WAS SIND DATENBANKEN?

- Definition: Eine Datenbank ist eine systematische Sammlung von Daten.
- Datenbanken unterstützen elektronische Speicherung und Manipulation von Daten.
- Eine Datenbank besteht aus Tabellen (= **Entitäten**), die in Reihen und Spalten dargestellt werden
- Sie ermöglichen den schnellen Zugriff auf Daten und die Verwaltung großer Datenmengenn durch bestimmte Befehle (Zumeist in der Sprache SQL = Structured Query language).
- Eine Datenbank wird normalerweise durch ein Datenbankmanagmentsystem (DBMS) verwaltet
- Bekannte DBMS sind zum Beispiel MySQL, Microsoft Access, Microsoft SQL Server, …

WARUM NICHT EXCELSHEETS?

Excel gut für eine Person

Zugriff auf Daten durch mehrere Personen schwierig (vor allem gleichzeitig)

Manipulation von Daten eventuell aufwendig

BEDEUTUNG VON DATENBANKEN

- Datenbanken sind entscheidend für das Management von Informationen in Unternehmen, Regierungen und anderen Organisationen.
- Sie ermöglichen komplexe Analysen und Datenverarbeitung.

BEISPIEL EINER TABELLE

UserID	Name	Email	DateOfBirth
1	Anna	anna@example.com	1985-06-15
2	Bob	bob@example.com	1990-09-20
3	Carlos	carlos@example.com	1982-12-05
4	Diana	diana@example.com	1995-04-10
5	Emily	emily@example.com	1998-07-22

BEISPIELTABELLE USER

- Spalten: UserID, Name, Email, DateOfBirth.
  - 'UserID' ist der eindeutige Identifikator für jeden Benutzer.
- Datentyp: Ganzzahl (Integer).
  - Bedeutung: Wird verwendet, um Benutzer eindeutig zu identifizieren und zu referenzieren = **Primary Key**
- 'Name' beinhaltet den Vornamen des Benutzers.
  - Datentyp: Zeichenkette (String).
- 'Email' speichert die E-Mail-Adresse des Benutzers.
  - Datentyp: Zeichenkette (String).
- 'DateOfBirth' gibt das Geburtsdatum des Benutzers an.
  - Datentyp: Datum (Date).

GEMEINSAMES BEISPIEL – STUDENT UND KURSE

- Folgende Probleme:
  - Die Kurse kommen eventuell öfter vor (Zeichenketten die wiederholt werden, nehmen mehr Speicher)
  - Wie kann ein Student mehrere Kurse belegen?
  - => Studenten stehen mit ihren Kursen in **Relation** => Relationale Datenbanken
- „Relation“: Menge von Entitäten mit gleichen Eigenschaften
- Relationale Datenbank ist Sammlung von Tabellen, die miteinander in Beziehung stehen

GRUNDLAGEN RELATIONALER DATENBANKEN

- Relationale Datenbank ist System zur Speicherung von Daten in Tabellenform.
- Entwickelt von E.F. Codd bei IBM in den 1970ern.
- Vorteile:
  - Strukturierte Datenspeicherung ermöglicht präzise Abfragen.
  - Skalierbar und sicher, ideal für komplexe Anwendungen.

GRUNDKONZEPTE DES RELATIONENMODELLS

- Daten werden in Tabellen (Relationen) gespeichert.
- Jede Tabelle besteht aus Zeilen (Datensätzen) und Spalten (Attributen).
- **Primärschlüssel** (**primary key**) identifiziert jeden Datensatz eindeutig.
- **Fremdschlüssel** (**foreign key**) stellt Beziehungen zu anderen Tabellen her.

BEISPIEL STUDENTEN-KURSE

Problem der Datenredundanz

Gleiche Daten belegen unnötigen Speicher

Lösung: Reduzierung von Redundanz und Verbesserung der Datenintegrität (Einzigartigkeit eines Datensatzes)
-> **Normalformen**

Normalformen sind Regeln im Datenbankdesign

Je höher die Normalform, die eingehalten wird, desto freier wird die Datenbank von Datenredundanzen sein

NULLTE NORMALFORM

ALLE INFORMATIONEN SIND IN EINER TABELLE ZUSAMMENGEFASST. Z.B.: BESTELLUNG:

BestellID	Datum	Name	Alter	Adresse	ProduktID	Produktbeschreibung	Anzahl	Produktpreis
1	01.01.2022	Max Mustermann	25	Musterstrasse 1/1, 1010 Wien	5	Handy	1	500 Euro

ERSTE NORMALFORM

ALLE INFORMATIONEN EINER TABELLE LIEGEN ATOMAR VOR. DAS HEIßT, DASS MAN DIE WERTE NICHT MEHR IN EINZELNE TEILE ZERLEGEN KANN

BestellID	Datum	Vorname	Nachname	Alter	Strasse	Hausnummer	Türnummer	PLZ	Stadt	ProduktID	Produktbeschreibung	Anzahl	Produktpreis	Währung
1	01.01.2022	Max	Mustermann	25	Musterstrasse	1	1	1010	Wien	5	Handy	1	500	Euro

ZWEITE NORMALFORM

ERSTE NORMALFORM + JEDES ATTRIBUT, DASS KEINEN TEIL DES SCHLÜSSELS BILDET (ES KÖNNEN NICHT NUR EINES SONDERN MEHRERE ATTRIBUTE SEIN) IST VOM SCHLÜSSEL VOLL FUNKTIONAL ABHÄNGIG.

