer $1 : n$-Relation eine partielle Funktion

$$f : E_2 \to E_1$$

haben, kann das Schlüsselattribut von E_1 als „Fremdschlüssel“ in E_2 als weiteres Attribut eingebettet werden. Dort ist es kein Schlüssel von E_2, es referenziert nur die assoziierte Entität in E_1.[a]

Sollte E_1 mehrere Schlüsselattribute haben, die den Primärschlüssel bilden, müssen selbstverständlich alle in E_2 vorkommen.

[a] Der Fremdschlüssel kann nur in E_2, da man umgekehrt ein mehrwertiges Attribut in E_1 führen müsste, was im Widerspruch zur Atomarität stünde.

Definition: 1:1 Relation (Relationales Modell)

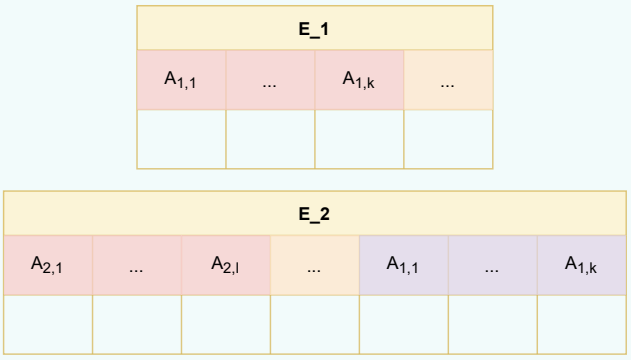
Bei einer $1:1$-Relation R hat man jede der bisher benannten Möglichkeiten:

1. je eine Tabelle E_1 bzw. E_2 mit
   a) Fremdschlüssel in E_1
   b) Fremdschlüssel in E_2
   c) Fremdschlüssel in beiden Tabellen
2. eine Tabelle E' mit allen Attributen (Spalten) aus E_1 und E_2
3. eine Tabelle für R (analog zu $m:n$-Relation)

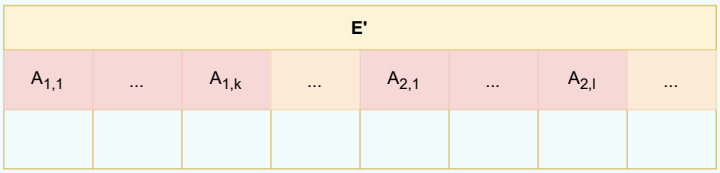
Welche der Möglichkeiten am geeignetsten ist, ist am Ende von den Anforderungen und der Verwendung der Relation in der Praxis abhängig. Diese sind jeweils:

1. Daten aus E_1 und E_2 häufig gemeinsam benötigt und man sucht ein e_2 zu einem e_1, oder umgekehrt
2. hebt zwar Trennung der modellierten Aspekte evtl. auf, kann aber sehr effizient sein
3. zwar komplexer, aber stabil gegenüber Änderungen zu einer $1:n$- bzw. $m:n$-Relation

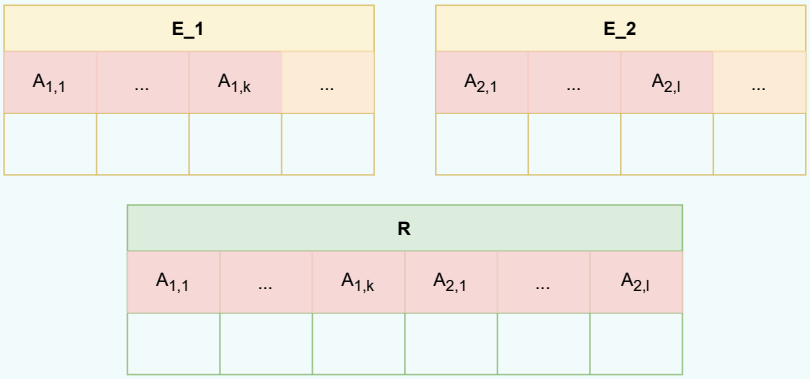
Variante 1



Variante 2



Variante 3



4 Relationale Algebra

Definition: Relationale Algebra

Eine *relationale Algebra* definiert (Mengen-)Operationen, die sich auf eine Menge von Relationen anwenden lassen. Damit können Relationen beispielsweise gefiltert, verknüpft oder aggregiert werden. Die Ergebnisse aller Operationen sind ebenfalls Relationen. Aus diesem Grund bezeichnet man die Relationenalgebra als abgeschlossen.

Eine Relation

$$R \subseteq D_1 \times \ldots \times D_n$$

ist wie bisher primär eine konkrete Tupelmenge

$$t = [a_1, \ldots, a_n] \in R, a_k \in D_k$$

Eine Tabelle ist eine visuelle Repräsentation einer Relation und eine Zeile in einer Tabelle repräsentiert ein Tupel.

Steht die Relation für einen Entitätstyp, so sind die Tupel konkrete Entitäten.

Bonus: Funktionen zum Aufbau einer Relation

Ist man statt an einer Relation R, also der konkreten Tupelmenge, nur am Aufbau der Relation, d.h. den Attributen und Datentypen, interessiert, helfen diese Funktionen:

- $\text{ident}(R) = \{a_i\}$
- $\text{schema}(R) = \{[a_1 : T_1, \ldots, a_n : T_n]\}$

Definition: Schlüsselkandidat (Relationale Algebra)

Ein *Schlüsselkandidat* K mit $K \subseteq \text{ident}(R)$ ist eine Menge von Attributen, deren Werte jeweils alle Tupel der Relation eindeutig identifizieren („identifier“).

So ist implizit garantiert, dass es keine zwei gleichen Tupel in der Relation gibt.

Grundsätzlich kann es mehrere Schlüsselkandidaten geben, aus denen dann der Schlüssel („primary key“) ausgewählt wird.

4.1 Mengenoperationen

Definition: Voraussetzungen für Mengenoperationen

Um Mengenoperationen auf Relationen durchzuführen, müssen diese kompatibel sein. Das bedeutet, Attributanzahl und Wertebereiche müssen übereinstimmen (*Vereinigungsverträglichkeit* bzw. *Typkompatibilität*).

Für zwei Relationen S und T muss also gelten, dass[a]

$$\text{schema}(S) = \text{schema}(T)$$

[a] Die Attribute müssen allerdings nicht formal gleich heißen, eine verträgliche Semantik ist aber sinnvoll.

Definition: Klassische Mengenoperationen

Seien S und T zwei kompatible Relationen. Dann sind definiert:

- *Vereinigung*:
$$S \cup T = \{r \mid r \in S \vee r \in T\}$$
- *Differenz*:
$$S - T = S \setminus T = \{r \mid r \in S \wedge r \notin T\}$$
- *Schnittmenge*:
$$S \cap T = \{r \mid r \in S \wedge r \in T\}$$

Definition: Kartesisches Produkt

Für zwei Relationen S und T mit

$$\text{ident}(S) \cap \text{ident}(T) = \emptyset$$

ist das *kartesische Produkt* $S \times T$ definiert durch

$$S \times T = \bigcup_{(s_1,\ldots,s_n)\in S} \left[\bigcup_{(t_1,\ldots,t_k)\in T} \{(s_1,\ldots,s_n,t_1,\ldots,t_k)\} \right]$$

Im üblichen kartesisches Produkt entstehen Paare (s,t) mit $s \in S$ und $t \in T$. Hier hingegen entstehen $|S \times T| = |S| \cdot |T|$ Tupel, bestehend aus allen Attributen einer Entität $s \in S$ verbunden mit den Attributen einer Entität $t \in T$.

Im Fall $\text{ident}(S) \cap \text{ident}(T) \neq \emptyset$ würde obige Definition doppelte Attribute erzeugen, was mathematisch ein Problem ist, aber in der Praxis durch Umbenennung der Attribute gelöst werden kann.

Eselsbrücke: Jedes Element von S mit jedem von T.

Definition: Selektion

Für eine Relation S und eine logische Bedingung Θ ist die *Selektion* definiert durch

$$\sigma_\Theta(S) = \{s \in S \mid s \text{ erfüllt } \Theta\}$$

Selektionsbedingungen sind häufig Vergleichsoperationen ($=, \neq, \leq, <, >, \geq$) auf den Attributen und logische Verknüpfungen ($\wedge, \vee, \neg$).

Die Selektion wirkt wie ein Filter auf der Relation S, wo sie auf jedes Element angewandt wird.

Definition: Projektion

Die *Projektion* $\pi_{a_{i1},\ldots,a_{ik}}$ wählt aus einer Relation S mit $\text{ident}(S) = \{a_1,\ldots,a_n\}$ die Attribute a_{i1} bis a_{ik} aus:

$$\pi_{a_{i1},\ldots,a_{ik}}(S) = \{(a_{i1},\ldots,a_{ik}) \mid a \in S\}$$

Die mathematische Menge eliminiert Duplikate, die Ergebnismenge in SQL aber nicht.

Definition: Theta-Join

Der *Theta-Join* bzw. *Theta-Verbund* $S \bowtie_\Theta T$ für zwei Relationen S und T und Selektionsbedingung Θ ist definiert durch

$$S \bowtie_\Theta T = \sigma_\Theta(S \times T)$$

Hier entstehen zunächst enorm große Datenmengen durch das kartesische Produkt, die dann durch die Selektion wieder reduziert werden. In der Praxis kann ein DBMS diese sehr häufigen Join-Operationen effizient durchführen.

Bonus: Equi-Join

Der *Equi-Join* ist ein Spezialfall des Theta-Join mit der Bedingung, dass der Inhalt bestimmter Attribute, z.B. a_1 und a_2, identisch sein muss, d.h. der speziellen Form $a_1 = a_2$ genügt:

$$S \bowtie_{a_1=a_2} T = \sigma_{a_1=a_2}(S \times T)$$

Definition: Natural-Join

Ausgehend von der Idee, dass Primär- und Fremdschlüssel gleich heißen, nutzt der *Natural-Join* dies aus und verbindet Entitäten, die in gleich benannten Attributen gleiche Werte besitzen[a].

Im Unterschied zum Theta-Verbund enthält der Natural-Join die gleichen Attribute nur einmal, eliminiert so also ungewünschte Redundanz.

Für zwei Relationen S und T mit $\text{schema}(S) = \{[a_1, \ldots, a_n, b_1, \ldots, b_k]\}$ und $\text{schema}(T) = \{[b_1, \ldots, b_k, c_1, \ldots, c_m]\}$ ist der Natural-Join $S \bowtie T$ definiert durch[b]

$$S \bowtie T = \pi_{a_1,\ldots,a_n,S.b_1,\ldots,S.b_k,c_1,\ldots,c_m} \sigma_{S.b_1=T.b_1 \wedge \ldots \wedge S.b_n=T.b_n}(S \times T)$$

Hinweis:

Von der Verwendung von Natural-Joins wird abgeraten! Beim Natural-Join werden immer alle gleichnamigen Attribute verwendet, d.h. Hinzufügen neuer Attribute oder Umbenennen vorhandener Attribute führt schnell zu einer Änderung der Abfrage.

[a] Also ein Equi-Join auf gleichen Attributen.

[b] Das entspricht dem Schema $\{[a_1, \ldots, a_n, b_1, \ldots, b_k, c_1, \ldots, c_m]\}$ und man sieht, dass die doppelten Attribute $S.b_i$ bzw. $T.b_i$ durch die Projektion π wegfallen.

Bonus: Semi-Join

Wenn einen nur die Existenz, nicht aber die Attributwerte der assoziierten Entität T beim (Natural-)Join interessiert, nutzt man den *Semi-Join*.

Der Semi-Join $S \ltimes T$ ist für zwei Relationen S und T definiert durch

$$S \ltimes T = \pi_{\text{ident}(S)}(S \bowtie T)$$

Bonus: Anti-Semi-Join

Beim *Anti-Semi-Join* $S \triangleright T$ zweier Relationen S und T werden die Tupel aus S selektiert, die am Natural-Join *nicht* teilnehmen:

$$S \triangleright T = S - (S \ltimes T) = S - \pi_{\text{ident}(S)}(S \bowtie T)$$

Definition: Outer-Join

Ein *Outer-Join* funktioniert im Grunde wie der Inner-Join. Im Kontrast dazu gibt er aber nicht nur die Datensätze beider Tabellen aus, die die Selektionsbedingung erfüllen, sondern zusätzlich auch alle übrigen Tupel der einen bzw. der anderen Tabelle.

Bezogen auf die Leserichtung spricht man von einer linken und einer rechten Relation.

Die jeweiligen Operationen heißen dementsprechend *Left-Outer-Join* und *Right-Outer-Join*.

Bei einem Left-Outer-Join zweier Relationen S und T

$$S \mathbin{⟕} T$$

werden alle Entitäten der Entitätenmenge links der Relation, also S, berücksichtigt, auch wenn es keine zugehörigen Entitäten in Entitätenmenge T gibt. Diese Attribute sind dann `NULL`.

Bei einem Right-Outer-Join zweier Relationen S und T

$$S \mathbin{⟖} T$$

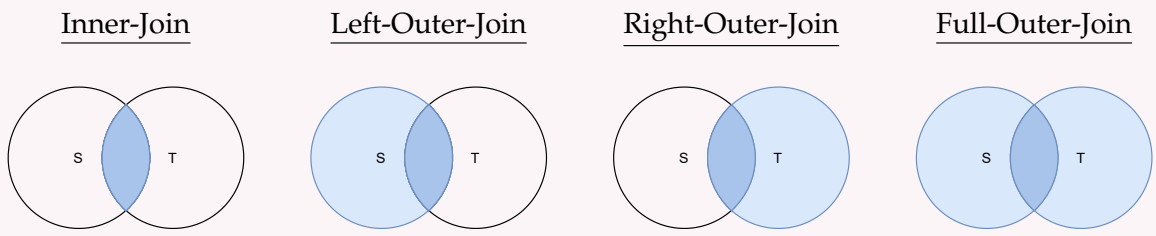
gilt das analog, nur mit vertauschten Rollen.

Der *Full-Outer-Join* zweier Relationen S und T

$$S \mathbin{⟗} T$$

ist die Vereinigung von Left- und Right-Outer-Join. Das bedeutet, es sind alle Entitäten beider Seiten dabei, nur ggf. mit `NULL`-Einträgen in den Attributen der anderen Seite, wenn es keine zugehörige Entität gibt.

Bonus: Inner-Join vs. Outer-Join



SQL: MINUS

Die *Differenz* ist in MySQL bzw. MariaDB als Operation so nicht vorhanden, kann aber leicht über Joins abgebildet werden.

Für zwei Relationen S und T mit $\text{schema}(S) = \text{schema}(T)$ und Schlüsselattribut K gilt:

$$S - T \iff \texttt{SELECT S.* FROM S LEFT OUTER JOIN T USING(K) WHERE isnull(T.K);}$$

SQL: INTERSECT

Die *Schnittmenge* ist in MySQL bzw. MariaDB als Operation so nicht vorhanden, kann aber leicht über Joins abgebildet werden.

Für zwei Relationen S und T mit $\text{schema}(S) = \text{schema}(T)$ und Schlüsselattribut K gilt:

$$S \cap T \iff \texttt{SELECT S.* FROM S JOIN T USING(K);}$$

Definition: Umbenennen von Relationen oder Attributen

Bei einem Join oder kartesischen Produkt kommt es zuweil vor, dass in der Ergebnisrelation eigentlich Attribute gleich heißen würden, was mathematisch und technisch ein Problem ist. Analog kann es notwendig sein, ganzen Relationen einen eigenen Namen zu geben, um etwa bei einem Self-Join diese zu unterscheiden.

Zum *Umbenennen von Relationen und Attributen* wird der Operator ρ verwendet. Das Umbenennen einer Relation S zu S' wird dann mittels

$$\rho_{S'}(S)$$

und das Umbenennen eines Attributes a zu a' einer Relation T mittels

$$\rho_{a \to a'}(T)$$

notiert.

Definition: Division

Für zwei Relationen S und T mit $\text{schema}(S) = \{[a_1, \ldots, a_n, b_1, \ldots, b_k]\}$ und $\text{schema}(T) = \{[b_1, \ldots, b_k]\}$ ist die *Division* $S \div T$ definiert durch

$$S \div T = \{(a_1, \ldots, a_n) \mid \forall (b_1, \ldots, b_k) \in T : (a_1, \ldots, a_n, b_1, \ldots, b_k) \in S\}$$

bzw. äquivalent

$$S \div T = \pi_{S-T}(S) - \pi_{S-T}((\pi_{S-T}(S) \times T) - S)$$

Die Division $\div$ ist also die „Umkehroperation“ zum kartesischen Produkt, denn es gilt symbolisch:

$$(S \times T) \div T = S$$

Hinweis:

Die Ergebnismenge ist nur der a-Anteil von S, also $\pi_a(S) = \pi_{a_1,\ldots,a_n}(S)$.

Beispiel: Schrittweise Division

Gegeben seien zum einen die Tabelle `effektivitaet` als S:

	Multiplikator	Angreifend	Verteidigend
1	1	Boden	Boden
2	1	Boden	Drache
3	1	Boden	Eis
...	...		
324	0.5	Wasser	Wasser

und T mit

	Multiplikator	Verteidigend
1	2	Eis
2	2	Gestein

Wir bestimmen schrittweise die Division $S \div T$:

1. $\pi_{S-T}(S)$:

	S.Angreifend
1	Boden
2	Drache
3	Eis
...	...
18	Wasser

2. $\pi_{S-T}(S) \times T$:

	S.Angreifend	T.Multiplikator	T.Verteidigend
1	Boden	2	Eis
2	Boden	2	Gestein
3	Drache	2	Eis
...	...		
36	Wasser	2	Gestein

3. $\pi_{S-T}(S) \times T - S$:

	S.Angreifend	T.Multiplikator	T.Verteidigend
1	Boden	2	Eis
2	Drache	2	Eis
3	Drache	2	Gestein
...	...		
27	Wasser	2	Gestein

Beispiel: Schrittweise Division (Fortsetzung)

4. $\pi_{S-T}(\pi_{S-T}(S) \times T - S)$

	S.Angreifend
1	Boden
2	Drache
3	Eis
...	...
16	Wasser

5. $\pi_{S-T}(S) - \pi_{S-T}(\pi_{S-T}(S) \times T - S)$:

	S.Angreifend
1	Kampf
2	Stahl

Damit wissen wir nun, dass die Typen Kampf und Stahl beide doppelt effektiv gegen Eis und Gestein sind. □

Definition: Aggregation und Gruppierung

Die *Gruppierung* wendet *Aggregatfunktionen*[a] auf gleiche Attribute in einer Relation an.

Der Operator γ erhält eine Liste von (Gruppierungs-)Attributen $a_1, \ldots, a_n$ und Aggregatfunktionen $f_1, \ldots, f_k$, die dann auf die Tupel angewendet werden, für die die Attribute der Attributliste gleich sind.

Man schreibt für eine Relation S dann

$$\gamma_{a_1,\ldots,a_n;f_1,\ldots,f_k}(S)$$

Jede Aggregatfunktion $f_1, \ldots, f_k$ ergibt eine Spalte in der Ergebnistabelle.

[a] Aggregatfunktionen sind z.B. `COUNT`, `AVG`, `SUM`, `MIN` und `MAX`.

Bonus: Zusammenfassung Notation relationale Algebra

Seien zwei Relationen S und T gegeben.

Funktion	Notation	Hinweise
Selektion	$\sigma_{\Theta}(S)$	filtert Entitäten nach Bedingung Θ
Projektion	$\pi_{a_{i1},\ldots,a_{ik}}(S)$	filtert Attribute zu $a_{i1},\ldots,a_{ik}$
Theta-Join *alternativ:*	$S \bowtie_{\Theta} T$ $\sigma_{\Theta}(S \times T)$	filtert kartesisches Produkt nach Bedingung Θ
Equi-Join *alternativ:*	$S \bowtie_{a_1=a_2} T$ $\sigma_{a_1=a_2}(S \times T)$	Spezialfall des Theta-Join
Natural-Join	$S \bowtie T$	verbindet Entitäten, die in gleich benannten Attributen gleiche Werte haben
Semi-Join *alternativ:*	$S \ltimes T$ $\pi_{\text{ident}(S)}(S \bowtie T)$	berechnet Anteil eines Natural Joins, der nach Reduktion auf S übrig bleibt
Anti-Semi-Join *alternativ:*	$S \rhd T$ $S - \pi_{\text{ident}(S)}(S \bowtie T)$	berechnet Anteil eines Natural Joins, der in keiner Relation zu T steht
Left-Outer-Join	$S \mathbin{⟕} T$	schließt alle Entitäten aus S ein, auch die ohne Relation zu T
Right-Outer-Join	$S \mathbin{⟖} T$	schließt alle Entitäten aus T ein, auch die ohne Relation zu S
Full-Outer-Join	$S \mathbin{⟗} T$	schließt alle Entitäten aus S und T ein
Umbenennung (Relation)	$\rho_{S'}(S)$	benennt Relation S um in S'
Umbenennung (Attribut)	$\rho_{a \to a'}(T)$	benennt Attribut a um in a'
Division	$S \div T$	beschreibt die Tupel aus S, die mit allen Tupeln aus T verknüpft sind
Gruppierung	$\gamma_{a_1,\ldots,a_n;f_1,\ldots,f_k}(S)$	gruppiert nach Attributen $a_1,\ldots,a_n$ und wendet Funktionen $f_1,\ldots,f_k$ an

4.2 Anfrageoptimierung

Definition: Ablauf der Anfrageoptimierung

1. Scanner, Parser, Sichtenauflösung
   - Input: Deklarative Anfrage (Query)
   - Output: Algebraischer Ausdruck
2. Anfrageoptimierer
   - Input: Algebraischer Ausdruck
   - Output: Auswertungsplan („Query Execution Plan (QEP)“)
3. Codeerzeugung, Ausführung
   - Input: Auswertungsplan
   - Führt optimierte Anfrage aus.

Definition: Äquivalenzerhaltende Transformationen

1. Aufbrechen von Konjunktionen im Selektionsprädikat:

$$\sigma_{\Theta_1 \wedge \Theta_2 \wedge \ldots \wedge \Theta_n}(R) \equiv \sigma_{\Theta_1}(\sigma_{\Theta_2}(\ldots(\sigma_{\Theta_n}(R))))$$

2. σ ist kommutativ:

$$\sigma_{\Theta_1}(\sigma_{\Theta_2}(R)) \equiv \sigma_{\Theta_2}(\sigma_{\Theta_1}(R))$$

3. π-Kaskaden:
   - falls $L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n$

$$\pi_{L_1}(\pi_{L_2}(\ldots(\pi_{L_n}(R)))) \equiv \pi_{L_1}(R)$$

4. Vertauschen von π und σ:
   - falls sich Θ nur auf $a_1, \ldots, a_n$ bezieht

$$\pi_{a_1,\ldots,a_n}(\sigma_\Theta(R)) \equiv \sigma_\Theta(\pi_{a_1,\ldots,a_n}(R))$$

5. $\times$, $\cup$, $\cap$, $\bowtie$ sind kommutativ
6. Pushing Selections („Selektiere so früh wie möglich“):
   - falls sich Θ sich nur auf Attribute von R bezieht

$$\sigma_\Theta(R \bowtie_{\Theta_J} S) \equiv \sigma_\Theta(R) \bowtie_{\Theta_J} S$$

   - falls Θ eine Konjunktion der Form $\Theta_1 \wedge \Theta_2$ ist und
     - sich Θ_1 nur auf Attribute von R bezieht
     - sich Θ_2 nur auf Attribute von S bezieht

$$\sigma_{\Theta_1 \wedge \Theta_2}(R \bowtie_{\Theta_J} S) \equiv \sigma_{\Theta_1}(R) \bowtie_{\Theta_J} \sigma_{\Theta_2}(S)$$

7. Vertauschen von π mit $\bowtie$ („Projeziere so früh wie möglich“):
   - Sei Projektionsliste $L = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$
     mit $\{a_1, \ldots, a_n\} \subseteq R$ und $\{b_1, \ldots, b_m\} \subseteq S$ und
   - Joinprädikat Θ_J bezieht sich ausschließlich auf Attribute aus L

$$\pi_L(R \bowtie_{\Theta_J} S) \equiv \pi_{a_1,\ldots,a_n}(R) \bowtie_{\Theta_J} \pi_{b_1,\ldots,b_m}(S)$$

   - Joinprädikat Θ_J bezieht sich auch auf weitere Attribute
     $\{a'_1, \ldots, a'_k\} \subseteq R$ und $\{b'_1, \ldots, b'_l\} \subseteq S$

$$\pi_L(R \bowtie_{\Theta_J} S) \equiv \pi_L(\pi_{a_1,\ldots,a_n,a'_1,\ldots,a'_k}(R) \bowtie_{\Theta_J} \pi_{b_1,\ldots,b_m,b'_1,\ldots,b'_l}(S))$$

8. $\times$, $\cup$, $\cap$, $\bowtie$ sind jeweils assoziativ[a]
9. σ ist distributiv mit $\cup$, $\cap$, $-$
10. π ist distributiv mit $\cup$:

$$\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$$

11. De-Morgans Regeln für Join- und Selektionsprädikate
12. Zusammenfassung von σ und $\times$ zu $\bowtie$:

$$\sigma_\Theta(R \times S) \equiv R \bowtie_\Theta S$$

[a] Die Regel hat viel Potential mit Join!

Bonus: Hinweise zu Transformationen

Das Ziel ist generell, dass ein Worst-Case vermieden wird, nicht aber die optimale Anfrage.

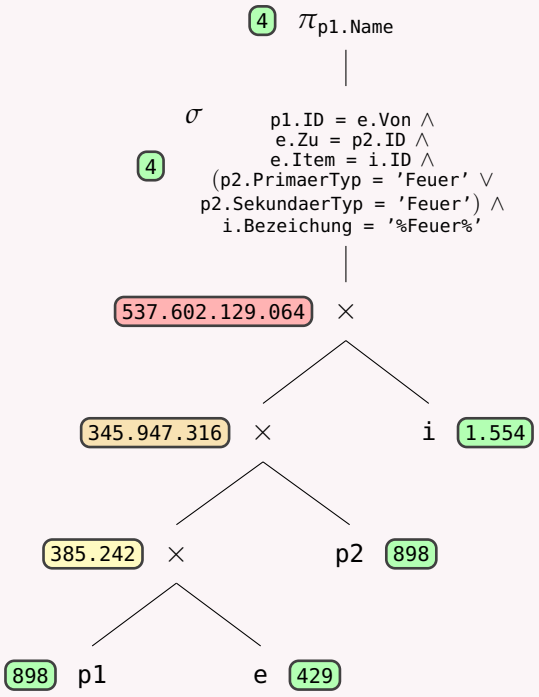
Hinweise zu den Transformationen:

- Die Regeln 2, 4, 6 und 9 verschieben Selektionen so weit nach hinten wie möglich.
- Regel 8 (Assoziativgesetz) vertauscht die Blattknoten so, dass derjenige, der das kleinste Zwischenergebnis liefert, zuerst ausgewertet wird.
- Eine $\times$-Operation, gefolgt von einer σ-Operation wird eine $\bowtie$-Operation.
- Die Regeln 3, 4, 7 und 10 verschieben Projektionen so weit nach hinten wie möglich.

Beispiel: Anfrageoptimierung

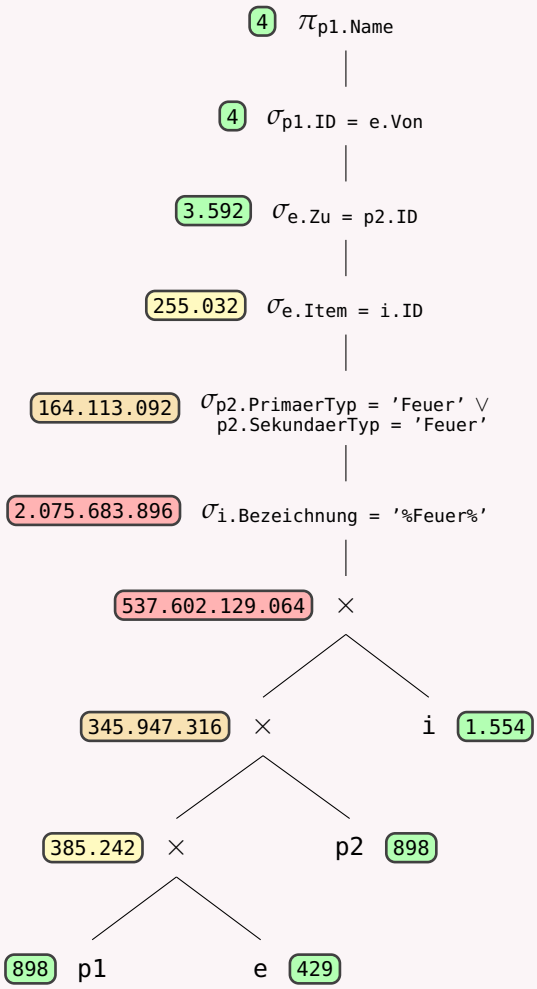
```sql
-- Alle Feuer-Pokemon, die sich durch ein Item mit 'Feuer' im Namen entwickeln
SELECT p1.Name -- Projektion
FROM -- kartesisches Produkt
    pokemon AS p1,
    entwicklung AS e,
    pokemon AS p2,
    item AS i
WHERE
    p1.ID = e.Von -- Join
    AND e.Zu = p2.ID -- Join
    AND e.Item = i.ID -- Join
    AND (
        p2.PrimaerTyp = 'Feuer'
        OR p2.SekundaerTyp = 'Feuer'
    ) -- Selektion
    AND i.Bezeichnung LIKE "%Feuer%"; -- Selektion
```

Die kanonische Übersetzung der Anfrage ist:



Beispiel: Anfrageoptimierung (Aufbrechen von Konjunktionen)

Wir brechen die Konjunktionen im Selektionsprädikat auf und erhalten:

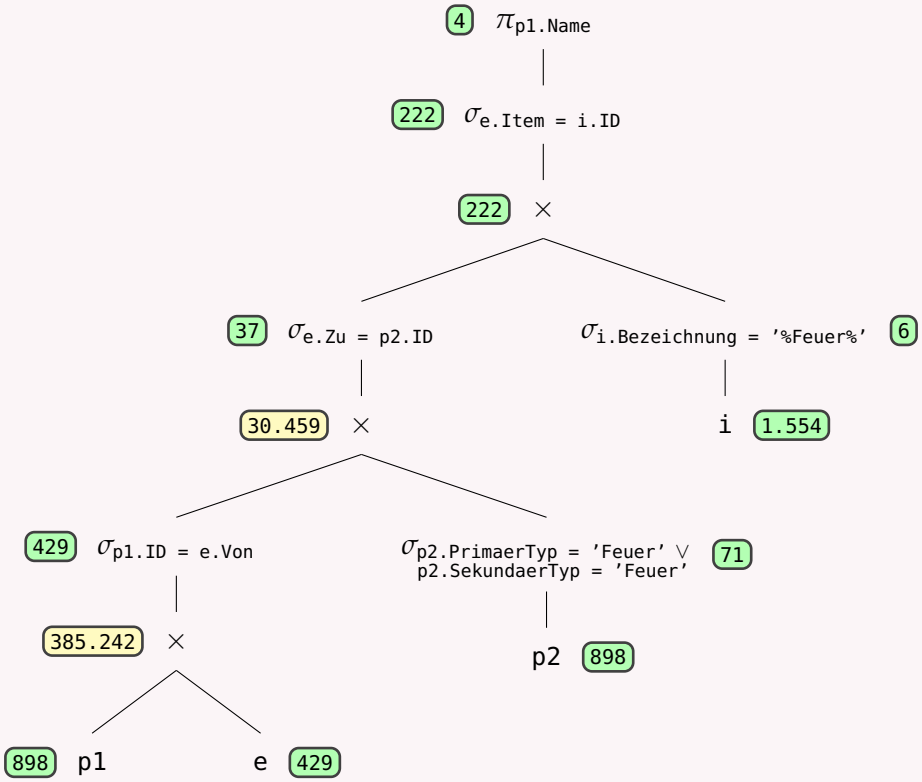


Beispiel: Anfrageoptimierung (Pushing Selections)

Wir wollen so früh wie möglich selektieren und erkennen z.B.:

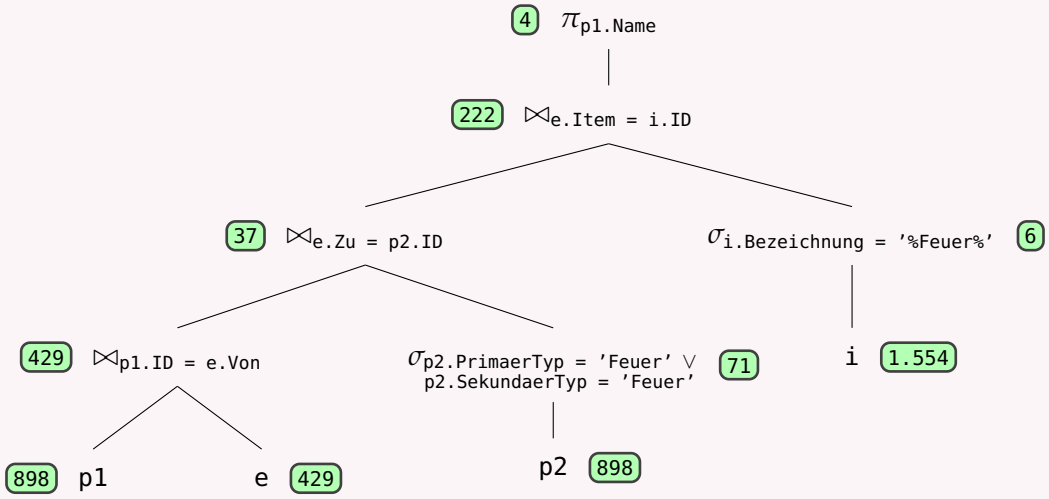
- `i.Bezeichnung = '%Feuer%'` bezieht sich nur auf `i`
- `p2.PrimaerTyp = 'Feuer'` und `p2.SekundaerTyp = 'Feuer'` beziehen sich nur auf `p2`

Wir verschieben die Selektionen „so weit nach unten wie möglich“ und erhalten:



Beispiel: Anfrageoptimierung (Selektionen und Kreuzprodukte zu Joins)

Wir können Kreuzprodukte gefolgt von Selektionen zu Joins zusammenfassen und erhalten damit:



5 Normalformen

Definition: Funktionale Abhängigkeit

Eine Relation wird durch Attribute definiert. Bestimmen einige dieser Attribute eindeutig die Werte anderer Attribute, so spricht man von *funktionaler Abhängigkeit*.

Sei $R = \{a, \ldots, a_n\}$ ein vereinfachtes Schema, $\alpha \subseteq \text{ident}(R)$, $\beta \subseteq \text{ident}(R)$, und $D \subseteq \text{dom}(R)$.

Dann ist β funktional abhängig von α genau dann, wenn für alle zulässigen D gilt:

$$\forall s, t \in D : s.\alpha = t.\alpha \implies s.\beta = t.\beta$$

Man schreibt:[a]

$$\alpha \to \beta \iff \text{Wenn } \alpha \text{ bekannt ist, dann auch } \beta \iff \beta \text{ ist funktional abhängig von } \alpha$$

Für eine Menge von funktionalen Abhängigkeiten einer Relation R schreibt man FD bzw. $\text{FD}(R)$.