Jedes Nicht-Schlüsselattribut in einer Tabelle sollte sich ausschließlich auf den gesamten Primärschlüssel beziehen, nicht auf einen Teil davon.

Kunde

KiD	Vorname	Nachname	Alter	Strasse	Hausnummer	Türnummer	PLZ	Stadt
77	Max	Musterman n	25	Musterstra sse	1	1	1010	Wien

Produkt

ProduktID	Produktbeschreibung	Produktpreis	WaehrungsID
5	Handy	500	1

Bestellung

BestellID	ProduktID	KiD	Anzahl
111	5	77	1

DRITTE NORMALFORM

ZWEITE NORMALFORM + ES GIBT KEIN NICHTSCHLÜSSELATTRIBUT, DASS TRANSITIV VON EINEM SCHLÜSSELKANDIDATEN ABHÄNGT

= JEDES NICHT SCHLÜSSEL ATTRIBUT DARF NUR VON SCHLÜSSELN ABHÄNGEN, NICHT ABER VON ANDEREN NICHT SCHLÜSSEL ATTRIBUTEN

IM GEZEIGTEN BEISPIEL IST DIE STADT IN DER TABELLE PERSON VON DER POSTLEITZAHL ABHÄNGIG UND DIESE WIEDERUM VON DER KID. DAHER MÜSSTE MAN DIE STADT VON DER PERSON TRENNEN UND FOLGENDE TABELLE KREIEREN:

ID	PLZ	Stadt
1	1010	Wien

BOYCE CODD NORMALFORM

- Dritte Normalform + es darf kein Teil eines Schlüsselkandidaten von einem Teil eines anderen Schlüsselkandidaten funktional abhängig sein.
- Angenommen, wir haben eine Tabelle für Studenten mit den folgenden Spalten:
  Matrikelnummer (Primärschlüssel), Vorname, Nachname, Kurs_ID (ein Schlüsselkandidat), Kursname (nicht der Primärschlüssel), Professor (nicht der Primärschlüssel)
  => Kursname und der Professor nur von der Kurs_ID abhängig
- Lösung?

BOYCE CODD NORMALFORM

- Student: Matrikelnummer, Vorname, Nachname
- Kurs: KursID, Kursname, Professor
- StudentKurs: Matrikelnummer, KursID

BEISPIEL STUDENT UND KURSE

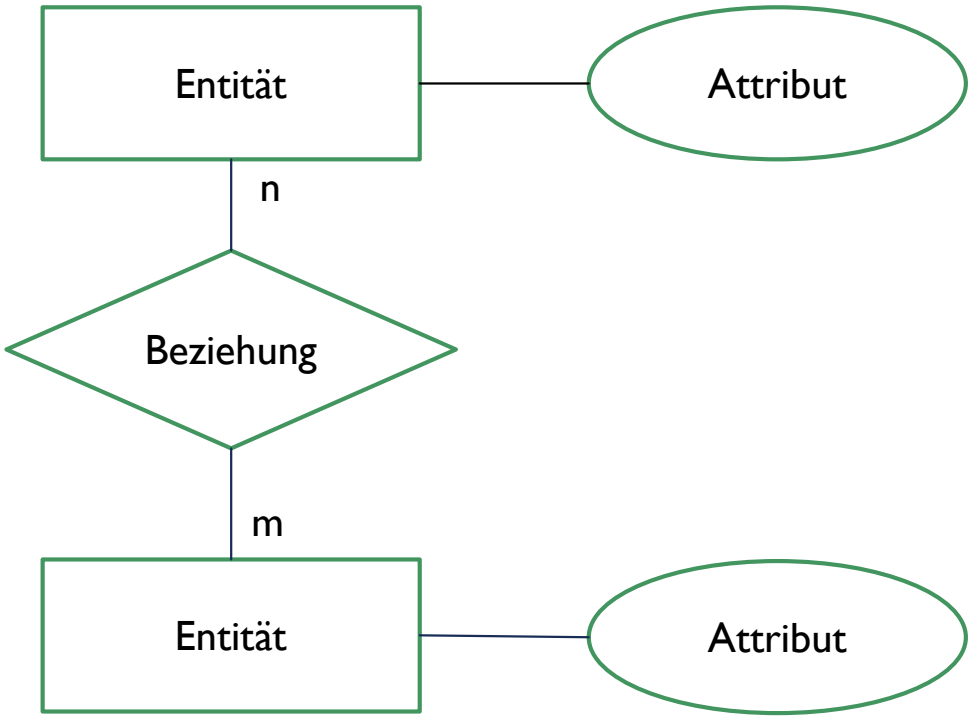
- Ein Student besucht nur einen Kurs, ein Kurs darf von mehreren Studenten besucht werden => **1:n Beziehung = 1:n Relation**
- Wenn ein Student mehrere Kurse besuchen dürfte und 1 Kurs von mehreren Studenten besucht werden könnte, wäre es eine **n:m Beziehung**. Wie würden dazu die Tabellen aussehen?
- Eine **1:1 Beziehung** würde einen Datensatz von einer Tabelle genau mit einem Datensatz einer anderen Tabelle verbinden.
- Ein **Entity-Relationship-Diagramm (ERD oder ER-Diagramm)** zeigt Entitäten und ihre Beziehungen zueinander.