[a] $\alpha \to \beta$ heißt, für alle Tupel s, t mit gleichen α-Attributen gilt, dass auch ihre β-Attribute gleich sind. Die Werte der Attribute aus der Attributmenge α bestimmen also eindeutig die Werte der Attribute aus der Attributmenge β.

Definition: Volle funktionale Abhängigkeit

Sei $R = \{a, \ldots, a_n\}$ ein vereinfachtes Schema und $\alpha \to \beta$ eine funktionale Abhängigkeit.

Dann ist β *voll funktional abhängig* von α, wenn

$$\forall \gamma \in \mathcal{P}(\alpha) - \alpha : \alpha - \gamma \not\to \beta$$

Man schreibt:[a]

$$\alpha \dot{\to} \beta \iff \alpha \text{ ist minimal}$$

[a] Volle funktionale Abhängigkeit bedeutet, dass α minimal ist, man also kein Attribut aus α weglassen kann, ohne dass man die funktionale Abhängigkeit verliert.

Definition: Schlüssel

Sei $R = \{a_1, \ldots, a_n\}$ ein vereinfachtes Schema und $\alpha \subseteq \text{ident}(R)$.

Dann ist

- α ein *Superschlüssel*, falls $\alpha \rightarrow \text{ident}(R)$, und
- α ein *Schlüsselkandidat*, falls $\alpha \dot{\rightarrow} \text{ident}(R)$.

Ein *Primärschlüssel* ist ein ausgesuchter bzw. festgelegter Schlüsselkandidat.

Ein Attribut $a \in \text{ident}(R)$ heißt *prim*, falls a Attribut eines Schlüsselkandidaten von R ist, sonst *nicht prim*.

Es macht Sinn, aus den möglichen Schlüsselkandidaten einer Relation R genau einen Primärschlüssel festzulegen, da Verweise auf Tupel aus R über Fremdschlüssel in anderen Relationen realisiert werden, also Schlüsselattribute dieses Primärschlüssels.

Nicht jedes α mit $\alpha \rightarrow \text{ident}(R)$ ist ein Schlüsselkandidat, denn α muss nicht minimal sein. Insbesondere ist $\alpha = \text{ident}(R)$ der triviale Superschlüssel.

Definition: Armstrong-Axiome

Mit Hilfe der *Armstrong-Axiome* lassen sich aus einer Menge von funktionalen Abhängigkeiten, die auf einer Relation gelten, weitere funktionale Abhängigkeiten ableiten.

Die folgenden drei Regeln reichen aus, um alle funktionalen Abhängigkeiten herzuleiten:

1. *Reflexivität*:
   Eine Menge von Attributen bestimmt eindeutig die Werte einer Teilmenge dieser Attribute (triviale Abhängigkeit):
   $$\beta \subseteq \alpha \implies \alpha \to \beta$$
2. *Erweiterungsregel, Verstärkung*:
   Gilt $\alpha \to \beta$, so gilt auch $\alpha\gamma \to \beta\gamma$ für jede Menge von Attributen $\gamma \in \text{schema}(R)$:
   $$\alpha \to \beta \implies \alpha\gamma \to \beta\gamma \qquad \forall \gamma \in \text{schema}(R)$$
3. *Transitivitätsregel*:
   Gilt $\alpha \to \beta$ und $\beta \to \gamma$, so gilt auch $\alpha \to \gamma$:
   $$(\alpha \to \beta) \wedge (\beta \to \gamma) \implies \alpha \to \gamma$$

Um Herleitungen einfacher zu gestalten, können zusätzlich die folgenden (abgeleiteten) Regeln verwendet werden:

4. *Vereinigungsregel*:
   Gelten $\alpha \to \beta$ und $\alpha \to \gamma$, so gilt auch $\alpha \to \beta\gamma$:
   $$(\alpha \to \beta) \wedge (\alpha \to \gamma) \implies \alpha \to \beta\gamma$$
5. *Dekompositions-/Zerlegungsregel*:
   Gilt $\alpha \to \beta\gamma$, so gelten auch $\alpha \to \beta$ und $\alpha \to \gamma$:
   $$\alpha \to \beta\gamma \implies (\alpha \to \beta) \wedge (\alpha \to \gamma)$$
6. *Pseudotransitivitätsregel*:
   Gilt $\alpha \to \beta$ und $\beta\gamma \to \delta$, so gilt auch $\alpha\gamma \to \delta$:
   $$(\alpha \to \beta) \wedge (\beta\gamma \to \delta) \implies \alpha\gamma \to \delta$$

Definition: Attributhülle

Zu einem Schema R und einer Menge von funktionalen Abhängigkeiten FD funktionaler Abhängigkeiten bestimmt die *Attributhülle*

$$\text{AttrHülle}(\text{FD}, \alpha) \quad \text{bzw.} \quad \alpha^+ \text{ (wenn FD klar)}$$

alle Attribute, die bzgl. der FD funktional abhängig von α sind.

Algorithmus: Bestimmen der Attributhülle

Um die *Attributhülle* für eine Menge von Attributen α bzgl. einer Menge von funktionaler Abhängigkeiten FD zu bestimmen, geht man folgendermaßen vor:

Input:

- FD: Menge funktionaler Abhängigkeiten
- α: Menge von Attributen

Output:

- AttrHülle(FD, α): Attributhülle bzw. transitiver Abschluss von α bzgl. FD

Algorithmus:

```
1: ergebnis = {α}                                   ▷ Attributhülle von α
2: alt = {∅}
3: while (ergebnis != alt) do                       ▷ Solange Änderungen stattfinden ...
4:     alt = ergebnis
5:     for each (β → γ in FD) do
6:         if β in ergebnis then                    ▷ Prüfen, ob Transitivitätsbedingung erfüllt ist
7:             ergebnis += γ
8:         end if
9:     end for
10: end while
11: return ergebnis
```

Beispiel: Bestimmen der Attributhülle

Gegeben sei ein abstraktes Relationenschema $R = \{A, B, C, D, E, F\}$ mit den funktionalen Abhängigkeiten:

$$\text{FD} = \{A \to BC,\ C \to DA,\ E \to ABC,\ F \to CD,\ CD \to BEF\}$$

Bestimmen Sie die Attributhülle von A.

Nach dem bekannten Algorithmus gilt:

Schritt	betrachtete FD	AttrHülle(A)
0		$\{A\}$
1	$A \to BC$	$\{A, B, C\}$
2	$C \to DA$	$\{A, B, C, D\}$
3	$E \to ABC$	$\{A, B, C, D\}$
4	$F \to CD$	$\{A, B, C, D\}$
5	$CD \to BEF$	$\{A, B, C, D, E, F\}$

Wir können nach Schritt 5 aufhören, da A dort bereits ein Superschlüssel ist. □

Bonus: Tipps zum Bestimmen der Schlüsselkandidaten

Alle Attributhüllen der Elemente aus $\mathcal{P}(\text{ident}(R))$ zu bestimmen kann schnell zu aufwendig werden. Man kann allerdings „klug“ anfangen:

- Alle Attribute, die nicht gefolgert werden können[a], müssen in die Schlüsselkandidaten.
- Im Laufe der Berechnung einer Attributhülle ergeben sich Attributkombinationen, die z.B. einen Schlüsselkandidaten enthalten. Dann ist man schon fertig mit dieser Attributhülle.

[a] Das sind die Attribute, die auf keiner „rechten Seite“ einer funktionalen Abhängigkeit stehen.

Beispiel: Bestimmen der Schlüsselkandidaten

Gegeben sei ein abstraktes Relationenschema $R = \{A, B, C, D, E, F\}$ mit den funktionalen Abhängigkeiten:

$$\text{FD} = \{A \to BC,\ C \to DA,\ E \to ABC,\ F \to CD,\ CD \to BEF\}$$

Bestimmen Sie alle Schlüsselkandidaten.

In dem voherigen Beispiel haben wir bereits festgestellt, dass A ein Superschlüssel ist. A ist offensichtlich minimal und damit bereits ein Schlüsselkandidat.

$$\text{FD} = \{A \to BC,\ C \to DA,\ E \to ABC,\ F \to CD,\ CD \to BEF\}$$

Wir erkennen aus den funktionalen Abhängigkeiten auch, dass A aus C und E gefolgert werden kann. Beide sind wieder minimal, da einelementig, und damit ebenfalls Schlüsselkandidaten.

$$\text{FD} = \{A \to BC,\ C \to DA,\ E \to ABC,\ F \to CD,\ CD \to BEF\}$$

Wieder erkennen wir, dass C aus F gefolgert werden kann. Analog ist F wieder ein Schlüsselkandidat.

$$\text{FD} = \{A \to BC,\ C \to DA,\ E \to ABC,\ F \to CD,\ CD \to BEF\}$$

Wir können keinen einelementigen Superschlüssel mehr identifizieren. Die letzte prüfenswerte Option wäre CD. CD ist zwar Superschlüssel, aber nicht minimal, da D weggelassen werden kann.

Damit sind die Schlüsselkandidaten:

$$A, C, E, F$$

□

Definition: Äquivalente Menge funktionaler Abhängigkeiten

Zu einem Schema R und einer Menge FD funktionaler Abhängigkeiten bezeichne FD^+ die Menge aller funktionalen Abhängigkeiten, die von FD impliziert werden.

Zwei Mengen funktionaler Abhängigkeiten FD_1 und FD_2 heißen *äquivalent*, falls

$$\text{FD}_1^+ = \text{FD}_2^+ \quad \Longleftrightarrow \quad \text{FD}_1 \equiv \text{FD}_2$$

Sind zwei Mengen funktionaler Abhängigkeiten nur in wenigen Abhängigkeiten verschieden, genügt die Betrachtung der relevanten Attributhüllen in den jeweiligen Mengen.[a]

[a] z.B. $R = \{A, B, C, D, E\}$, $\text{FD}_1 = \{A \to B, B \to C, AB \to CD\}$, $\text{FD}_2 = \{A \to B, B \to C, A \to CD\}$.
Hier reduziert sich $\text{FD}_1^+ = \text{FD}_2^+$ auf $CD \subseteq \text{AttrHülle}(\text{FD}_1, AB)$ und $CD \subseteq \text{AttrHülle}(\text{FD}_2, A)$.

Definition: Kanonische Überdeckung

Zu einer gegebenen Menge FD von funktionalen Abhängigkeiten nennt man FD^c eine *kanonische Überdeckung*, wenn folgende drei Eigenschaften erfüllt sind:

- $\text{FD}^c \equiv \text{FD}$, d.h. $(\text{FD}^c)^+ = \text{FD}^+$
- Alle funktionalen Abhängigkeiten in $\alpha \to \beta \in \text{FD}^c$ sind minimal, d.h.
  - $\forall a \in \alpha : \text{FD}^c - (\alpha \to \beta) \cup (\alpha - a \to \beta) \not\equiv \text{FD}^c$
  - $\forall b \in \beta : \text{FD}^c - (\alpha \to \beta) \cup (\alpha \to \beta - b) \not\equiv \text{FD}^c$
- Jede linke Seite einer funktionalen Abhängigkeit in FD^c ist einzigartig.

Algorithmus: Berechnung der kanonischen Überdeckung

Gegeben sind ein Schema R und eine Menge FD funktionaler Abhängigkeiten.

1. *Linksreduktion*:
   Für alle $\alpha \to \beta \in \text{FD}$ überprüfe, ob ein $a \in \alpha$ überflüssig ist, d.h. ob:
   $$\beta \subseteq \text{AttrHülle}(\text{FD}, \alpha - a)$$
   In diesem Fall wird $\alpha \to \beta$ durch $\alpha - a \to \beta$ in FD ersetzt.
2. *Rechtsreduktion*:
   Für alle $\alpha \to \beta \in \text{FD}$ überprüfe, ob ein $b \in \beta$ überflüssig ist, d.h. ob:
   $$b \subseteq \text{AttrHülle}(\text{FD} - \{\alpha \to \beta\} \cup \{\alpha \to \beta - b\}, \alpha)$$
   In diesem Fall wird $\alpha \to \beta$ durch $\alpha \to \beta - b$ in FD ersetzt.
3. *Entfernung* potentiell entstandener $\alpha \to \emptyset$.
4. *Vereinigung* potentiell entstandener $\alpha \to \beta_1$ und $\alpha \to \beta_2$ zu $\alpha \to \beta_1\beta_2$ (Vereinigung).

Beispiel: Berechnung der kanonischen Überdeckung (Linksreduktion)

Gegeben sei ein abstraktes Relationenschema $R = \{A, B, C, D, E, F\}$ mit den funktionalen Abhängigkeiten:

$$\text{FD} = \{A \to BC,\ C \to DA,\ E \to ABC,\ F \to CD,\ CD \to BEF\}$$

Bestimmen Sie zu den gegegben funktionalen Abhängigkeiten die kanonische Überdeckung.

Linksreduktion:

Die einzige zu betrachtende funktionale Abhängigkeit ist $CD \to BEF$[a]

- Ist C überflüssig?

$$\text{AttrHülle}(\text{FD}, \{D\}) = \{D\} \quad \wedge \quad \{B, E, F\} \not\subseteq \{D\}$$

- Ist D überflüssig?
  Wir berechnen $\text{AttrHülle}(\text{FD}, \{C\})$:

Schritt	betrachtete FD	AttrHülle(C)
0		$\{C\}$
1	$A \to BC$	$\{C\}$
2	$C \to DA$	$\{A, C, D\}$
3	$E \to ABC$	$\{A, C, D\}$
4	$F \to CD$	$\{A, C, D\}$
5	$CD \to BEF$	$\{A, B, C, D, E, F\}$

  Damit gilt:

$$\text{AttrHülle}(\text{FD}, \{C\}) = \{A, B, C, D, E, F\} \quad \wedge \quad \{B, E, F\} \subseteq \{A, B, C, D, E, F\}$$

  Damit können wir $CD \to BEF$ zu $C \to BEF$ reduzieren.

Damit sind unsere funktionalen Abhängigkeiten reduziert zu:

$$\text{FD} = \{A \to BC,\ C \to DA,\ E \to ABC,\ F \to CD,\ C \to BEF\}$$

[a] Alle anderen funktionalen Abhängigkeiten sind „links einstellig“.

Beispiel: Berechnung der kanonischen Überdeckung (Rechtsreduktion)

Wir haben durch eine Linksreduktion unsere funktionalen Abhängigkeiten reduziert auf:

$$\text{FD} = \{A \to BC,\ C \to DA,\ E \to ABC,\ F \to CD,\ C \to BEF\}$$

Rechtsreduktion:

- Betrachte $A \to BC$:
  - Ist B überflüssig? ($A \to BC$ wird zu $A \to C$)

    $$A \to C \to BEF \implies B \in \text{AttrHülle}(\text{FD} - \{A \to BC\} \cup \{A \to C\}, \{A\})$$

    Damit ist B überflüssig.
  - Ist C überflüssig? ($A \to BC$ wird zu $A \to B$)

    $$\text{AttrHülle}(\text{FD} - \{A \to BC\} \cup \{A \to B\}, \{A\}) = \{A, B\} \not\ni C$$

  Damit können wir $A \to BC$ zu $A \to C$ reduzieren und erhalten:

  $$\boxed{\text{FD} = \{A \to C,\ C \to DA,\ E \to ABC,\ F \to CD,\ C \to BEF\}}$$

- Betrachte $C \to DA$:
  - Ist D überflüssig? ($C \to DA$ wird zu $C \to A$)

    $$C \to BEF, F \to CD \implies D \in \text{AttrHülle}(\text{FD} - \{C \to DA\} \cup \{C \to A\}, \{C\})$$

    Damit ist D überflüssig.
  - Ist A überflüssig? ($C \to DA$ wird zu $C \to D$)

    $$C \to BEF, E \to ABC \implies A \in \text{AttrHülle}(\text{FD} - \{C \to DA\} \cup \{C \to D\}, \{C\})$$

    Damit ist A überflüssig.

  Damit können wir $C \to DA$ zu $C \to \emptyset$ reduzieren und erhalten:

  $$\boxed{\text{FD} = \{A \to C,\ C \to \emptyset,\ E \to ABC,\ F \to CD,\ C \to BEF\}}$$

Beispiel: Berechnung der kanonischen Überdeckung (Rechtsreduktion) (Fortsetzung)

- Betrachte $E \to ABC$:
  - Ist A überflüssig? ($E \to ABC$ wird zu $E \to BC$)

    $$\text{AttrHülle}(\text{FD} - \{E \to ABC\} \cup \{E \to BC\}, \{E\}) = \{B, C, D, E, F\} \not\ni A$$

  - Ist B überflüssig? ($E \to ABC$ wird zu $E \to AC$)

    $$E \to AC, C \to BEF \implies B \in \text{AttrHülle}(\text{FD} - \{E \to ABC\} \cup \{E \to AC\}, \{E\})$$

    Damit ist B überflüssig.
  - Ist C überflüssig? ($E \to ABC$ wird zu $E \to AB$)

    $$E \to AB, A \to C \implies C \in \text{AttrHülle}(\text{FD} - \{E \to ABC\} \cup \{E \to AB\}, \{E\})$$

    Damit ist C überflüssig.