TEXTUELLE DARSTELLUNG EINER ENTITÄT

- = Entitätsbeschreibung
- Beschreibung der Struktur von Tabellen oder Entitäten
- Insbesondere in der konzeptionellen oder logischen Phase des Datenbankdesigns verwendet
- Format: Tabellenname (Schlüsselfeld, Feld2, Feld3, ...)
- Beispiel:

  Users (UserID, Name, Email, DateOfBirth)

ENTITY RELATIONSHIP DIAGRAM



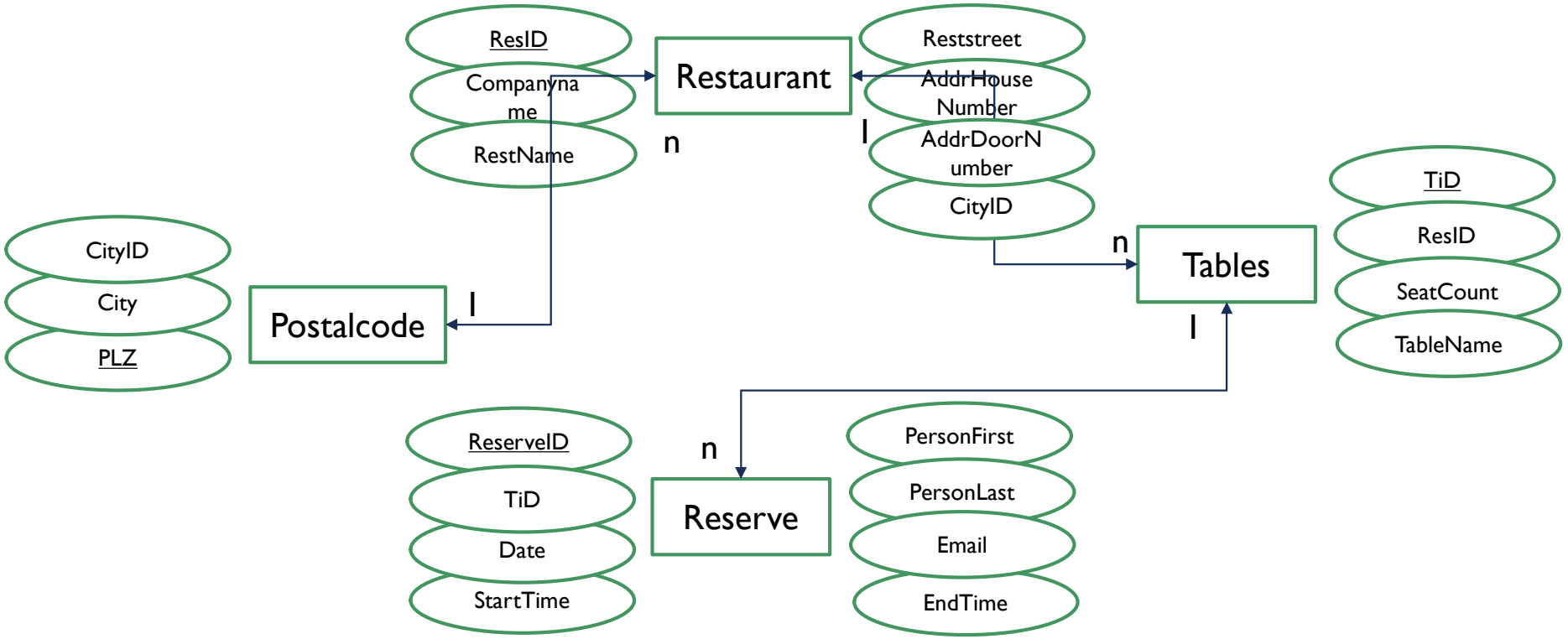
ENTITY RELATIONSHIP DIAGRAM

- Das Entity Relationship Diagram beschreibt die Beziehung zwischen den einzelnen Entitäten (Tabellen)
- Eine Entität beschreibt dabei eine Tabelle, diese beinhaltet mehrere Attribute unterschiedlicher Datentypen, darunter einen Primärschlüssel (kann auch aus mehreren Attributen bestehen).
- Die Beziehung beschreibt, wie sich 2 Entitäten zueinander verhalten.
- Die Beziehung einer Tabelle zu einer anderen drückt sich in 3 Relationsformen aus: 1:n, 1:1, n:m