  Damit können wir $E \to ABC$ zu $E \to A$ reduzieren und erhalten:

  $$\boxed{\text{FD} = \{A \to C, C \to \emptyset, E \to A, F \to CD, C \to BEF\}}$$

- Betrachte $F \to CD$:
  - Ist C überflüssig? ($F \to CD$ wird zu $F \to D$)

    $$\text{AttrHülle}(\text{FD} - \{F \to CD\} \cup \{F \to D\}, \{F\}) = \{D, F\} \not\ni C$$

  - Ist D überflüssig? ($F \to CD$ wird zu $F \to C$)

    $$\text{AttrHülle}(\text{FD} - \{F \to CD\} \cup \{F \to C\}, \{F\}) = \{B, C, E, F\} \not\ni D$$

  Damit können wir $F \to CD$ nicht reduzieren.
- Betrachte $C \to BEF$:
  - Ist B überflüssig? ($C \to BEF$ wird zu $C \to EF$)

    $$\text{AttrHülle}(\text{FD} - \{C \to BEF\} \cup \{C \to EF\}, \{C\}) = \{C, D, E, F\} \not\ni B$$

  - Ist E überflüssig? ($C \to BEF$ wird zu $C \to BF$)

    $$\text{AttrHülle}(\text{FD} - \{C \to BEF\} \cup \{C \to BF\}, \{C\}) = \{B, C, D, F\} \not\ni E$$

  - Ist F überflüssig? ($C \to BEF$ wird zu $C \to BE$)

    $$\text{AttrHülle}(\text{FD} - \{C \to BEF\} \cup \{C \to BE\}, \{C\}) = \{B, C, E\} \not\ni F$$

  Damit können wir $C \to BEF$ nicht reduzieren.

Damit sind unsere funktionalen Abhängigkeiten reduziert zu:

$$\text{FD} = \{A \to C, C \to \emptyset, E \to A, F \to CD, C \to BEF\}$$

Beispiel: Berechnung der kanonischen Überdeckung (Entfernung und Vereinigung)

Wir haben durch Links- und Rechtsreduktion unsere funktionalen Abhängigkeiten reduziert auf:

$$\text{FD} = \{A \to C, C \to \emptyset, E \to A, F \to CD, C \to BEF\}$$

Entfernung:

Wir erkennen, dass wir $C \to \emptyset$ weglassen können und erhalten:

$$\text{FD} = \{A \to C, E \to A, F \to CD, C \to BEF\}$$

Vereinigung:

Wir haben keine gleichen linken Seiten, also können wir keine funktionalen Abhängigkeiten vereinigen.

Wir erhalten final unsere kanonische Überdeckung FD^c mit

$$\text{FD}^c = \{A \to C, E \to A, F \to CD, C \to BEF\}$$

□

Bonus: Tipps zur Berechnung der kanonischen Überdeckung

- Linksreduktion:
  - Hier sind nur die funktionalen Abhängigkeiten zu überprüfen, die links mindestens zwei Attribute besitzen
- Rechtsreduktion:
  - Hier sind alle funktionalen Abhängigkeiten zu prüfen.
  - Es kann hilfreich sein, alle funktionalen Abhängigkeiten aus FD mit mehr auf einem Attribut auf der rechten Seite aufzuspalten (Dekomposition).
    Das führt zwar zu einer größeren Menge FD', aber die Komplexität der Abfragen wird deutlich reduziert.
    im letzten Schritt müssen die aufgespaltenen funktionalen Abhängigkeiten ohnehin wieder vereinigt werden.

Bonus: CRUD

Unter *CRUD* versteht man die fundamentalen Datenbankoperationen:

- *Create*: Datensatz anlegen,
- *Read/Retrieve*: Datensatz lesen,
- *Update*: Datensatz aktualisieren und
- *Delete/Destroy*: Datensatz löschen.

Im Gegensatz zum Lesezugriff (Read) können die Daten verändernden Operationen Create, Update und Delete zu Anomalien innerhalb der Datenbank führen.

Definition: Anomalie

Anomalien bezeichnen in relationalen Datenbanken Fehlverhalten der Datenbank durch Verletzung der Regel „every information once“. Das bedeutet, dass das zugrunde liegende Datenmodell Tabellen mit Spalten gleicher Bedeutung und darüber hinaus auch noch mit abweichenden (anomalen) Inhalten zulässt, so dass nicht mehr erkennbar ist, welche Tabelle bzw. Spalte den richtigen Inhalt enthält (Dateninkonsistenz)

Man unterscheides zwischen:

- *Update-Anomalien*
  - tritt generell auf, wenn redundante Daten in einem Tupel nur teilweise bzw. falsch aktualisiert werden
- *Einfüge-Anomalien*
  - generell Daten so verbunden, dass sie nicht ohne andere (nicht `NULL`) eingegeben werden können
- *Lösch-Anomalien*
  - entsteht, wenn durch das Löschen eines Datensatzes mehr Informationen als erwünscht verloren gehen

Definition: Verlustlosigkeit und Abhängigkeitserhaltung

Gegeben sei eine Relation R mit einer Menge funktionaler Abhängigkeiten. Diese Relation soll in neue Relationen R_i aufgeteilt werden, z.B. um Anomalien zu vermeiden.

- *Verlustlosigkeit/Verbundtreue*:
  - Die in der ursprünglichen Relation R enthaltenen Informationen müssen aus den neuen Relationen $R_1, \ldots, R_n$ mittels natürlichen Verbunds (Natural-Join) rekonstruierbar sein.
  - Eine Zerlegung einer Relation R in zwei Relationen R_1 und R_2 ist *verlustlos*, wenn man mindestens eine der beiden aus dem gemeinsamen Bereich (Überlappung der Attribute) wieder ableiten kann.
  - Es gilt:
    $$R_1 \cap R_2 \to R_1 \quad \text{oder} \quad R_1 \cap R_2 \to R_2$$
- *Abhängigkeitserhaltung*:
  - Die ursprünglich geltenden *funktionalen Abhängigkeiten* müssen auch auf der Zerlegung, also den neuen Relationen, gelten.

Definition: Erste Normalform

Ein Schema in *erster Normalform* (1NF, auch NF1) besitzt nur atomare bzw. elementare Attribute, d.h. kein Attribut ist zusammengesetzt oder mehrwertig.

Funktionale Abhängigkeiten spielen keine Rolle.

Algorithmus: Überführung in erste Normalform

Um 1NF zu erreichen, müssen Sachverhalte in mehrere Attribute getrennt und Mehrwertigkeiten, typischerweise in eine 1:n-Relation, aufgespalten werden.

Definition: Zweite Normalform

Ein Schema R ist in *zweiter Normalform* (2NF, auch NF2), wenn es in 1NF vorliegt und jedes Attribut entweder

- prim ist, oder
- voll funktional abhängig von jedem Schlüsselkandidaten ist.

Algorithmus: Überführung in zweite Normalform

Um 1NF zu erreichen, müssen Sachverhalte in mehrere Attribute getrennt und Mehrwertigkeiten, typischerweise in eine 1:N-Relation, aufgespalten werden.

Für den Test, ob 2NF vorliegt, benötigt man alle Schlüsselkandidaten.

Wenn jedes Attribut in irgendeinem Schlüsselkandidaten vorkommt, ist 2NF erfüllt, da es nur prim Attribute gibt.

2NF kann nur verletzt sein, wenn ein Schlüsselkandidat zusammengesetzt ist.

Definition: Dritte Normalform

Ein Schema R ist in *dritter Normalform* (3NF, auch NF3), wenn es in 2NF vorliegt und jedes nicht-prim Attribut direkt, also nicht transitiv, von einem Schlüsselkandidaten abhängt.

Gemeint ist, dass bei einer vorliegenden transitiven Abhängigkeit b von β, also

$$\beta \to a \to b$$

wobei β ein Schlüsselkandidat ist, hier a kein Schlüsselkandidat ist.

Definition: Dritte Normalform (Alternative)

Ein Schema R mit einer Menge funktionaler Abhängigkeiten FD ist in *dritter Normalform* (3NF, auch NF3), wenn für jede funktionale Abhängigkeit $\alpha \to \beta \in \text{FD}^+$ mindesten eine der folgenden Bedingungen gilt:

- $\alpha \to \beta$ ist trivial, also $\beta \subseteq \alpha$,
- β enthält nur prim Attribute,
- α ist Superschlüssel.

Für den Test, ob 3NF vorliegt, benötigt man alle Schlüsselkandidaten.

Die zweite Normalform ist eingeschlossen.

Algorithmus: Überführung in dritte Normalform

Der Synthesealgorithmus überführt R verlustlos und abhängigkeitserhaltend in 3NF.

Algorithmus: Synthesealgorithmus

1. Bestimme die kanonische Überdeckung FD^c zu FD:
   a) Linksreduktion
   b) Rechtsreduktion
   c) Entfernung von funktionalen Abhängigkeiten der Form $\alpha \to \emptyset$
   d) Zusammenfassung gleicher linker Seiten
2. Für jede funktionale Abhängigkeit $\alpha \to \beta \in \mathrm{FD}^c$:
   - Kreiere ein Relationenschema $R_a := \alpha \cup \beta$
   - Ordne R_a die funktionalen Abhängigkeiten

   $$\mathrm{FD}_a = \{\alpha' \to \beta' \in \mathrm{FD}^c \mid \alpha' \cup \beta' \subseteq R_a\}$$

   zu.
3. Falls eines der in Schritt 2. erzeugten Schemata einen Schlüsselkandidaten von R bzgl. FD^c enthält, sind wir fertig.
   Sonst wähle einen Schlüsselkandidaten $x \in R$ aus und definiere folgendes Schema:

   $$R_x = x \quad \wedge \quad \mathrm{FD}_x = \emptyset$$

4. Eliminiere diejenigen Schemata R_a, die in einem anderen Relationenschema R'_a enthalten sind, d.h.:

   $$R_a \subseteq R'_a$$

Beispiel: Synthesealgorithmus

Gegeben sei ein abstraktes Relationenschema $R = \{A, \underline{B}, C, \underline{D}, E, F, \underline{G}\}$ mit der kanonischen Überdeckung:

$$\mathrm{FD}^c = \{CD \to A,\ A \to C,\ B \to C,\ D \to E,\ E \to F\}$$

und dem Schlüsselkandidaten BDG.

Überführen Sie die Relation in die dritte Normalform, indem Sie den Synthesealgorithmus anwenden.

Schritt 1:

Die kanonische Überdeckung FD^c wurde bereits gegeben.

Schritt 2:

Wir erstellen für jede funktionale Abhängigkeit ein Relationenschema:

- $CD \to A \implies R_1 = \{A, C, D\} \quad \wedge \quad \mathrm{FD}_1 = \{CD \to A,\ A \to C\}$
- $A \to C \implies R_2 = \{A, C\} \quad \wedge \quad \mathrm{FD}_2 = \{A \to C\}$
- $B \to C \implies R_3 = \{B, C\} \quad \wedge \quad \mathrm{FD}_3 = \{B \to C\}$
- $D \to E \implies R_4 = \{D, E\} \quad \wedge \quad \mathrm{FD}_4 = \{D \to E\}$
- $E \to F \implies R_5 = \{E, F\} \quad \wedge \quad \mathrm{FD}_5 = \{E \to F\}$

Schritt 3:

Kein Relationenschema enthält einen Schlüsselkandidaten, deswegen müssen wir ein weiteres Schema mit dem ursprünglichen Schlüssel anlegen:

$$R_S = \{\underline{B}, \underline{D}, \underline{G}\}$$

Schritt 4:

Wir erkennen:

$$R_2 \subseteq R_1$$

Damit können wir R_2 entfernen.

Übrig bleibt nun die überführte Relation in 3NF. □

Definition: Boyce-Codd Normalform

Ein Schema R mit einer Menge funktionaler Abhängigkeiten FD ist in *Boyce-Codd Normalform* (BCNF), wenn für jede funktionale Abhängigkeit $\alpha \to \beta \in \mathrm{FD}^+$ mindestens eine der folgenden Bedingungen gilt:

- $\alpha \to \beta$ ist trivial, also $\beta \subseteq \alpha$
- α ist Superschlüssel

Die dritte Normalform ist eingeschlossen.

Man kann jedes Schema R mit funktionalen Abhängigkeiten FD so in Schemata $R_1, \ldots, R_n$ zerlegen, dass die BCNF erfüllt ist.[a]

[a] Die Zerlegung nicht notwendigerweise abhängigkeitserhaltend.

Algorithmus: Überführung in Boyce-Codd Normalform

Der Dekompositionsalgorithmus setzt genau das um und durch die Aufteilung der Relationen können funktionale Abhängigkeiten verloren gehen.

Algorithmus: Dekompositionsalgorithmus

1. Starte mit $Z = \{R\}$
2. Solange es noch ein Relationenschema R_i in Z gibt, das nicht in BCNF ist, mache folgendes:[a]
   a) Finde eine solche funktionale Abhängigkeit[b]
   b) Zerlege R_i in $R_{i_1} = \alpha \cup \beta$ und $R_{i_2} = R_i - \beta$[c]
   c) Entferne R_i aus Z und füge R_{i_1} und R_{i_2} ein:

$$Z = (Z - \{R_i\}) \cup \{R_{i_1}\} \cup \{R_{i_2}\}$$

[a] Es gibt also eine für R_i geltende funktionale Abhängigkeit $\alpha \to \beta$ mit $\alpha \cap \beta = \emptyset$ (nicht trivial) und $\neg(\alpha \to R_i)$ (α also kein Superschlüssel)

[b] Man sollte die funktionale Abhängigkeit so wählen, dass β alle von α funktional abhängigen Attribute $b \in (R_i - \alpha)$ enthält, damit der Dekompositionsalgorithmus möglichst schnell terminiert.

[c] In diesem Schritt gehen potentiell Abhängigkeiten verloren.

Beispiel: Dekompositionsalgorithmus

Gegeben sei die Relation $R = \{A, B, C, D, E, F\}$ mit den funktionalen Abhängigkeiten

$$\text{FD} = \{B \to DA,\ DEF \to B,\ C \to EA\}$$

Wenden Sie den Dekompositionsalgorithmus an, um die Relation R in die BCNF zu zerlegen und unterstreichen Sie die Schlüssel der Teilrelationen des Endergebnisses.

- Starte mit $Z = \{R\}$.
- Ist R in BCNF? $\lightning$
  - $B \to DA$ verletzt BCNF[a]
    - Zerlegung von R anhand der funktionalen Abhängigkeit $B \to DA$:
      - $R_1 = \alpha \cup \beta = \{A, B, D\} \quad \wedge \quad \text{FD}_1 = \{B \to DA\}$
      - $R_2 = R - \beta = \{B, C, E, F\} \quad \wedge \quad \text{FD}_2 = \{C \to E\}$[b]
  - Ist R_1 in BCNF? ✓
  - Ist R_2 in BCNF? $\lightning$
    - Zerlegung von R_2 anhand der funktionalen Abhängigkeit $C \to E$:
      - $R_{2_1} = \alpha \cup \beta = \{C, E\} \quad \wedge \quad \text{FD}_{2_1} = \{C \to E\}$
      - $R_{2_2} = R_2 - \beta = \{B, C, F\} \quad \wedge \quad \text{FD}_{2_1}$ trivial
  - Ist R_{2_1} in BCNF? ✓
  - Ist R_{2_2} in BCNF? ✓

Damit erhalten wir unser Relationenschema in BCNF mit:

$$R_1 = \{A, \underline{B}, D\} \quad \wedge \quad \text{FD}_1 = \{B \to DA\}$$

$$R_{2_1} = \{\underline{C}, E\} \quad \wedge \quad \text{FD}_{2_1} = \{C \to E\}$$

$$R_{2_2} = \{\underline{B}, \underline{C}, \underline{F}\} \quad \wedge \quad \text{FD}_{2_1} \text{ trivial}$$

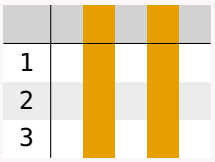
[a] $\beta \notin \alpha$ und B ist kein Superschlüssel.
[b] Hier geht $DEF \to B$ komplett verloren, und $C \to AE$ geht teilweise verloren (Dekompositionsalgorithmus).

6 SQL

6.1 Selektion, Projektion

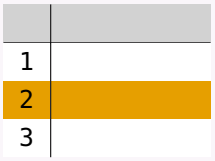
SQL-Bonus: Projektion in SQL

Eine *Projektion* filtert in SQL Spalten durch Angabe von Attributen einer Tabelle.



SQL-Bonus: Selektion in SQL

Eine *Selektion* filtert in SQL Zeilen durch Angabe eines Kriteriums, welches pro Zeile erfüllt sein muss.



SQL: SELECT

`SELECT` gibt alle Datensätze einer Tabelle aus.

Möchte man sich z.B. alle Pokémon aus der Tabelle `pokemon` anzeigen lassen, nutzt man:[a]

```
SELECT *
FROM pokemon;
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	1	Bisasam	0.7	6.9	1	Pflanze	Gift
2	2	Bisaknosp	1	13	1	Pflanze	Gift
3	3	Bisaflor	2	100	1	Pflanze	Gift
...				...			
898	898	Coronospa	1.1	7.7	8	Psycho	Pflanze

Möchte man eine *Projektion* auf die Ergebnisse der Abfrage anwenden, sich also nur bestimmte Spalten anzeigen lassen, kann man dies im `SELECT` spezifizieren:

```
SELECT
    ID,
    Name,
    PrimaerTyp,
    SekundaerTyp
FROM pokemon;
```

	ID	Name	PrimaerTyp	SekundaerTyp
1	1	Bisasam	Pflanze	Gift
2	2	Bisaknosp	Pflanze	Gift
3	3	Bisaflor	Pflanze	Gift
...			...	
898	898	Coronospa	Psycho	Pflanze

Die Reihenfolge der Datensätzer ist hierbei nicht vorgegeben.

[a] Der $*$ dient als Platzhalter für alle Attribute

SQL: WHERE

Möchte man eine *Selektion* auf die Ergebnisse der Abfrage anwenden, also für jedes Element eine Bedingung voraussetzen, nutzt man `WHERE`.

Beispielsweise werden im folgendem Befehl alle Pokémon mit dem Typ Feuer ausgegeben:

```sql
SELECT *
FROM pokemon
WHERE
    PrimaerTyp = 'Feuer'
    OR SekundaerTyp = 'Feuer';
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	4	Glumanda	0.6	8.5	1	Feuer	NULL
2	5	Glutexo	1.1	19	1	Feuer	NULL
3	6	Glurak	1.7	90.5	1	Feuer	Flug
...					...		
71	851	Infernopod	3	120	8	Feuer	Kaefer

SQL: AS

Um komplexere Ausdrücke zu kürzen oder Zweideutigkeiten aufzulösen, kann man Tabellen mit einem *Alias* versehen.

Beispielsweise wäre im folgenden Befehl die Tabelle `pokemon` unter `p` erreichbar:

```sql
SELECT p.*
FROM pokemon AS p;
```

Das Schlüsselwort `AS` ist optional, d.h folgende Abfrage ist äquivalent:

```sql
SELECT p.*
FROM pokemon p;
```

SQL: ORDER BY

`ORDER BY` sortiert die Ergebnismenge nach Kriterien auf- (`ASC`) oder absteigend (`DESC`).

Sollten mehrere Kriterien angegeben sein, so wird die Menge nach dem ersten gegebenen Kriterium sortiert. Wenn nach diesem Schritt mehrere Einträge dieses Kriterium gleichermaßen erfüllen, wird diese Untermenge nach dem zweiten Kriterium sortiert, usw.

`NULL` ist immer der erste Eintrag.

Wenn man sich also alle Pokémon nach Namen sortiert ausgeben möchte, so erhält man[a]:

```sql
SELECT *
FROM pokemon
ORDER BY Name;
```

```sql
SELECT *
FROM pokemon
ORDER BY Name ASC;
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	367	Aalabyss	1.7	27	3	Wasser	NULL
2	63	Abra	0.9	19.5	1	Psycho	NULL
3	359	Absol	1.2	47	3	Unlicht	NULL
...					...		
898	579	Zytomega	1	20.1	5	Psycho	NULL

Wenn man alternativ alle Pokémon erst nach Primärtyp (`ASC`), dann nach Sekundärtyp (`DESC`) sortiert ausgeben möchte, erhält man:

```sql
SELECT *
FROM pokemon
ORDER BY
    PrimaerTyp ASC,
    SekundaerTyp DESC;
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	551	Ganovil	0.7	15.2	5	Boden	Unlicht
2	552	Rokkaiman	1	33.4	5	Boden	Unlicht
3	553	Rabigator	1.5	96.3	5	Boden	Unlicht
...					...		
898	224	Octillery	0.9	28.5	2	Wasser	NULL

[a] ASC ist dabei der Default-Wert

6.1.1 Funktionen

SQL: CONCAT

`CONCAT` kombiniert verschiedene Strings zu einem.

In folgendem Beispiel werden Primärtyp und Sekundärtyp zu einem Eintrag zusammengefasst:

```sql
SELECT
    ID,
    Name,
    CONCAT('[', PrimaerTyp, ', ', SekundaerTyp, ']') AS 'Typ'
FROM pokemon
    WHERE SekundaerTyp IS NOT NULL;
```

	ID	Name	Typ
1	1	Bisasam	[Pflanze, Gift]
2	2	Bisaknosp	[Pflanze, Gift]
3	3	Bisaflor	[Pflanze, Gift]
...		...	
242	898	Coronospa	[Psycho, Pflanze]

SQL: ROUND

`ROUND` rundet numerische Werte. Der zweite Parameter gibt die gewünschte Anzahl an Nachkommastellen an.

In folgendem Beispiel wird das Gewicht eines Pokémon auf ganzzahlige Werte gerundet:

```sql
SELECT
    ID,
    Name,
    Gewicht,
    ROUND(Gewicht, 0)
FROM pokemon;
```

	ID	Name	Gewicht	ROUND(Gewicht, 0)
1	1	Bisasam	6.9	7
2	2	Bisaknosp	13	13
3	3	Bisaflor	100	100
...			...	
898	898	Coronospa	7.7	8

SQL: COUNT

`COUNT` zählt alle nicht-`NULL` Datensätze. Sollte $*$ angegeben werden, beachtet er alle Entitäten, welche mindestens ein nicht-`NULL` Attribut besitzen.

Dadurch, dass `ID` und `PrimaerTyp` nicht `NULL` sein können, erhält man mit folgenden äquivalenten Befehlen die Anzahl aller Pokémon:

```sql
SELECT COUNT(*)
FROM pokemon;
```

```sql
SELECT COUNT(ID)
FROM pokemon;
```

```sql
SELECT COUNT(PrimaerTyp)
FROM pokemon;
```

	COUNT(*)
1	898

Möchten wir nun alle Pokémon mit einem Sekundärtypen zählen, so erhalten wir:

```sql
SELECT COUNT(SekundaerTyp)
FROM pokemon;
```

	COUNT(SekundaerTyp)
1	442

SQL: CASE

`CASE` führt die erste passende `WHEN`-Klausel aus.

Sollte uns die Spalte `Generation` in `pokemon` verloren gehen, könnten wie diese wir folgt nachbauen:

```sql
SELECT
    ID,
    Name,
    (
        CASE
            WHEN ID >= 810 THEN 8
            WHEN ID >= 722 THEN 7
            WHEN ID >= 650 THEN 6
            WHEN ID >= 494 THEN 5
            WHEN ID >= 387 THEN 4
            WHEN ID >= 252 THEN 3
            WHEN ID >= 152 THEN 2
            ELSE 1
        END
    ) AS Generation_Case
FROM pokemon;
```

6.1.2 Operatoren

SQL: IS NULL

`IS NULL` bzw. `IS NOT NULL` werden genutzt, um mit `NULL`-Einträgen zu arbeiten.

In der folgenden Abfrage werden Pokémon mit exakt einem Typen zurückgegeben:

```sql
SELECT
    ID,
    Name,
    PrimaerTyp,
    SekundaerTyp
FROM pokemon
    WHERE SekundaerTyp IS NULL;
```

	ID	Name	PrimaerTyp	SekundaerTyp
1	4	Glumanda	Feuer	NULL
2	5	Glutexo	Feuer	NULL
3	7	Schiggy	Wasser	NULL
...		...		
456	897	Phantoross	Geist	NULL

SQL: COALESCE

`COALESCE` gibt den ersten nicht-`NULL` Eintrag aus der Parameterliste zurück.

In diesem Beispiel wird aus der Tabelle `entwicklung` jede `NULL`-Tageszeit mit dem String `Immer` ersetzt:

```sql
SELECT
    Von,
    Zu,
    Tageszeit,
    COALESCE(Tageszeit, 'Immer') AS Entwicklungszeitraum
FROM entwicklung;
```

	Von	Zu	Tageszeit	Entwicklungszeitraum
1	1	2	NULL	Immer
2	2	3	NULL	Immer
3	4	5	NULL	Immer
...			...	
85	133	136	NULL	Immer
86	133	196	Tag	Tag
87	133	197	Nacht	Nacht
88	133	470	NULL	Immer
...			...	
429	891	892	NULL	Immer

SQL: Bedingungen

Bedingungen werden von Selektionen, sowie von `IF`-Abfragen genutzt:

```
SELECT *
FROM <Tabelle>
WHERE <Bedingung>;
```

```
SELECT IF(
    <Bedingung>,
    <Wert wenn TRUE>,
    <Wert wenn FALSE>
);
```

Verschiedene Bedingungen lassen sich mit `AND` und `OR` verbinden, sowie mit `NOT` negieren.

Operatoren, welche für die meisten Datentypen definiert sind:

- `=` überprüft auf exakte Gleichheit.
- `<>` überprüft auf Ungleichheit.
- `IN` überprüft, ob der Wert in einer angegebenen Liste zu finden ist.

Explizit für String Werte ist definiert:

- `LIKE`: Benutzt reguläre Ausdrücke zum Vergleichen:
  - `%` lässt beliebig viele beliebige Zeichen zu.
  - `_` lässt exakt ein beliebiges Zeichen zu.

Des Weiteren kann man für numerische Werte folgende Operatoren nutzen:

- `<, <=, >, >=`
- `BETWEEN X AND Y` überprüft, ob der Wert zwischen den angegebenen Grenzen `X` und `Y` (beide inklusive) liegt.

Achtung: Ausdrücke sind `NULL`, wenn einer der Operanden `NULL` ist.

SQL: DISTINCT

`DISTINCT` berücksichtigt keine doppelten Daten.

Wenn man sich jede vorhandene Kombination an Typen anzeigen lassen möchte, so erhält man in folgender Abfrage doppelte Werte:

```
SELECT
    PrimaerTyp,
    SekundaerTyp
FROM pokemon;
```

	PrimaerTyp	SekundaerTyp
1	Pflanze	Gift
2	Pflanze	Gift
3	Pflanze	Gift
...	...	...
898	Psycho	Pflanze

Diese entfallen, wenn man das Schlüsselwort `DISTINCT` ergänzt:

```
SELECT DISTINCT
    PrimaerTyp,
    SekundaerTyp
FROM pokemon;
```

	PrimaerTyp	SekundaerTyp
1	Pflanze	Gift
2	Feuer	NULL
3	Feuer	Flug
...	...	...
175	Unlicht	Pflanze

6.1.3 Beispiele

Beispiel: SELECT

Ermitteln Sie, wieviele verschiedene Typkombinationen in der Tabelle `pokemon` vorhanden sind und geben Sie die Anzahl aus.

```
SELECT COUNT(
    DISTINCT PrimaerTyp,
    COALESCE(SekundaerTyp, 'NULL')
)
FROM pokemon;
```

	COUNT(DISTINCT PrimaerTyp, COALESCE(SekundaerTyp, 'NULL'))
1	175

Beispiel: SELECT

Ermitteln Sie den BMI[a] aller Pokémon, welche mehr als 50kg wiegen.

Runden Sie das Ergebnis auf 2 Nachkommastellen.

```sql
SELECT
    ID,
    Name,
    Gewicht,
    Groesse,
    ROUND(Gewicht / (POW(Groesse, 2)), 2) AS BMI
FROM pokemon
WHERE Gewicht > 50
ORDER BY BMI DESC;
```

	ID	Name	Gewicht	Groesse	BMI
1	790	Cosmovum	999.9	0.1	99990.00
2	304	Stollunior	60	0.4	375.00
3	366	Perlu	52.5	0.4	328.12
...			...		
40	143	Relaxo	460	2.1	104.31
...			...		
288	321	Wailord	398	14.5	1.89

[a] $\text{BMI} = \frac{\text{Gewicht}}{\text{Größe}^2}$

Beispiel: SELECT

Wieviele Pokémon-Namen beginnen mit einem `P`?

Und wieviele enthalten ein `P`?

Ermitteln Sie die jeweilige Anzahl.

```sql
SELECT COUNT(*)
FROM pokemon
WHERE Name LIKE 'P%';
```

	COUNT(*)
1	73

```sql
SELECT COUNT(*)
FROM pokemon
WHERE Name LIKE '%P%';
```

	COUNT(*)
1	195

6.2 Join

SQL-Bonus: Kartesisches Produkt in SQL

Wenn man alle Entitäten einer Tabelle mit jeder Entität aus einer anderen Tabelle kombinieren möchte, benötigt man das *kartesische Produkt*.

Möchte man jede mögliche Typkombination[a] erhalten, nutzt man `SELECT` zwei mal auf `Typ`:

```
SELECT *
FROM
    typ AS typ1,
    typ AS typ2;
```

	typ1.Bezeichnung	typ2.Bezeichnung
1	Boden	Boden
2	Drache	Boden
3	Eis	Boden
...	...	
19	Boden	Drache
20	Drache	Drache
21	Eis	Drache
...	...	
324	Wasser	Wasser

Wenn eine Relation verschiedener Tabellen über einen Fremdschlüssel realisiert ist, kann man Entitäten aus diesen Tabellen zueinander zuordnen. So wird nicht nur der Fremdschlüssel angezeigt, sondern alle zugehörigen Attribute.

Da wir die aus dem kartesischen Produkt erhaltene Menge noch sinnvoll selektieren müssen, nutzen wir `WHERE`.

In unserem Beispiel kann kein Pokemon zwei mal den selben Typen besitzen:

```
SELECT *
FROM
    typ AS typ1,
    typ AS typ2
WHERE NOT typ1.Bezeichnung = typ2.Bezeichnung;
```

	typ1.Bezeichnung	typ2.Bezeichnung
1	Drache	Boden
2	Eis	Boden
3	Elektro	Boden
...	...	
18	Boden	Drache
19	Eis	Drache
20	Elektro	Drache
...	...	
306	Unlicht	Wasser

[a] Sollte man das kartesische Produkt einer Tabelle mit sich selbst bilden, benötigt man Aliasse für die Tabellen

SQL: JOIN

`JOIN` bestimmt ebenfalls das kartesische Produkt zweier Tabellen.[a]

Das Keyword `ON` verknüpft dann die beiden Tabellen anhand einer Bedingung.[b]

```sql
SELECT *
FROM typ AS typ1
JOIN typ AS typ2
    ON NOT typ1.Bezeichnung = typ2.Bezeichnung;
```

[a] `JOIN` und `SELECT` von mehreren Tabellen unterscheiden sich ausschließlich in der Lesbarkeit und Optimierungsmöglichkeiten für das DBMS.
Bei `JOIN` muss das DBMS nicht alle Kombinationen ermitteln, nur um diese dann zu filtern.

[b] Entspricht einem Theta-Join.

SQL: CROSS JOIN

`CROSS JOIN` gibt das unselektierte kartesische Produkt aus.

Dementsprechend sind die folgenden Abfragen erneut äquivalent:

```sql
SELECT *
FROM typ AS typ1
CROSS JOIN typ AS typ2;
```

```sql
SELECT *
FROM typ AS typ1
JOIN typ as typ2
    ON TRUE;
```

```sql
SELECT *
FROM
    typ AS typ1,
    typ AS typ2;
```

SQL: INNER JOIN

`INNER JOIN` wird automatisch genutzt, wenn man `JOIN` nutzt.

Dementsprechend sind folgende abfragen äquivalent:

```sql
SELECT *
FROM typ AS typ1
JOIN typ AS typ2
    ON NOT typ1.Bezeichnung = typ2.Bezeichnung;
```

```sql
SELECT *
FROM typ AS typ1
INNER JOIN typ AS typ2
    ON NOT typ1.Bezeichnung = typ2.Bezeichnung;
```

Wenn man sich beispielsweise alle Pokémon samt ihrer Entwicklung anzeigen lassen möchte, so erhält man:

```sql
SELECT
    pokemon.ID,
    pokemon.Name,
    entwicklung.Zu
FROM pokemon
JOIN entwicklung
    ON pokemon.ID = entwicklung.Von;
```

	ID	Name	Zu
1	1	Bisasam	2
2	2	Bisaknosp	3
3	4	Glumanda	5
...	...		
429	891	Dakuma	892

SQL-Bonus: SELF JOIN

Als `SELF JOIN` werden `JOIN`s bezeichnet, welche eine reflexive Fremdschlüsselbeziehung abfragen.

Hierbei ist zu beachten, dass `SELF` kein Keyword ist, sondern die reine Bezeichnung für solche Relationen.

Dies wurde mit dem Beispiel `typ` auf `typ` bereits dargestellt.

SQL-Bonus: OUTER JOIN

`OUTER JOIN` berücksichtigt ebenfalls Entitäten, für die kein passendes Pendant existiert. Für diese Entitäten sind alle Attribute der jeweils anderen Entitätenmenge `NULL`.

`OUTER JOIN` ist keine gültige Abfrage. Das Keyword `OUTER` benötigt immer eine Leserichtung.

Wenn wir erneut das Beispiel von `INNER JOIN` betrachten, sehen wir, dass das Pokémon mit der ID 3 nicht vorkommt. Dies ist dem Fakt geschuldet, dass dieses Pokémon keine Entwicklung besitzt. Sollte man dieses Pokemon trotzdem angezeigt bekommen, benötigt man einen `OUTER JOIN`.

SQL: LEFT OUTER JOIN

`LEFT OUTER JOIN` beachtet alle Entitäten *links* der Relation.

Nun werden alle Pokémon dargestellt, ggf. mit `NULL`-Attributen bei Entwicklung:

```sql
SELECT
    pokemon.ID,
    pokemon.Name,
    entwicklung.Von,
    entwicklung.Zu
FROM pokemon
LEFT OUTER JOIN entwicklung
    ON pokemon.id = entwicklung.von;
```

	ID	Name	Von	Zu
1	1	Bisasam	1	2
2	2	Bisaknosp	2	3
3	3	Bisaflor	NULL	NULL
...		...		
920	898	Coronospa	NULL	NULL

SQL: RIGHT OUTER JOIN

`RIGHT OUTER JOIN` beachtet alle Entitäten *rechts* der Relation.

Durch Vertauschen von Entitätstypen und Leserichtung erhält man äquivalente Befehle.

So würde unser Beispiel aus `LEFT OUTER JOIN` folgendermaßen aussehen:

```sql
SELECT
    pokemon.ID,
    pokemon.Name,
    entwicklung.Von,
    entwicklung.Zu
FROM entwicklung
RIGHT OUTER JOIN pokemon
    ON pokemon.id = entwicklung.von;
```

SQL: FULL OUTER JOIN

`FULL OUTER JOIN` ist eine Vereinigung von `LEFT OUTER JOIN` und `RIGHT OUTER JOIN`. Das bedeutet, dass beidseitig alle Entitäten betrachtet werden, ggf. mit `NULL` aufgefüllt[a].

```sql
SELECT
    pokemon.ID,
    pokemon.Name,
    entwicklung.Von,
    entwicklung.Zu
FROM pokemon
FULL OUTER JOIN entwicklung
    ON pokemon.id = entwicklung.von;
```

```sql
SELECT
    pokemon.ID,
    pokemon.Name,
    entwicklung.Von,
    entwicklung.Zu
FROM pokemon
LEFT OUTER JOIN entwicklung
    ON pokemon.id = entwicklung.von
UNION
SELECT
    pokemon.ID,
    pokemon.Name,
    entwicklung.Von,
    entwicklung.Zu
FROM pokemon
RIGHT OUTER JOIN entwicklung
    ON pokemon.id = entwicklung.von;
```

[a] Manche DBMS bieten keinen Befehl für einen `FULL OUTER JOIN`. In diesem Fall vereinigt man die Ergebnisse der beiden `OUTER JOIN`s mit `UNION`.