WEITERE BEISPIELE FÜR BEZIEHUNGEN

- 1:n bedeutet ein Eintrag der linken Tabelle kann mehreren Einträgen der rechten Tabelle (n) zugeordnet sein, umgekehrt sind aber ein oder mehrere Einträge der rechten Tabelle max. einem Eintrag der linken Tabelle Zugeordnet. Z.B.:
- Trainer – Fussballer (1:n – 1 Trainer trainiert mehrere Fußballer, jeder dieser Fußballer hat einen Trainer)
- 1:1 bedeutet ein Eintrag der linken Tabelle ist einem Eintrag der rechten Tabelle (1) zugeordnet und umgekehrt. Z.B.:
  CEO – Firma (1:1 – eine Person ist CEO von einer Firma, eine Firma hat einen CEO)
- n:m bedeutet n Einträge der linken Tabelle können m Einträgen der rechten Tabelle zugeordnet sein, und umgekehrt. Z.B.:
  Lehrer – Schüler (Ein Lehrer kann 1-n Schüler haben, 1-m Schüler können 1-n Lehrer haben)

BEISPIEL – TISCHRESERVIERUNG – WELCHE NF?



SQL

- SQL => Durch bestimmte Befehle kann man Daten aus einer Datenbank selektieren (erhalten) oder manipulieren
- Wichtigsten Befehle für die Selektion von Daten:
  - SELECT – um Daten zu selektieren
  - FROM – wählt eine Tabelle
  - WHERE – beschreibt eine Bedingung, die gelten muss, damit Daten selektiert werden
  - JOIN – verknüpft Tabellen, die durch Ihre keys verbunden sind
- Eine selektive Query besteht immer mindestens aus einem SELECT und einem FROM statement
- Syntax: **SELECT** * oder **SELECT** „spalte1“, „spalte2“, … **FROM** tabelle (**WHERE** condition)

SQL BEISPIELE

- SELECT * FROM Personen => Selektiere die Werte aller Spalten von der Tabelle Personen
- SELECT * FROM Personen WHERE Age > 50 ?
- SELECT * FROM Personen WHERE Age > 30 AND Age < 50 ?
- Selektiere alle Personen die jünger als 20 oder älter als 60 sind?

GEMEINSAME BEISPIELE – NORTHWIND DATENBANK

- Northwind Datenbank ist eine Beispieldatenbank für ein Bestellmanagmentsystem und eignet sich zum Üben von Queries
- Installation:
  1. Öffne MySQL Workbench
  2. Gehe zu https://github.com/harryho/db-samples/blob/master/mysql/northwind.sql
  3. Kopiere den Text dort mit Strg-C
  4. Füge den Text in einem Queryfile ein (oben rechts das erste Symbol öffnet ein neues Queryfile)
  5. Starte das Skript
  6. Eventuell muss man einen Rechtsklick auf die leere Fläche machen, in der die Datenbanken aufgelistet werden und dann „Refresh“ wählen

JOINS

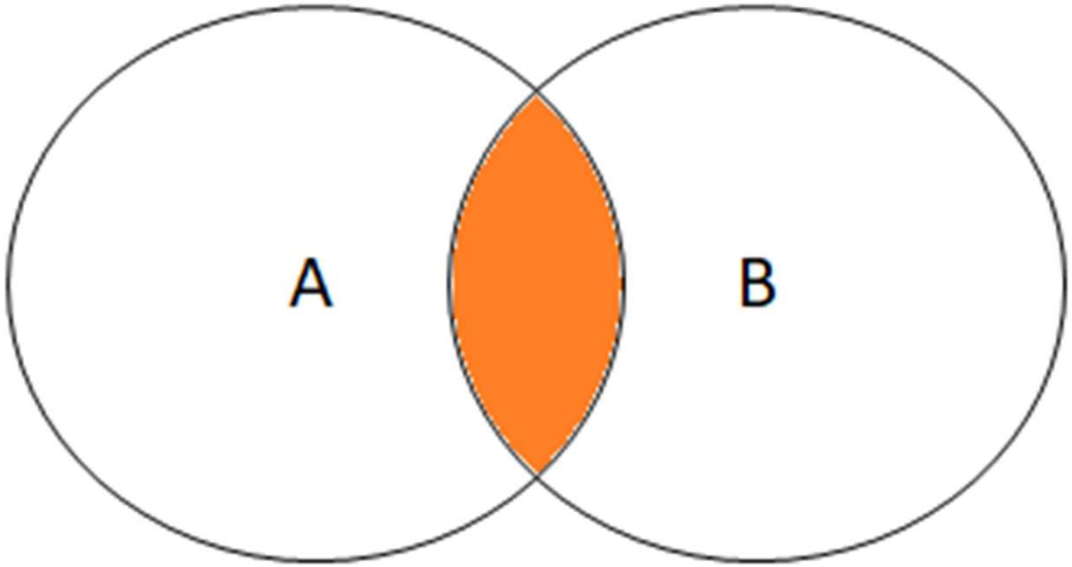
- Mit Joins kann man Daten kombiniert aus Tabellen erhalten. Dazu müssen die Tabellen in Relation stehen.
- 6 Arten von Joins:
  - **Inner Join**: Gibt Zeilen zurück, die in beiden Tabellen eine Übereinstimmung haben.
  - **Left (Outer) Join**: Gibt alle Zeilen der linken Tabelle und die übereinstimmenden Zeilen der rechten Tabelle zurück.
  - **Right (Outer) Join**: Gibt alle Zeilen der rechten Tabelle und die übereinstimmenden Zeilen der linken Tabelle zurück.
  - **Full (Outer) Join**: Kombiniert Left Join und Right Join – gibt alle Zeilen zurück, wenn es eine Übereinstimmung in einer der Tabellen gibt.

  -

  - **Cross Join**: Erzeugt kartesisches Produkt der beiden Tabellen – jede Zeile der einen Tabelle wird mit jeder Zeile der anderen Tabelle kombiniert.
  - **Self Join**: Eine Tabelle wird mit sich selbst verbunden, oft verwendet, um hierarchische oder sequentielle Daten zu analysieren.

INNER JOIN

- Inner Join selektiert alle Daten von beiden Tabellen in denen die WHERE Bedingung gültig ist:
- SELECT table1.column1,table1.column2, table2.column1,....
  FROM table1
  INNER JOIN table2
  ON table1.matching_column = table2.matching_column;



LEFT JOIN – RIGHT JOIN

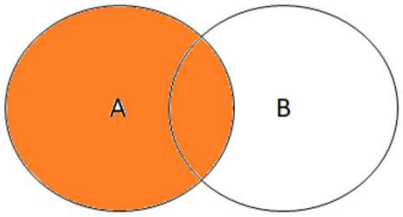
- Der Left Join returniert alle Zeilen der linken Tabelle und die passenden Zeilen der rechten Tabelle. Gibt es aus einem Eintrag der linken Tabelle keinen passenden Eintrag aus der rechten, wird für den Eintrag aus der linken Tabelle „null“ returniert. Das gleiche gilt für den Right Join nur mit der rechten statt mit der linken Tabelle