SQL-Bonus: NATURAL JOIN

Von der Verwendung von `NATURAL JOIN`s wird abgeraten!

Bei Tabellen mit gleichnamigen Attributen kann auf die explizite Angabe der Bedingung verzichtet werden. Das DBMS verwendet dann *alle* gleichnamigen Attribute.

6.2.1 Beispiele

Beispiel: JOIN

Geben Sie für alle Pokemon jeweils die Entwicklung mit Level, sowie die Typen aus.

Nutzen Sie:

a) einen `INNER JOIN` mit `WHERE`
b) einen `INNER JOIN` mit `JOIN`

Fügen Sie noch den Namen, sowie die Typen der Entwicklung an.

---

a) `WHERE`

```sql
SELECT
    von.ID,
    von.Name,
    von.PrimaerTyp,
    von.SekundaerTyp,
    entwicklung.Level,
    zu.ID,
    zu.Name
FROM
    pokemon AS von,
    pokemon AS zu,
    entwicklung
WHERE
    von.ID = entwicklung.Von
    AND entwicklung.Zu = zu.ID;
```

b) `JOIN`

```sql
SELECT
    von.ID,
    von.Name,
    von.PrimaerTyp,
    von.SekundaerTyp,
    entwicklung.Level,
    zu.ID,
    zu.Name
FROM pokemon
AS von
JOIN entwicklung
    ON von.ID = entwicklung.Von
JOIN pokemon AS zu
    ON entwicklung.Zu = zu.ID;
```

	von.ID	von.Name	PrimaerTyp	SekundaerTyp	Level	zu.ID	zu.Name
1	1	Bisasam	Pflanze	Gift	16	2	Bisaknosp
2	2	Bisaknosp	Pflanze	Gift	32	3	Bisaflor
3	4	Glumanda	Feuer	NULL	16	5	Glutexo
...				...			
429	891	Dakuma	Kampf	NULL	NULL	892	Wulaosu

Beispiel: JOIN

Geben Sie zusätzlich auch Pokémon aus, die (noch) keine Entwicklung haben.

```sql
SELECT
    von.ID,
    von.Name,
    von.PrimaerTyp,
    von.SekundaerTyp,
    entwicklung.Level,
    zu.ID,
    zu.Name
FROM pokemon AS von
LEFT OUTER JOIN entwicklung
    ON von.ID = entwicklung.Von
LEFT OUTER JOIN pokemon AS zu
    ON entwicklung.Zu = zu.ID;
```

	von.ID	von.Name	PrimaerTyp	SekundaerTyp	Level	zu.ID	zu.Name
1	1	Bisasam	Pflanze	Gift	16	2	Bisaknosp
2	2	Bisaknosp	Pflanze	Gift	32	3	Bisaflor
3	3	Bisaflor	Pflanze	Gift	NULL	NULL	NULL
...				...			
920	898	Coronospa	Psycho	Pflanze	NULL	NULL	NULL

Beispiel: JOIN

Geben Sie alle Pokémon aus, die bislang keine Entwicklung besitzen.

```sql
SELECT
    pokemon.ID,
    pokemon.Name,
    pokemon.PrimaerTyp,
    pokemon.SekundaerTyp
FROM pokemon
LEFT OUTER JOIN entwicklung
    ON pokemon.ID = entwicklung.Von
WHERE entwicklung.Zu IS NULL;
```

	ID	Name	PrimaerTyp	SekundaerTyp
1	3	Bisaflor	Pflanze	Gift
2	6	Glurak	Feuer	Flug
3	9	Turtok	Wasser	NULL
...		...		
491	898	Coronospa	Psycho	Pflanze

Beispiel: JOIN

Geben Sie alle Pokémon aus, die sich aus Evoli entwickeln.

```sql
SELECT
    von.ID,
    von.Name,
    zu.ID,
    zu.Name,
    zu.PrimaerTyp,
    zu.SekundaerTyp
FROM pokemon AS zu
JOIN entwicklung
    ON zu.ID = entwicklung.Zu
JOIN pokemon AS von
    ON entwicklung.Von = von.ID
WHERE von.Name = 'Evoli';
```

	von.ID	von.Name	zu.ID	zu.Name	PrimaerTyp	SekundaerTyp
1	133	Evoli	134	Aquana	Wasser	NULL
2	133	Evoli	135	Blitza	Elektro	NULL
3	133	Evoli	136	Flamara	Feuer	NULL
4	133	Evoli	196	Psiana	Psycho	NULL
5	133	Evoli	197	Nachtara	Unlicht	NULL
6	133	Evoli	470	Folipurba	Pflanze	NULL
7	133	Evoli	471	Glaziola	Eis	NULL
8	133	Evoli	700	Feelinara	Fee	NULL

6.3 Group Functions

SQL: GROUP BY

`GROUP BY` teilt die Gesamtmenge in Teilmengen bzw. Gruppierungen auf.

Auf diese Teilmengen können ausschließlich Methoden speziell für Datensätze, wie beispielsweise `COUNT`, aufgerufen werden.

Attributwerte, die sich innerhalb einer Teilmenge unterscheiden, können nicht direkt aufgerufen werden.[a]

Gibt man bei dem Keyword `GROUP BY` mehrere Attribute an, wird der Datensatz nach dem Tupel dieser gruppiert. Dabei wird zunächst nach dem ersten Attribut gruppiert, folgend innerhalb dieser Teilgruppen dem Zweiten, etc.

[a] Da es in der Tabelle `attacke` Attacken mit dem selben `Namen` gibt, welche sich insbesondere durch ihre `Schadensklasse` unterscheiden, kann man, wenn man nach dem `Namen` gruppiert, die jeweilige `Schadensklasse` nicht aufrufen. Je nach DBMS ist es trotzdem möglich die gleichbleibenden Attribute `Name`, `Typ`, `Stärke`, `Genauigkeit`, `AP` und `Generation` zu verwenden.

SQL: HAVING

`HAVING` filtert die mit `GROUP BY` gruppierten Teilmengen.

Dabei werden Bedingungen auf das Ergebnis der Abfrage angewandt.

Gibt man sich beispielsweise alle Typen aus, welche mindestens von 25 Attacken genutzt werden, so erhält man:

```sql
SELECT
    Typ,
    COUNT(*) AS Anzahl
FROM attacke
GROUP BY Typ
HAVING Anzahl >= 25
ORDER BY Anzahl DESC;
```

	Typ	Anzahl
1	Normal	188
2	Psycho	69
3	Pflanze	52
...	...	
17	Drache	27

6.3.1 Funktionen

SQL: MIN, MAX

`MIN` gibt den alphanumerisch ersten Eintrag eines Datensatzes zurück.

`MAX` gibt den alphanumerisch letzten Eintrag eines Datensatzes zurück.

SQL: AVG

`AVG` gibt den durchschnittlichen Zahlwert eines Datensatzes zurück.

SQL: SUM

`SUM` gibt die Summe alles Zahlenwerte eines Datensatzes zurück

6.3.2 Beispiele

Beispiel: GROUP BY

```sql
SELECT
    typ,
    MIN(Staerke),
    MAX(Staerke),
    SUM(Staerke),
    AVG(Staerke)
FROM attacke
GROUP BY typ;
```

	typ	COUNT(*)	MIN(Staerke)	MAX(Staerke)	SUM(Staerke)	AVG(Staerke)
1	Boden	29	10	120	1365	68.2500
2	Drache	27	10	185	2005	91.1364
3	Eis	29	10	140	1575	71.5909
4	Elektro	43	10	210	2740	88.3871
5	Fee	30	10	190	1250	89.2857
6	Feuer	40	35	180	3370	96.2857
7	Flug	30	10	140	1735	75.4348
8	Geist	30	10	200	1815	90.7500
9	Gestein	23	10	190	1290	80.6250
10	Gift	31	10	120	1080	63.5294
11	Käfer	32	10	120	1340	63.8095
12	Kampf	51	10	150	2810	75.9459
13	Normal	188	10	250	5441	72.5467
14	Pflanze	52	10	150	2560	77.5758
15	Psycho	69	10	200	2385	91.7308
16	Stahl	32	10	200	1815	86.4286
17	Unlicht	47	10	180	1950	72.2222
18	Wasser	42	10	195	2635	77.5000

Beispiel: GROUP BY

Geben Sie alle Attacken aus mit einer echter Stärke (> 0) geordnet aus. Gruppieren Sie diese nach Typ und ermitteln Sie die `MIN` und `MAX` Stärke. Ergänzen Sie diese um die Durchschnittsstärke und geben Sie nur die Einträge aus, bei denen diese über 75 liegt.

```sql
SELECT
    Typ,
    COUNT(*),
    MIN(Staerke),
    MAX(Staerke),
    AVG(Staerke)
FROM attacke
WHERE Staerke > 0
GROUP BY Typ
HAVING AVG(Staerke) > 75
ORDER BY AVG(Staerke) DESC;
```

	Typ	COUNT(*)	MIN(Staerke)	MAX(Staerke)	AVG(Staerke)
1	Feuer	35	35	180	96.2857
2	Psycho	26	10	200	91.7308
3	Drache	22	10	185	91.1364
...			...		
12	Flug	23	10	140	75.4348

Beispiel: GROUP BY

Errechnen Sie

- Erwartungswert (μ, E),
- Varianz (σ^2, Var), sowie
- Standardabweichung (σ, SD)

der jeweiligen nach Typ gruppierten Attacken.

```sql
SELECT
    Typ,
    ROUND(AVG(Staerke), 3) AS E,
    ROUND(AVG(POW(Staerke, 2)) - POW(AVG(Staerke), 2), 3) AS Var,
    ROUND(SQRT(AVG(POW(Staerke, 2)) - POW(AVG(Staerke), 2)), 3) AS SD
FROM attacke
GROUP BY Typ
ORDER BY SD;
```

	Typ	μ	σ^2	σ
1	Käfer	63.810	747.395	27.339
2	Boden	68.250	795.688	28.208
3	Unlicht	72.222	859.88	29.324
...		...		
18	Geist	90.750	2633.188	51.315

Beispiel: GROUP BY

Gruppieren Sie alle Pokémon nach Generation. Geben Sie diese Teilmengen sortiert nach Anzahl aus. Ermitteln Sie zusätzlich das gerundete Durchschnittsgewicht der Pokémon jeder Generation.

```sql
SELECT
    Generation, COUNT(*), ROUND(AVG(Gewicht), 3)
FROM pokemon
GROUP BY Generation
ORDER BY COUNT(*);

```

	Generation	COUNT(*)	ROUND(AVG(Gewicht), 3)
1	6	72	51.401
2	7	88	109.661
3	8	89	76.267
...		...	
8	5	156	52.403

6.4 Subselects

Definition: Unterabfrage

Unterabfragen bzw. Subselects sind verschachtelte Abfragen. Dabei stellen die inneren Abfragen Daten bereit, welche von den Äußeren genutzt werden.

Unterabfragen werden genutzt, wenn die zu vergleichenden Werte sich dynamisch verändern oder noch unbekannt sind.

Unterabfragen lassen sich in verschiedene Kategorien aufteilen:

- Einzeilige bzw. mehrzeilige Unterabfragen
- Korrelierte bzw. unkorrelierte Unterabfragen
- Skalare bzw. nicht skalare Unterabfragen

Wenn man von Unterabfragen spricht, so meint man meist nicht korrelierte, skalare Unterabfragen, da diese die simpelste Implementierung sind.

Beispiel: Unterabfrage

So könnte man z.B. alle IDs von Pokémon, welche sich aus Glumanda entwickeln, folgendermaßen herausfinden:

Zuerst müssen wir die ID von Glumanda herausfinden:

```sql
-- ID von 'Glumanda' herausfinden
SELECT ID
FROM pokemon
WHERE Name = 'Glumanda';
```

	ID
1	4

Diese nutzen wir nun manuell um alle Entwicklungen herauszufinden:

```sql
-- ID von 'Glumanda' nutzen um alle Entwicklungen herauszufinden
SELECT *
FROM entwicklung
WHERE Von = 4;
```

	Von	Zu	Level	Item	GetragenesItem	Tageszeit
1	4	5	16	NULL	NULL	NULL

Diese beiden Abfragen lassen sich nun zu einer Abfrage zusammenfassen:

```sql
-- ID von 'Glumanda' nutzen um alle Entwicklungen herauszufinden
SELECT *
FROM entwicklung
WHERE Von = (
    -- ID von 'Glumanda' herausfinden
    SELECT ID
    FROM pokemon
    WHERE Name = 'Glumanda'
);
```

Definition: Mehrzeilige Unterfrage

In *mehrzeiligen Unterabfragen* geben die inneren Abfragen mehrere Entitäten zurück.

So kann man z.B. nicht auf exakte Gleichheit überprüfen, sondern muss Mengenoperationen (wie z.B. `IN`) nutzen.

Beispiel: Unterabfrage (mehrzeilig)

Das Resultat des ersten Beispiels nutzen wir nun um den Namen der Entwicklung herauszufinden:

```sql
SELECT *
FROM pokemon
WHERE ID = (
    -- Eine Entitaet
    SELECT Zu
    FROM entwicklung
    WHERE Von = (
        SELECT ID
        FROM pokemon
        WHERE Name = 'Glumanda'
    )
);
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	5	Glutexo	1.1	19	1	Feuer	NULL

Sollten wir nun alternativ die Entwicklungen von Evoli anzeigen lassen, können wir die Bedingung = nicht benutzen, da die Abfrage aus `entwicklung` eine Menge an Entitäten zurück gibt. Dementsprechend muss man das bekannte `IN` verwenden.

```sql
SELECT *
FROM pokemon
WHERE ID IN (
    -- Eine Menge an Entitaeten
    SELECT Zu
    FROM entwicklung
    WHERE Von = (
        SELECT ID
        FROM pokemon
        WHERE Name = 'Evoli'
    )
);
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	134	Aquana	1	29	1	Wasser	NULL
2	135	Blitza	0.8	24.5	1	Elektro	NULL
3	136	Flamara	0.9	25	1	Feuer	NULL
4	196	Psiana	0.9	26.5	2	Psycho	NULL
5	197	Nachtara	1	27	2	Unlicht	NULL
6	470	Folipurba	1	25.5	4	Pflanze	NULL
7	471	Glaziola	0.8	25.9	4	Eis	NULL
8	700	Feelinara	1	23.5	6	Fee	NULL

Definition: Korrelierte Unterabfrage

In *korrelierten Unterabfragen* („Correlated Subselects“) verweist die innere Abfragen auf die Äußere.[a]

Da eine Bedingung für jede Entität neu überprüft werden muss, haben korrelierte Unterabfragen unter Umständen eine längere Laufzeit.

Bei nicht korrelierten Unterabfragen wird eine Bedingung zu Beginn der Abfrage überprüft. Dementsprechend wird hier die Laufzeit optimiert.

[a] Dies ist vergleichbar mit einer Variable, welche in einem inneren Scope erneut genutzt wird.

Beispiel: Unterabfrage (korreliert)

In folgender Abfrage wird jede Attacke ausgegeben, welche eine höhere Stärke besitzt, als der Durchschnitt aller Attacken mit dem selben Typ.

```sql
SELECT
    ID,
    Name,
    Staerke,
    Typ
FROM attacke AS attacke1
WHERE Staerke > (
    SELECT AVG(Staerke)
    FROM attacke AS attacke1
    -- attacke2.Typ unterscheidet sich potentiell fuer jede Entitaet
    WHERE attacke1.Typ = attacke2
);
```

	ID	Name	Staerke	Typ
1	5	Megahieb	80	Normal
2	8	Eishieb	75	Eis
3	13	Klingensturm	80	Normal
...		...		
230	825	Astralfragmente	120	Geist

Definition: Nicht skalare Unterabfrage

In *nicht skalaren Unterabfragen* werden mehrere Attribute der Entitäten in der Unterabfrage genutzt. So müssen mehrere Tupel auf Gleichheit überprüft werden.

Sollte eine Unterabfrage nicht einzeilig sein, so ist diese automatisch nicht skalar. Dementsprechend ist eine Unterabfrage ausschließlich dann skalar, wenn man exakt eine Entität mit exakt einem Attribut erhält.

Beispiel: Unterabfrage (nicht skalar)

Möchte man sich beispielsweise alle Pokémon ausgeben, welche exakt die selbe Typkombination wie Bisasam haben, so nutzt man:

```sql
SELECT *
FROM pokemon
WHERE
    (PrimaerTyp, SekundaerTyp) = (
        -- Entspricht einem Tupel aus 2 Typen
        -- Obacht: Keiner der Typen darf NULL sein
        SELECT
            PrimaerTyp,
            SekundaerTyp
        FROM pokemon
        WHERE Name = 'Bisasam'
    )
    AND Name <> 'Bisasam';
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	2	Bisaknosp	1	13	1	Pflanze	Gift
2	3	Bisaflor	2	100	1	Pflanze	Gift
3	43	Myrapla	0.5	5.4	1	Pflanze	Gift
...				...			
13	591	Hutsassa	0.6	10.5	5	Pflanze	Gift

6.4.1 Funktionen

SQL: ANY

- `< ANY` gibt Einträge zurück, welche kleiner als ein beliebiges Element[a] einer Menge sind.
- `> ANY` gibt Einträge zurück, welche größer als ein beliebiges Element[b] einer Menge sind.
- `= ANY` gibt Einträge zurück, welche mit einem Werte einer Menge übereinstimmen. Diese Bedingung entspricht dem Keyword `IN`.

[a] d.h. kleiner als das Maximum
[b] d.h. größer als das Minimum

SQL: ALL

- `< ALL` gibt Einträge zurück, welche kleiner als alle Elemente[a] einer Menge sind.
- `> ALL` gibt Einträge zurück, welche größer als alle Elemente[b] einer Menge sind.

[a] d.h. kleiner als das Minimum
[b] d.h. größer als das Maximum

SQL: EXISTS

Bedingungen mit `EXISTS` sind erfüllt, sollte bei einer Unterabfrage eine nicht leere Menge zurückgegeben werden.

`EXISTS` wird meist mit korrelierenden Unterabfragen genutzt.

SQL-Bonus: INSERT INTO mit Unterabfragen

Wenn man automatisch Werte aus bestehenden Tabellen in neue Tabellen einfügen möchte, kann man Unterabfragen nutzen.

Hierbei entfällt das Keyword `VALUES`:

```sql
CREATE TABLE feuerpokemon (
    ID int NOT NULL,
    Bezeichnung varchar(255),
    ...
);
INSERT INTO feuerpokemon
    -- Unterabfrage
    SELECT *
    FROM pokemon
    WHERE
        PrimaerTyp = 'Feuer'
        OR SekundaerTyp = 'Feuer';
```

6.4.2 Beispiele

Beispiel: Unterabfrage

Welche Kombinationen von Typen werden von mehr Pokémon genutzt, als die Kombination `('Feuer', NULL)`?

```sql
SELECT
    PrimaerTyp,
    SekundaerTyp
FROM pokemon
GROUP BY
    PrimaerTyp,
    SekundaerTyp
HAVING COUNT(*) > (
        -- COUNT(*) = 33
        SELECT COUNT(*)
        FROM pokemon
        WHERE
            PrimaerTyp = 'Feuer'
            AND SekundaerTyp IS NULL
        GROUP BY
            PrimaerTyp,
            SekundaerTyp
);
```

	PrimaerTyp	SekundaerTyp
1	Wasser	NULL
2	Normal	NULL
3	Psycho	NULL
4	Pflanze	NULL

Beispiel: Unterabfrage

Geben Sie alle Typen aus, welche nicht as Sekundärtyp in Kombination mit dem Primärtypen `'Feuer'` für mindestens ein Pokémon auftreten.

```sql
SELECT *
FROM typ
WHERE NOT EXISTS (
    SELECT *
    FROM pokemon
    WHERE
        PrimaerTyp = 'Feuer'
        AND SekundaerTyp = typ.bezeichnung
);
```

	Bezeichnung
1	Eis
2	Elektro
3	Fee
...	...
6	Pflanze

Beispiel: Unterabfrage

Geben Sie alle Pokémon aus, welche exakt die selbe Typkombination wie Glumanda, also `('Feuer', NULL)`, haben.

Da der Sekundärtyp von Glumanda `NULL` ist, muss man das Beispiel aus nicht skalaren Unterabfragen anpassen.

In der ersten, simplen, Lösung ersetzt man alle `NULL`-Sekundärtypen mit einem String:

```sql
SELECT *
FROM pokemon
WHERE
    (PrimaerTyp, COALESCE(SekundaerTyp, 'NULL')) = (
        SELECT
            PrimaerTyp,
            COALESCE(SekundaerTyp, 'NULL')
        FROM pokemon
        WHERE Name = 'Glumanda'
    )
    AND Name <> 'Glumanda';
```

Alternativ nutzt man korrelierende Unterabfragen:

```sql
SELECT *
FROM pokemon AS pokemon1
WHERE
    EXISTS (
        SELECT *
        FROM pokemon AS pokemon2
        WHERE
            Name = 'Glumanda'
            AND pokemon1.PrimaerTyp = pokemon2.PrimaerTyp
            AND (
                -- pokemon1 hat den selben Sekundaertyp wie pokemon2
                pokemon1.SekundaerTyp = pokemon2.SekundaerTyp
                -- der Sekundaertyp von pokemon1, sowie pokemon2 ist NULL
                OR (
                    pokemon1.SekundaerTyp IS NULL
                    AND pokemon2.SekundaerTyp IS NULL
                )
            )
    )
    AND Name <> 'Glumanda';
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	5	Glutexo	1.1	19	1	Feuer	NULL
2	37	Vulpix	0.6	9.9	1	Feuer	NULL
3	38	Vulnona	1.1	19.9	1	Feuer	NULL
...					...		
32	815	Liberlo	1.4	33	8	Feuer	NULL

6.5 Schemas und Tabellen

SQL: USE

`USE` legt ein Default-Schema fest. So kann bei Tabellen in SQL-Ausdrücken die explizite Angabe des Schemas entfallen.

Möchte man also z.B. die Datenbank `db_pokemon` auswählen, so nutzt man:

```
1 USE db_pokemon;
```

Folgend kann man den folgenden Befehl kürzen:

```
1 SELECT * FROM pokemon;
```

```
1 SELECT * FROM pokemon.pokemon;
```

SQL: SHOW

`SHOW` wird genutzt um DBMS-spezifische Strukturen abzufragen.

Dieser Befehl gibt eine Tabelle zurück. Dementsprechend lassen sich Projektionen sowie Selektionen auf das Ergebnis anwenden.

Um alle Schemas bzw Datenbanken anzuzeigen nutzt man:

```
SHOW SCHEMAS;
```

```
SHOW DATABASES;
```

	Database
1	pokemon

Alternativ kann man sich innerhalb einer Datenbank alle Tabellen abfragen:

```
SHOW TABLES;
```

```
SHOW TABLES FROM pokemon;
```

	Tables_in_pokemon
1	arenaleiter
2	attacke
3	attacke_tm
4	effektivitaet
5	entwicklung
6	generation
7	item
8	lernt
9	pokemon
10	typ
11	version

Um sich die Struktur einer Tabelle anzuzeigen nutzt man:

```
DESCRIBE pokemon;
```

```
SHOW COLUMNS FROM pokemon;
```

	Field	Type	Null	Key	Default	Extra
1	ID	int	NO	PRI	NULL	
2	Name	varchar(255)	NO		NULL	
3	Groesse	float	NO		NULL	
4	Gewicht	float	NO		NULL	
5	Generation	int	NO	MUL	NULL	
6	PrimaerTyp	varchar(255)	YES	MUL	NULL	
7	SekundaerTyp	varchar(255)	YES	MUL	NULL	

SQL: DROP

`DROP` bzw. `DELETE` löscht DBMS-spezifische Strukturen.

Wenn man die gesamte Datenbank pokemon löschen möchte, so nutzt man:

```
DROP DATABASE <Tabellenname>;
```

Möchte man die Tabelle `pokemon` löschen:

```
DROP TABLE <Tabellenname>;
```

SQL: CREATE

`CREATE` erstellt DBMS-spezifische Strukturen.

Die Datenbank `pokemon` wurde wir folgt erstellt:

```
CREATE DATABASE pokemon;
```

Um Tabellen zu erstellen muss man die jeweilige Spalten beschreiben:

```
CREATE TABLE <Tabellenname> (
    <Spaltenname> [<Parameter>, ...],
    ...,
    PRIMARY KEY (<Spaltenname>, ..., <Spaltenname>),
    FOREIGN KEY (<Spaltenname>) REFERENCES <Tabellenname> (Spaltenname),
    ...
);
```

Einige relevanten Parameter sind:

- *Typ* beschreibt den Typen der Spaltenwerte. Die meisten Typen benötigen eine Länge; einige haben jedoch eine Standardlänge:
  - `INT(n)` eine ganze Zahl der Länge n
  - `FLOAT(n)` eine Zahl der Länge n
  - `DECIMAL(n, m)` eine Zahl der Länge n mit m Nachkommastellen
  - `CHAR(n)` ein String fester Länge n
  - `VARCHAR(n)` ein String variabler Länge 0 bis n
  - `BOOLEAN` bzw. `TINYINT(1)` speichert `TRUE` bzw. 1 oder `FALSE` bzw. 0
- `DEFAULT` setzt einen Standardwert fest
- `UNIQUE` legt fest, ob der Wert einzigartig soll
- `NOT NULL` legt fest, ob der Wert `NULL` sein darf
- `CHECK(<Bedingung>)` Einzufügende Werte müssen die Bedingung erfüllen

Ein `PRIMARY KEY` ist ein Primärschlüssel, der die Entität eindeutig beschreibt. Dieser darf nie `NULL` sein.

Ein `FOREIGN KEY` ist ein Fremdschlüssel, der Primärschlüssel einer anderen Entität referenziert.

SQL-Bonus: Konfigurationen in MySQL

MySQL nutzt `SESSION` und `GLOBAL` Konfigurationen.

Die `SESSION` Konfiguration wird beim Abschalten der Datenbank auf `GLOBAL` zurückgesetzt.

```
1 SELECT @@SESSION.TIME_ZONE, @@GLOBAL.TIME_ZONE;
```

	@@SESSION.TIME_ZONE	@@GLOBAL.TIME_ZONE
1	SYSTEM	SYSTEM

Die Zeitzone kann mit folgendem Befehl angepasst werden:

```
1 SET SESSION TIME_ZONE = <Zeitzone>
```

Dabei kann als Zeitzone folgendes eingestellt werden:

- Eine exakte Zeitzone: `'Europe/Berlin'`
- Ein Offset: `'+02:00'`

SQL: Zeit- und Datumstypen

Zusätzlich gibt es komplexere Typen, welche zur Speicherung von Daten genutzt werden.

Einige Datumstypen sind:

- `DATE`
- `TIME`
- `DATETIME`

Ein Cast in verschiedene Typen ist wie folgt möglich:[a]

```sql
SELECT
    NOW() AS 'Session',
    CAST(NOW() AS DATE) AS 'Date',
    CAST(NOW() AS TIME) AS 'Time',
    CAST(NOW() AS DATETIME) AS 'Datetime';
```

	Session	Date	Time	Datetime
1	1996-02-27 10:00:00	1996-02-27	10:00:00	1996-02-27 10:00:00

Zusätzlich ist es ebenfalls möglich von Strings zu einem Datum zu casten:

```sql
SELECT
    STR_TO_DATE('27.02.1996', GET_FORMAT(DATE, 'EUR')) AS 'STR_TO_DATE',
    DATE_FORMAT(NOW(), GET_FORMAT(DATE, 'EUR')) AS 'DATE',
    -- %W = Wochentag, %D Tag im Monat, %M = Monat, %Y = Jahr
    DATE_FORMAT(NOW(), '%W, %D %M %Y') AS 'CUSTOMIZED DATE';
```

	STR_TO_DATE	DATE	CUSTOMIZED_DATE
1	1996-02-27	27.02.1996	Tuesday, 27th February 1996

Ein Datum kann über `DATE_ADD()`, oder per `+` bzw. `-`, modifiziert werden:

```sql
SELECT
    NOW() + INTERVAL 1 YEAR,
    DATE_ADD(NOW(), INTERVAL 1 YEAR);
```

Alternativ kann ebenfalls eine Zeitspanne ermittelt werden:

```sql
SELECT
    DATEDIFF(DATE_ADD(NOW(), INTERVAL 1 YEAR), NOW()),
    TIMEDIFF(DATE_ADD(NOW(), INTERVAL 1 HOUR), NOW());
```

[a] `NOW()` gibt das aktuelle Datum in der Formatierung `session.TIME_ZONE` zurück

SQL: ALTER TABLE

`ALTER TABLE` modifiziert eine Tabelle.

`ADD` wird genutzt, um Spalten hinzuzufügen. Die Syntax ist dabei identisch zu der Syntax beim Erstellen der Tabelle.

```
ALTER TABLE <Tabellenname>
ADD (
    <Spaltenname> [<Parameter>, ...],
    ...
);
```

`MODIFY` verändert existierende Spalten.

```
ALTER TABLE <Tabellenname>
MODIFY <Spaltenname> [<Neuer Spaltenname>] [<Neuer Parameter>, ...];
```

`DROP` löscht angegebene Spalten.

```
ALTER TABLE <Tabellenname>
DROP <Spaltenname>;
```

`RENAME TO` benennt Tabellen um.

```
ALTER TABLE <Tabellenname>
RENAME TO <neuer Tabellenname>;
```

6.5.1 Beispiele

Beispiel: Schemas und Tabellen

Die SQL Befehle, die zur Erstellung unserer Beispieldatenbank genutzt wurden sind in `pokemon.sql` zu finden.[a]

Einige relevante Befehle für die DBMS-Struktur sind hier zusammengefasst:

Falls `pokemon` existiert wird diese Tabelle gelöscht:

```
DROP TABLE IF EXISTS pokemon;
```

Im Nachhinein wird eine neue Tabelle erstellt:

```
CREATE TABLE pokemon (
    ID int(11) NOT NULL,
    Name varchar(255) NOT NULL,
    Groesse float NOT NULL,
    Gewicht float NOT NULL,
    Generation int(11) NOT NULL,
    PrimaerTyp varchar(255) DEFAULT NULL,
    SekundaerTyp varchar(255) DEFAULT NULL,
    PRIMARY KEY (ID),
    FOREIGN KEY (PrimaerTyp) REFERENCES typ (Bezeichnung),
    FOREIGN KEY (SekundaerTyp) REFERENCES typ (Bezeichnung),
    FOREIGN KEY (Generation) REFERENCES generation (ID)
)
```

[a] Siehe GitHub.

6.6 Sichten

Definition: Sicht

Eine *Sicht* ist eine logische Relation in einem Datenbanksystem.

Diese logische Relation wird über eine im DBMS gespeicherte Abfrage definiert. Der Benutzende kann eine Sicht wie eine normale Tabelle abfragen.

Wann immer eine Abfrage diese Sicht benutzt, wird diese zuvor durch das DBMS berechnet.

Eine Sicht stellt im Wesentlichen einen Alias für eine Abfrage dar.

Man kann unterscheiden zwischen:

- *temporären Sichten*:
  - im Kontext von `WITH`-Definitionen
  - überdauern die Anfrage nicht
- *statischen Sichten*:
  - Datenschutz
  - Vereinfachung von Aufgaben
  - Darstellung der Generalisierung

Sichten arbeiten auf den Daten der logischen Gesamtsicht. Somit sind entsprechende Transformationsregeln notwendig.

Änderungen einer Sicht müssen immer noch auf die Basisrelationen abgebildet werden. Dies ist je nach Sichtendefinition nicht immer möglich.

Auch können Änderungen von Tupeln ein „Löschen“ des Tupels aus der entsprechenden Sicht bewirken.

Damit eine Sicht änderbar ist, muss sie durch eine einzelne `SELECT`-Anweisung definiert sein.[a]

Weiterhin gilt für Änderbarkeit:[b]

- Die `SELECT`-Klausel darf nur Attributnamen enthalten und jeden Namen nur einmal, keine Aggregatfunktionen, berechnete oder konstante Ausdrücke, ebenso kein `DISTINCT`.
- Die `FROM`-Klausel darf nur einen einzigen Relationsnamen enthalten. Dieser muss eine Basisrelation oder eine änderbare Sicht bezeichnen.
- Falls die `WHERE`-Klausel geschachtelte Anfragen beinhaltet, darf in deren `FROM`- Klauseln der Relationsname auch `FROM` nicht auftauchen, d.h. keine korrelierten Unterabfragen.

[a] d.h. kein `JOIN`, `UNION`, etc.
[b] Mehr dazu: edb-Online-Lexikon, TH Köln

SQL: VIEW

Eine `VIEW` erzeugt aus einem `SELECT` eine temporäre Tabelle.