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```

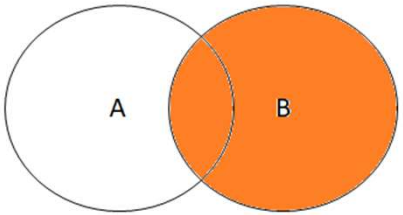
```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```

LEFT JOIN – RIGHT JOIN

- Left Join:

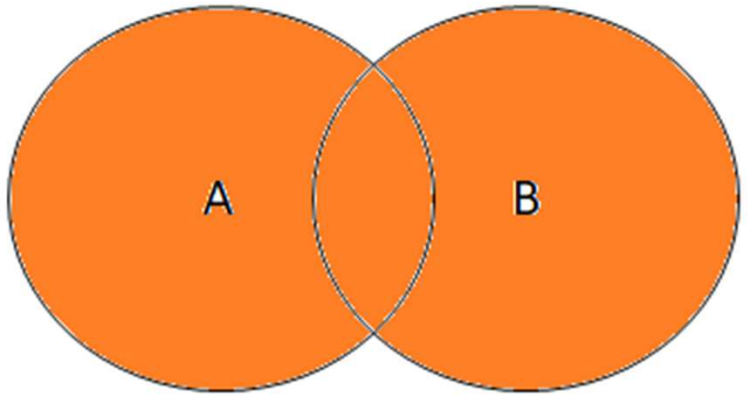


- Right Join:



FULL JOIN

- Full Join kombiniert alle Ergebnisse beider Tabellen. Für die Ergebnisse, für die es in der jeweils anderen keine Referenzierung gibt, werden NULL Werte returniert.



BEISPIEL JOIN

- Folgende Tabelle für Studenten ist gegeben:

StudentenID	Vorname	Nachname	Matrikelnummer	Alter
1	Max	Mustermann	2212345	20
2	Sandra	Müller	2267890	21
3	Mathias	Maier	2239847	20
4	Katharina	Mustermann	2276511	19

- Beispiel Join

StudentenID	KursID
1	1
1	4
1	5
2	1
3	3
3	4

INNER JOIN AUF STUDENTENID

StudentenID	Vorname	Nachname	Matrikelnummer	Alter	KursID
1	Max	Mustermann	2212345	20	1
1	Max	Mustermann	2212345	20	4
1	Max	Mustermann	2212345	20	5
2	Sandra	Müller	2267890	21	1
3	Mathias	Maier	2239847	20	3
3	Mathias	Maier	2239847	20	4

LEFT JOIN

StudentenID	Vorname	Nachname	Matrikelnummer	Alter	KursID
1	Max	Mustermann	2212345	20	1
1	Max	Mustermann	2212345	20	4
1	Max	Mustermann	2212345	20	5
2	Sandra	Müller	2267890	21	1
3	Mathias	Maier	2239847	20	3
3	Mathias	Maier	2239847	20	4
4	Katharina	Mustermann	2276511	19	NULL

RIGHT JOIN

StudentenID	Vorname	Nachname	Matrikelnummer	Alter	KursID
1	Max	Mustermann	2212345	20	1
1	Max	Mustermann	2212345	20	4
1	Max	Mustermann	2212345	20	5
2	Sandra	Müller	2267890	21	1
3	Mathias	Maier	2239847	20	3
3	Mathias	Maier	2239847	20	4
NULL	NULL	NULL	NULL	NULL	2

FULL JOIN

StudentenID	Vorname	Nachname	Matrikelnummer	Alter	KursID
1	Max	Mustermann	2212345	20	1
1	Max	Mustermann	2212345	20	4
1	Max	Mustermann	2212345	20	5
2	Sandra	Müller	2267890	21	1
3	Mathias	Maier	2239847	20	3
3	Mathias	Maier	2239847	20	4
4	Katharina	Mustermann	2276511	19	NULL
NULL	NULL	NULL	NULL	NULL	2

GEMEINSAME BEISPIELE – NORTHWIND DATENBANK

- Bearbeite die Beispiele aus dem File 2_Northwind_Joins.txt

ORDER BY

- **ORDER BY** wird verwendet, um Ergebnisse einer SQL-Abfrage in einer bestimmten Reihenfolge zu sortieren.
- Essentiell für die übersichtliche Darstellung von Daten.
- Standardmäßig erfolgt die Sortierung aufsteigend (ASC).
- Beispiel: **SELECT * FROM Tabelle ORDER BY Spalte1;**
- Beispiel: **SELECT * FROM Tabelle ORDER BY Spalte1 DESC;**
- Strings werden standarmäßig auch in aufsteigender alphabetischer Reihenfolge sortiert.

EINFÜHRUNG IN SUBQUERIES

- Ein Subquery ist eine SQL-Abfrage innerhalb einer anderen SQL-Abfrage.
- Subqueries sind besonders nützlich, wenn man Werte aus einer Tabelle filtern möchte, die auf einer Bedingung basieren, welche Daten aus einer anderen Tabelle beinhaltet.
- Subqueries können in verschiedenen Teilen einer SQL Abfrage verwendet werden.
- Zum Beispiel **SELECT * FROM Products WHERE UnitPrice > (SELECT AVG(UnitPrice) FROM Products)** = Auswahl aller Produkte mit einem Preis über dem Durchschnitt.
- Best Practices is die Vermeidung von komplexen Subqueries und die Verwendung von Joins wenn möglich.

VEREINFACHUNG UND KLARHEIT IN SQL – „AS“

- Das Schlüsselwort **AS** wird in SQL verwendet, um einem Tabellen- oder Spaltennamen einen temporären Alias (alternativen Namen) zuzuweisen.
- Aliases machen SQL-Abfragen leichter lesbar und verständlich, besonders bei komplexen Abfragen mit mehreren Tabellen.
- Aliases reduzieren den Schreibaufwand, indem sie es ermöglichen, lange Tabellen- oder Spaltennamen durch kürzere zu ersetzen.
- Beispiel: SELECT * FROM UnglaublichLangerTabellenNameDerEinfachUnheimlichNervigZumSchreibenIst AS shorty WHERE shorty.number > 50

VEREINFACHUNG UND KLARHEIT IN SQL - ALIASES

- Aliases sind besonders nützlich in JOIN-Operationen, um schnell auf die Tabellen zu verweisen.
- Beispiel: **SELECT o.OrderID, c.CustomerName FROM Orders AS o JOIN Customers AS c ON o.CustomerID = c.CustomerID**
- Bei der Verwendung von Subqueries können Aliases verwendet werden, um die äußere und innere Abfrage klar voneinander zu unterscheiden.
- Beispiel: **SELECT * FROM (SELECT OrderID, CustomerID FROM Orders) AS OrderInfo WHERE OrderInfo.CustomerID = 'ALFKI'**
- Aliases für Spalten werden oft verwendet, um berechneten oder abgeleiteten Spalten einen verständlichen Namen zu geben.
- Beispiel: **SELECT Price * Quantity AS TotalCost FROM OrderDetails**

VEREINFACHUNG UND KLARHEIT IN SQL - ALIASES

- Aliases können in GROUP BY und ORDER BY Klauseln verwendet werden, um die Abfrage zu vereinfachen.
- Beispiel: **SELECT CustomerID, COUNT(*) AS TotalOrders FROM Orders GROUP BY CustomerID ORDER BY TotalOrders DESC**
- Aliases werden eingesetzt, um Spalten in Ergebnismengen benutzerfreundliche Namen zu geben.
- Beispiel: **SELECT FirstName + ' ' + LastName AS FullName FROM Employees**

AGGREGATSFUNKTIONEN

- Aggregatsfunktionen führen eine Berechnung auf einer Datenmenge aus und liefern ein einzelnes Ergebnis.
- Wesentlich für die Zusammenfassung und Analyse großer Datenmengen.
- Funktionen wie COUNT(), SUM(), AVG(), MAX(), MIN()

COUNT() - ZÄHLEN VON DATENSÄTZEN

- Zählt die Anzahl der Zeilen in einer Tabelle.
- Beispiel: **SELECT COUNT(*) FROM Orders;** - Zählt alle Bestellungen.

SUM() - SUMMIERUNG VON WERTEN

- Berechnet die Summe von numerischen Werten einer Spalte.
- Beispiel: SELECT SUM(Quantity) FROM OrderDetails; - Summiert die Mengen aller Bestellungen.

AVG() - DURCHSCHNITTSBERECHNUNG

- Berechnet den Durchschnittswert einer numerischen Spalte.
- Beispiel: **SELECT AVG(Price) FROM Products;** - Ermittelt den durchschnittlichen Preis der Produkte.

MIN() UND MAX() - MINIMALE UND MAXIMALE WERTE

- Ermitteln den kleinsten und größten Wert einer Spalte.
- Beispiel:
  - **SELECT MIN(Price) FROM Products;** - Findet den niedrigsten Preis.
  - **SELECT MAX(Price) FROM Products;** - Findet den höchsten Preis.

GRUNDKONZEPT VON GROUP BY

- Gruppierung nach einer Spalte: "Gruppierung der Daten nach einem bestimmten Attribut."
- Beispiel: **SELECT Spalte1, COUNT(*) FROM Tabelle GROUP BY Spalte1; -** "Zählt, wie oft jeder Wert in 'Spalte1' vorkommt.,,
- Auch die Gruppierung der Daten nach mehr als einer Spalte ist möglich
- Beispiel: **SELECT Spalte1, Spalte2, COUNT(*) FROM Tabelle GROUP BY Spalte1, Spalte2; -** Zählt, wie oft jede einzigartige Kombination von Werten in 'Spalte1' und 'Spalte2' vorkommt.

GRUPPIERUNG MIT AGGREGATSFUNKTIONEN

- Aggregatsfunktionen sind besonders mächtig in Kombination mit GROUP BY
- Beispiel: **SELECT CategoryID,AVG(Price) FROM Products GROUP BY CategoryID;** - Berechnet den durchschnittlichen Preis in jeder Kategorie.

EINFÜHRUNG IN HAVING

- **HAVING** ist eine SQL-Klausel, die zum Filtern von Gruppenergebnissen verwendet wird, ähnlich wie WHERE, aber für Gruppen.
- WHERE filtert Zeilen vor der Gruppierung, HAVING filtert Gruppen nach der Gruppierung
- Wird in Kombination mit GROUP BY verwendet, um Bedingungen auf Gruppenebene zu setzen.
- Having wird benötigt, wenn eine Bedingung auf Aggregatwerte angewendet werden soll.
- Beispiel: HAVING SUM(Price) > 1000 – wählt Gruppen mit einem Gesamtpreis von über 1000.„
- Beispiel: "Filtert Kategorien, die mehr als 5 Produkte haben.„ =>