Dementsprechend kann man wie folgt eine temporäre Tabelle aller Feuer-Pokémon erstellen[a], um bei den folgenden Abfragen Laufzeit zu sparen.[b]

```sql
-- Erstelle View feuerpokemon
CREATE OR REPLACE VIEW feuerpokemon
AS (
    SELECT *
    FROM pokemon
    WHERE
        PrimaerTyp = 'Feuer'
        OR SekundaerTyp = 'Feuer'
);
-- Gebe Entitaeten der View feuerpokemon aus
SELECT * FROM feuerpokemon;
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	4	Glumanda	0.6	8.5	1	Feuer	NULL
2	5	Glutexo	1.1	19	1	Feuer	NULL
3	6	Glurak	1.7	90.5	1	Feuer	Flug
...					...		
71	851	Infernopod	3	120	8	Feuer	Käfer

Um die View zu entfernen, nutzt man `DROP`:

```sql
DROP VIEW feuerpokemon;
```

[a] Das Schlüsselwort `OR REPLACE` überschreibt eine eventuell existierende Tabelle

[b] Wenn der innere `SELECT` z.B. mit einem oder mehreren `JOIN`-Befehlen kombiniert wird oder die Bedingung in der Selektion eine gewisse Komplexität überschreitet und die selbe Abfrage mehrfach benötigt wird

6.7 Transaktionen

Definition: ACID

Der Begriff *ACID* beschreibt Regeln und Eigenschaften zur Durchführung von Transaktionen in Datenbankmanagementsystemen.

Dabei steht ACID für:

- *Atomicity* (Atomarität)
  - atomar in dem Sinne, dass entweder alle Operationen der Transaktion bedingten Änderungen in der Datenbank umgesetzt werden, oder gar keine
- *Consistency* (Konsistenz)
  - Ausgangspunkt ist konsistente Datenbank vor Beginn einer Transaktion
  - am Ende einer Transaktion ist Datenbank in konsistentem Zustand, d.h.
    - entweder jede Integritätsregel ist erfüllt[a]
    - Transaktion wird komplett zurückgesetzt und Datenbank befindet sich wieder in dem (konsistenten) Zustand wie zu Beginn
- *Isolation* (Nebenläufigkeit)
  - parallele bzw. gleichzeitig ausgeführte Transaktionen beeinflussen sich nicht[b]
  - Locking (Lese- bzw. Schreibsperre)
- *Durability)* (Dauerhaftigkeit)
  - nach Abschluss einer Transaktion haben ausgeführte Änderungen dauerhaft Bestand

ACID wird häufig zur Charakterisierung eines DBMS genannt, insbesondere auch für Nicht-Relationale DBMS.

[a] z.B. referentielle Integrität (Fremdschlüssel)
[b] Fehlerfälle sind z.B. „Lost Updates“ oder „Dirty Reads“

6.7.1 Locking

Definition: Lost Update

Zwei Transaktionen modifizieren parallel denselben Datensatz und nach Ablauf dieser beiden Transaktionen wird nur die Änderung von einer von ihnen übernommen.

Beispiel: Lost Update

Wir haben zwei nebenläufige Transaktionen, die denselben Datensatz modifizieren:

- T_1: Buche Betrag 300 von Konto A auf Konto B
- T_2: Schreibe Konto A 3% Zinsen zu

	T_1	T_2
01	`a1 = read(A)`	
02	`a1 = a1 - 300`	
03		`a2 = read(A)`
04		`a2 = a2 * 1.03`
05		`write(A, a2)`
06	`write(A, a1)`	
07	`b1 = read(B)`	
08	`b1 = b1 + 300`	
09	`write(B, b1)`	

Die Änderung von T_2 in Zeile 5 geht verloren, da T_1 im nächsten Schritt die Änderung überschreibt.

Definition: Dirty Read

Daten einer noch nicht abgeschlossenen Transaktion werden von einer anderen Transaktion gelesen.

Beispiel: Dirty Read

Wir haben zwei nebenläufige Transaktionen, die denselben Datensatz modifizieren:

- T_1: Buche Betrag 300 von Konto A auf Konto B
- T_2: Schreibe Konto A 3% Zinsen zu

	T_1	T_2
01	`a1 = read(A)`	
02	`a1 = a1 - 300`	
03	`write(A, a1)`	
04		`a2 = read(A)`
05		`a2 = a2 * 1.03`
06		`write(A, a2)`
07	`b1 = read(B)`	
08	`b1 = b1 + 300`	
09	`write(B, b1)`	
10	`...`	
n	`abort`	

Sollte T_1 abbrechen (`abort`), oder eine andere Änderung an A vornehmen, war die Sicht auf A in Schritt 3 für T_2 falsch.

Definition: Locking

Unter *Locking* versteht man das Sperren des Zugriffs auf ein Betriebsmittel.

Eine solche Sperre ermöglicht den exklusiven Zugriff eines Prozesses auf eine Ressource, d. h. mit der Garantie, dass kein anderer Prozess diese Ressource liest oder verändert, solange die Sperre besteht.

Regeln beim Locking:

- Jedes Objekt muss vor der Benutzung gesperrt werden.
- Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
- Eine Transaktion respektiert vorhandene Sperren und wird ggf. in eine Warteschlange eingereiht.
- Jede Transaktion durchläuft zwei Phasen:
  - Wachstumsphase (nur Sperren anfordern)
  - Freigabephase (nur Sperren freigeben)
- Spätestens bei Transaktionsende muss eine Transaktion all ihre Sperren zurückgeben.

Bonus: Lesesperre

Besitzt eine Ressource eine *Lesesperre* („Shared Lock“), so möchte der Prozess, der diese Sperre gesetzt hat, von der Ressource nur lesen.

Somit können auch andere Prozesse auf diese Ressource lesend zugreifen, dürfen diese aber nicht verändern.

Bonus: Schreibsperre

Besitzt eine Ressource eine *Schreibsperre* („Exclusive Lock“), wird verhindert, dass die Ressource von anderen Prozessen gelesen oder geschrieben wird, da der Prozess, der den Lock gesetzt hat, die Ressource verändern möchte.

Es wird vorausgesetzt, dass keine Lesesperre auf die Ressource gesetzt ist.

SQL: TRANSACTION

Um eine Transaktion zu starten, nutzt man:

```
1  START TRANSACTION;
```

Nun kann man Daten einfügen, löschen oder verändern bzw. die DBMS-spezifische Struktur anpassen.

Sollten dabei etwaige Fehler auftreten, welche irreparable Schäden an der Datenbank verursachen, ist erst mal nur die Session der Verursachenden betroffen.

Um die Transaktion zu abzuschließen, nutzt man:

```
1  COMMIT;
```

Alternativ bricht man die Transaktion folgendermaßen ab:

```
1  ROLLBACK;
```

SQL: AUTOCOMMIT

Um jeden Befehl sofort zu verarbeiten muss die Einstellung `AUTOCOMMIT` auf `1` gesetzt werden.

Standardmäßig ist der Wert bereits auf `1`, jedoch nutzen wir um sicher zu gehen:

```
1 SET AUTOCOMMIT = 1;
```

Wenn der Wert auf `0` gesetzt wurde, werden Änderungen nur in der eigenen `SESSION` vorgenommen. Wenn man seine Änderungen veröffentlichen möchte, nutzt man `COMMIT;`.

SQL: FOREIGN_KEY_CHECKS

Der folgende Befehl schaltet das automatische Überprüfen von Fremdschlüssel-Beziehungen aus:

```
1 SET FOREIGN_KEY_CHECKS = 0
```

Analog schaltet man die Überprüfung folgendermaßen wieder ein:

```
1 SET FOREIGN_KEY_CHECKS = 1
```

Dieser Befehl birgt auf den ersten Blick große Gefahren, ist jedoch vor allem bei Selbstrelationen nötig, wenn sich noch nicht existierende Entitäten gegenseitig referenzieren.

6.8 Daten anlegen, verändern und löschen

SQL: INSERT INTO

`INSERT INTO` trägt Daten in eine Datenbank ein.

```
INSERT INTO <Tabellenname>
    (<Spaltenname>, ...)
VALUES
    (Wert, ...),
    ...;
```

Wenn man alle Attribute in der vorgegebenen Reihenfolge angibt, kann man die Angabe der Spalten weglassen.

Es ist ebenfalls möglich Unterabfragen zum Herausfinden von Daten mit `INSERT INTO` zu verwenden.

Beispiel: INSERT INTO

Hier wird die Entität `Glumanda` erstellt:

```
INSERT INTO pokemon
    (ID, Name, Groesse, Gewicht, Generation, PrimaerTyp, SekundaerTyp)
VALUES
    (4, 'Glumanda', 0.6, 8.5, 1, 'Feuer', NULL);
```

```
INSERT INTO pokemon
VALUES
    (4, 'Glumanda', 0.6, 8.5, 1, 'Feuer', NULL);
```

Beispiel: INSERT INTO

Da `NULL` der Standardwert für `SekundaerTyp` ist, und Glumanda keinen Sekundärtyp hat, kann man `Glumanda` ebenfalls wie folgt erstellen:

```
INSERT INTO pokemon
    -- kein SekundaerTyp
    (ID, Name, Groesse, Gewicht, Generation, PrimaerTyp)
VALUES
    (4, 'Glumanda', 0.6, 8.5, 1, 'Feuer');
```

Beispiel: INSERT INTO

Mehrere Pokemon ließen sich wie folgt gleichzeitig erstellen:

```
INSERT INTO pokemon
VALUES
    (5, 'Glutexo', 1.1, 19, 1 'Feuer', NULL),
    (6, 'Glurak', 1.7, 90.5, 1 'Feuer', 'Flug');
```

SQL: UPDATE

Um Werte anzupassen nutzt man `UPDATE`:

```
UPDATE <Tabellenname>
SET
    <Spaltenname> = <Wert>,
    ...
WHERE <Bedingung>
```

Wenn man also den Typ von Glumanda auf Wasser setzen wollen würde, nutzt man:

```
UPDATE pokemon
SET PrimaerTyp = 'Wasser'
WHERE Name = 'Glumanda'
```

`UPDATE` verändert alle Entitäten, auf die die Bedingung zutrifft.

Dementsprechend sollte man, wenn man eine einzige bestimmte Entität anpassen möchte, den Primärschlüssel bzw. `UNIQUE` Felder abfragen.

SQL: DROP

`DROP` bzw. `DELETE` wird genutzt um spezifische Entitäten zu löschen.

```
DROP FROM <Tabellenname>
[WHERE <Bedingung>];
```

Möchte man nun Schiggy löschen, nutzt man:

```
DROP FROM pokemon
WHERE Name = 'Schiggy';
```

SQL: TRUNCATE

`TRUNCATE` löscht Entitäten einer Tabelle, lässt die Tabelle an sich jedoch bestehen.

```
TRUNCATE TABLE <Tabellenname>
```

SQL-Bonus: LOCK TABLES

Wenn man mit mehreren Benutzenden an einer Tabelle arbeitet, ist es sinnvoll eine Tabelle zu sperren. So kann bis zum freigeben der Tabelle niemand die Tabelle bearbeiten.

```
LOCK TABLES <Tabellenname> WRITE;
```

```
UNLOCK TABLES;
```

6.8.1 Beispiele

Beispiel: INSERT INTO

Die SQL Befehle, welche zur Erstellung unserer Beispieldatenbank genutzt wurden sind in `pokemon.sql` zu finden.

Hier ein Auszug, durch den die ersten Pokemon erstellt wurden:

```sql
INSERT INTO `pokemon`
VALUES
    (1,'Bisasam',0.7,6.9,1,'Pflanze','Gift'),
    (2,'Bisaknosp',1,13,1,'Pflanze','Gift'),
    (3,'Bisaflor',2,100,1,'Pflanze','Gift'),
    (4,'Glumanda',0.6,8.5,1,'Feuer',NULL),
    (5,'Glutexo',1.1,19,1,'Feuer',NULL),
    (6,'Glurak',1.7,90.5,1,'Feuer','Flug'),
    (7,'Schiggy',0.5,9,1,'Wasser',NULL),
    (8,'Schillok',1,22.5,1,'Wasser',NULL),
    (9,'Turtok',1.6,85.5,1,'Wasser',NULL);
```

Beispiel: UPDATE

Erhöhen Sie das Gewicht aller Pokémon um 5kg und verringern Sie gleichzeitig die Größe um 25cm.

Sollte dabei die Größe unter 1cm fallen, bleibt diese bei diesem Wert.

```sql
UPDATE pokemon
SET
    Gewicht = Gewicht + 5,
    Groesse = MIN(Groesse - 0.25, 0.01)
```

Beispiel: UPDATE

Um die Spiele fair zu halten sollen alle Attacken auf die Durchschnittsstärke aller Attacken mit dem selben Typen gesetzt werden.

```sql
UPDATE attacke AS attacke1
SET Staerke = (
    SELECT ROUND(AVG(Staerke), 0)
    FROM attacke AS attacke2
    WHERE attacke1.Typ = attacke2.Typ
);
```

6.9 Trigger, Stored Procedures

SQL: DELIMITER

Da wir in PROCEDUREs einen DELIMITER nutzen, welcher von dem DBMS fälschlicherweise als Ende für den Befehl zum Erstellen der PROCEDURE erkannt wird, können wir diesen ändern.

```
DELIMITER <Trennzeichen>
```

SQL: PROCEDURE

Eine PROCEDURE wird genutzt um SQL-Befehle methodenähnlich zum Auszuführen zu speichern.

Diese PROCEDUREs sind vergleichbar mit VIEWs. Jedoch wird bei einer VIEW ein Snapshot der aktuellen Tabelle erstellt und bei PROCEDURE jedes mal erneut die Datenbank aufgerufen.

Sollte man nun eine PROCEDURE zum Anzeigen aller Feuer-Pokémon erstellen wollen, nutzt man:

```
-- Aendern des Trennzeichens
DELIMITER //
-- Procedure anlegen
CREATE PROCEDURE ShowFeuerpokemon()
BEGIN
    SELECT *
    FROM pokemon
    WHERE
        PrimaerTyp = 'Feuer'
        OR SekundaerTyp = 'Feuer';
END //
-- Aendern des Trennzeichens rueckgaengig machen
DELIMITER ;
```

Um diese PROCEDURE nun aufzurufen nutzen wir:

```
CALL ShowFeuerpokemon();
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	4	Glumanda	0.6	8.5	1	Feuer	NULL
2	5	Glutexo	1.1	19	1	Feuer	NULL
3	6	Glurak	1.7	90.5	1	Feuer	Flug
...					...		
71	851	Infernopod	3	120	8	Feuer	Kaefer

Zum Löschen einer PROCEDURE nutzen wir:

```
DROP PROCEDURE ShowFeuerpokemon;
```

SQL: PROCEDURE (mit Parameter)

Alternativ kann man Parameter mitgeben:

```
DELIMITER //
CREATE PROCEDURE ShowPokemonFromType(typ varchar(255))
BEGIN
    SELECT *
    FROM pokemon
    WHERE
        PrimaerTyp = typ
        OR SekundaerTyp = typ;
END //
DELIMITER ;
```

Wenn man erneut alle Feuer-Pokémon ausgeben möchte, erhält man äquivalentes Ergebnis wie folgt:

```
CALL ShowPokemonFromType('Feuer');
```

Jedoch können wir uns ebenfalls jeden anderen Typen ausgeben lassen:

```
CALL ShowPokemonFromType('Pflanze');
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	1	Bisasam	0.7	6.9	1	Pflanze	Gift
2	2	Bisaknosp	1	13	1	Pflanze	Gift
3	3	Bisaflor	2	100	1	Pflanze	Gift
...					...		
107	898	Coronospa	1.1	7.7	8	Psycho	Pflanze

```
CALL ShowPokemonFromType('Wasser');
```

	ID	Name	Groesse	Gewicht	Generation	PrimaerTyp	SekundaerTyp
1	7	Schiggy	0.5	9	1	Wasser	NULL
2	8	Schillok	1	22.5	1	Wasser	NULL
3	9	Turtok	1.6	85.5	1	Wasser	NULL
...					...		
141	883	Pescryodon	2	175	8	Wasser	Eis

SQL: TRIGGER

Um auf Eingaben zu reagieren kann man `TRIGGER` verwenden.

Diese lösen automatisch aus, sollte ein bestimmtes Event stattfinden.

```
DELIMITER $$
CREATE TRIGGER <Triggername>
<Triggerzeitpunkt> <Triggerevent>
ON <Tabellenname>
FOR EACH ROW BEGIN
    <TODO>
END $$
DELIMITER ;
```

Zulässige Werte sind dabei:

- Triggerzeitpunkt
  - `BEFORE`
  - `AFTER`
- Triggerevent
  - `INSERT`
  - `UPDATE`
  - `DELETE`
- In der Aufgabe verwendbare Keywords
  - `OLD`
    - die zu löschende Entität oder
    - die zu verändernde Entität vor der Änderung
  - `NEW`
    - die einzufügende Entität oder
    - die zu verändernde Entität nach der Änderung

Sollte man beispielsweise beim Einfügen von neuen Typen die Effektivitätsliste direkt anpassen wollen, indem man den neuen Typen mit allen existierenden kreuzt, so nutzt man:

```
DELIMITER $$
CREATE TRIGGER after_insert_typ
BEFORE INSERT
ON typ
FOR EACH ROW BEGIN
    INSERT INTO effektivitaet (Angreifend, Verteidigend)
        SELECT NEW.Bezeichnung, typ.Bezeichnung FROM typ;
    INSERT INTO effektivitaet (Angreifend, Verteidigend)
        SELECT typ.Bezeichnung, NEW.Bezeichnung FROM typ;
END $$
DELIMITER ;
```

6.10 ORM und Hibernate

Definition: Object-Relational Mapping

Object-Relational Mapping bzw. ORM ist eine Technik der Softwareentwicklung, mit der ein in einer objektorientierten Programmiersprache geschriebenes Anwendungsprogramm seine Objekte in einer relationalen Datenbank ablegen kann. Dem Programm erscheint die Datenbank dann als objektorientierte Datenbank, was die Programmierung erleichtert.

Implementiert wird diese Technik normalerweise mit Klassenbibliotheken, wie beispielsweise Entity Framework für .NET-Programmiersprachen, Hibernate für Java.

Man unterscheidet zwischen den Varianten:

- Statisches Mapping
  - Klassen
  - Vererbung
  - Aggregation, Komposition, Assoziation
- Dynamisches Mapping[a]
  - Constraints und Trigger
  - Performance: Dirty Checking, Lazy Fetching, Caching

[a] Nicht behandelt in der Vorlesung.

Definition: Abbildung von Vererbungshierarchien

Es gibt im Wesentlichen drei verschiedene Verfahren, um Vererbungshierarchien auf Datenbanktabellen abzubilden:

- *Tabelle pro Vererbungshierarchie* bzw. *Single Table*:
  - alle Attribute der Basisklasse und aller abgeleiteten Klassen in gemeinsamer Tabelle
  - Diskriminator in weiterer Spalte abgelegt, der festlegt, welcher Klasse das in dieser Zeile gespeicherte Objekt angehört
  - Attribute von abgeleiteten Klassen dürfen den meisten Fällen nicht mit einem `NOT-NULL`-Constraint versehen werden
  - Beschränkungen der Anzahl erlaubter Spalten pro Tabelle können Ansatz bei großen Klassen bzw. Klassenhierarchien vereiteln
- *Tabelle pro Unterklasse* bzw. *Joined* :
  - eine Tabelle für die Basisklasse angelegt und für jede davon abgeleitete Unterklasse eine weitere Tabelle
  - Diskriminator wird nicht benötigt, weil Klasse eines Objekts durch 1:1-Beziehung zwischen Eintrag in Tabelle der Basisklasse und Eintrag in einer der Tabellen der abgeleiteten Klassen festgelegt ist
- *Tabelle pro konkrete Klasse* bzw. *Table per Class*:
  - Attribute der abstrakten Basisklasse in die Tabellen für die konkreten Unterklassen mit aufgenommen
  - Tabelle für Basisklasse entfällt
  - Nachteil besteht darin, dass es nicht möglich ist, mit einer Abfrage Instanzen verschiedener Klassen zu ermitteln

Index

Index SQL

Beispiele