```
SELECT CategoryID, COUNT(*) AS ProductCount
FROM Products
GROUP BY CategoryID
HAVING COUNT(*) > 5;
```

EINFÜHRUNG IN DEN DATE-DATENTYP

- Der Date Datentyp speichert Datumswerte (Jahr, Monat, Tag)
- Darstellung von Datumswerten in verschiedenen Formaten möglich – z.B. YYYY-MM-DD, DD-MM-YYYY, etc.
- Extraktionsfunktionen: YEAR(date), MONTH(date), DAY(date)
- Addieren und Subtrahieren: DATE_ADD(), DATE_SUB()
- Vergleichsoperationen: =, >, <, >=, <=
- Wichtige Funktionen: CURDATE(), NOW(), DATEDIFF(), DATE_FORMAT()

ERSTELLEN VON TABELLEN

- **CREATE TABLE** ist eine SQL-Anweisung zur Erstellung einer neuen Tabelle in einer Datenbank.
- Wird verwendet, um die Struktur und das Schema der Tabelle zu definieren.
- Syntax: CREATE TABLE table_name (column1 datatype, column2 datatype, ...);
  - table_name: Der Name der zu erstellenden Tabelle.
  - column1, column2, ...: Die Spalten der Tabelle, jede mit einem Namen und einem Datentyp.

SPALTEN DEFINIEREN

- Jede Spalte hat einen Namen, der eindeutig in der Tabelle sein muss.
- Datentypen legen den Typ der Daten fest, die in der Spalte gespeichert werden können (z. B. INT, VARCHAR, DATE).
- Die Spalte(n), die den Primärschlüssel der Tabelle definieren, müssen festgelegt werden.

OPTIONALE CONSTRAINTS

- Constraints wie **UNIQUE**, **NOT NULL**, **PRIMARY KEY** und **FOREIGN KEY** können für Spalten hinzugefügt werden, um Datenintegrität zu gewährleisten.
- Beispiel: CREATE TABLE Mitarbeiter (ID INT PRIMARY KEY, Name VARCHAR(255), Geburtsdatum DATE, AbteilungID INT);
- Es gibt weitere Optionen wie **DEFAULT**-Werte, **AUTO_INCREMENT** und **CHECK** Constraints können für Spalten festgelegt werden.
- Beispiel:

```
CREATE TABLE Bestellungen (
  OrderID INT AUTO_INCREMENT PRIMARY KEY,
  ProductName VARCHAR(255),
  Price DECIMAL(10, 2) DEFAULT 2 CHECK (Price > 0)
);
```

CASCADE DELETE CONSTRAINT

- Man kann die Option "CASCADE DELETE" verwenden, um sicherzustellen, dass das Löschen eines Datensatzes in einer Tabelle automatisch auch alle verknüpften Datensätze in anderen Tabellen löscht.
- Z.B.: CREATE TABLE Orders (

```
OrderID int PRIMARY KEY,
CustomerID int,
OrderDate date,
….
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
   ON DELETE CASCADE
);
```

NÜTZLICHE LINKS

- Löschen eines Constraints: https://stackoverflow.com/questions/45014622/can-i-retroactively-add-a-foreign-key-id-column-and-populate-it-with-the-values
- Ny SQL Befehle und mehr: https://dev.mysql.com/doc/refman/8.0/en/

MANIPULATION VON DATEN

- Datenmanipulationssprache (DML) in SQL umfasst Befehle, die zum Einfügen, Aktualisieren und Löschen von Daten verwendet werden.
- Drei Hauptbefehle: "**INSERT**, **UPDATE**, **DELETE**."

INSERT - EINFÜGEN VON DATEN

- INSERT wird verwendet, um neue Datensätze in eine Tabelle einzufügen.
- Bei „Autoincrement-Feldern“ wird der Schlüsselwert automatisch erhöht.
- Grundstruktur ohne Autoincrement: INSERT INTO Tabelle (Spalte1, Spalte2) VALUES (Wert1, Wert2);
- Grundstruktur mit Autoincrement: INSERT INTO Tabelle (Spalte2) VALUES (Wert2); - Hier wird Spalte1 als Autoincrement-Feld angenommen.
- Autoincrement-Felder werden häufig als Primärschlüssel verwendet. Sie erleichtern die Eindeutigkeit und die Verwaltung von Datensätzen.
- Autoincrement-Felder werden entweder im Nachhinein festgelegt (mit **ALTER TABLE**) oder beim Anlegen der Spalte wird das Keyword „AUTO_INCREMENT“ hinzugefügt.

UPDATE - AKTUALISIEREN VON DATEN

- UPDATE modifiziert die Werte bestehender Datensätze.
- Grundstruktur: UPDATE Tabelle SET Spalte1 = Wert1 WHERE Bedingung;
- Beispiel: Aktualisieren des Preises eines Produkts.

DELETE - LÖSCHEN VON DATEN

- DELETE entfernt Datensätze aus einer Tabelle.
- Grundstruktur: "DELETE FROM Tabelle WHERE Bedingung;„
- Beispiel: Entfernen eines veralteten Kunden aus der Kundentabelle.

TABELLEN VERÄNDERN – ALTER TABLE

- Mit **ALTER TABLE** kann eine bestehende Tabelle modifiziert werden, einschließlich Hinzufügen, Löschen oder Ändern von Spalten.
- Zum Beispiel kann ein bestehendes Feld nachträglich als Autoincrement-Feld definiert werden, sofern es sich um eine Primärschlüsselspalte handelt.
- ALTER TABLE Produkte MODIFY COLUMN ProduktID INT AUTO_INCREMENT;