Datenbanken Cheatsheat

- Datenbankentwurf

- Phasen des Datenbankentwurfs
  - Anforderungsanalyse —Anforderungsspezifikation—>
    - Welche Organisationseinheiten mit DB arbeiten
    - Welche Prozesse damit unterstützt werden sollen
    - Welche Daten in welcher Form gespeicehrt werden
    - WIe die Daten strukturiert sind (wichtig für Auswahl der DB)
    - Wie die technischen Anforderungen sind (Anzahl Zugriffe, Datenmengen, etc.)
    - Phasen nach Kempler: Identifikation von organisationseinheiten, Identifikation der zu unterstützdenden Aufgaben, anfroderungs-Sammelplan, Anforderungs-Sammlung, Filterung, Satzklassifikationen, Formalisierung

    - Entwurf Objektbeschreibung
      - Klassename
      - Anzahl
      - Attribute (eigenschaften)
        - Typ (, Länge), Wertebereich, Identifizierend
    - Entwurf Beziehungsbeschreibung
      - Beteiligte Objekte (mit Rollen)
      - Attribute der Beziehung
      - Anzahl
    - Entwurf Prozessberschreibung
      - Häufigkeit
      - Benötigte Daten
      - Priorität
      - Zu verarbeitende Datenmenge
  - Konzeptueller Entwurf —Informationsstruktur—>
  - Implementationsentwurf —Logische DB-Struktur—>
  - Physischer Entwurf —>Physischhe DB-Struktur

- Konzeptueller Entwurf:
  (1. Schritt: Konzeptuelle Modellierung (Abbildung der Sitation in Form von Objekten und Beziehungen , z.b. mit ER-Modell, alternativ z.b. UML)))
  - Abbildung von Wirklichkeit/Realwelt in verarbeitbares Modell: Konzeptuelles Schema (ER-Schema)
    - Entitätstyp:
      - Kategorisiierung (Zusammenfassung/Abstraktion) gleichartiger Entitäten (gleiche Eigenschaften, aber unterschiedliche Eigenschaftswerten). (Dabei is für Kategorisierung nur deren Anzahl und Art, nicht aber die Werte entscheidend)
      - Rechteck
    - Entität: Unterscheidbare (identifizierbare) pysisch oder gedanklich existierende Konzepte der zu modellierenden Welt.
      - Bsp. Janis, Patrik
      - Unterheiden sich durch ihre Attribute (Attributswerte/Eigenschaftswerte)
    - Attribut (Eigenschaft)
      - Charakterisieren Entität, Entitätstyp, Beziehung, Beziehungstyp
      - Besitzen
        - Name
        - Wert
          - aus Domäne
            - zulässiger Wertebereich eines Attributs
            - z.b. Natürliche Zahlen N; Fest vorgeschriebene Werte (Montag, Dienstag, …, Sonntag); Bereiche (wie 0-10.000)
      - Ellipse
    - Beziehung
      - Drücken Wechselwirkung/Abhängigeit von Entätaten aus
      - können binär (hören), ternär (prüfen) oder n-är sein
      - Können Eigenschaften besitzen
      - Beispiele

    - Janis hört DB1 (Janis, DB1), Gärtner liest DB1 (Binär)
    - Gärtner prüft Bob in DB mit Note 1,3 (ternär, mit Eigenschaft)
- Beziehungstyp
    - Abstraktion/Kategorisierung gleichartiger Beziehungen
    - Raute
    - z.b. prüfen, hören, lesen
    - Rollen
        - zum genaueren beschreiben der Instanzen des Relationstyps
        - bsp für Beziehungstyp vorraussetzen: Vorlesung x Vorlesung
            - (Vorgaenger : v_1, Nachfolger : v_2)

- Schlüssel
    - (Eine Entität wird durch Kombination aller ihrer Attributwerte eindeutig beschieben (sonst nicht unterscheidbar vgl. def Entitäten).) Im Allgemeinen reicht ein Teil der Attribute (Teilmenge9), um eine Entität eindeutig zu beschreiben. Eine minimale identifizierende Teilmenge wird Schlüssel genannt.
    - Primärschlüssel
        - Sind mehrere Schlüsselkandidaten vorhanden, wählt man einen als sogenannten Primärschlüssel aus.
        - Attribute die zu Primärschlüssel gehären werden unterstrichen
        - Manchmal sind die “natürlichen Attribute” nicht ausreichend, ann wird ein künstliches Attribut als Primärschlüssel hinzugefügt (z.B. PersNr)
    - Superschlüssel
        - Eine Menge von Attributen a $\subseteq$ R ist ein Superschlüssel fals gilt: a—>R, also: die Attribute a bestimmen alle anderen Attribute der Relation R
    - Kandidatenschlüssel
        - a $\subseteq$ R ist ein Kandidatenschlüssel, falls a —>▪ R, also, falls die Relation voll Funktional Abhängig ist von a
            - Volle funktionale Abhängigkeit (a —>▪ b): b ist voll funktional abhängig von a, gdw gilt:
                - a —> b und
                - a kann nicht mehr verkleinert verden, also $\forall A \in a.\neg((a-\{A\}) \text{ —>▪ } b)$
        - in anderen Worten: ein Kandidatenschlüssel ist ein minimaler Sueprschlüssel, von ihm hängt also die gesamte Relation ab
- Kardinalität (in ER)
    - Mengenangabe mit denen für jeden Beziehungstyp festgelegt wird wieviele Entitäten eines Entitätstyp mit genau einer Entität des/der anderen am Beziehungstyp beteiligten Entitätstyp(s)/en (oder umgekehrt) in Beziehung stehen können oder müssen.
    - Achtung: Kardinalität in DB: Kardinalität einer Menge/Tabelle = Anz der Elemente/Zeilen
    - Funktionalitäten
        - 1:1, 1:N, N:1, N:M, N:N
        - ist 1 in zielrichtung, so hat man partielle Funktion (vgl Algebra: Partielle Funktion: Jedem x wert (Element aus A) ist genau ein oder kein y wert (Element au B) zugeordnet)
        - Jede Beziehung ist eine N:M-Beziehung (ist nur für andere Beziehungstypen zu unspezifische Angabe)
        - bei Ternären Beziehungstypen (bez(A, B, C))
            - bei A steht 1 (der Beziheungstyp A hat Kardinalität 1), wenn gilt:
              $B \times C \text{ —>}_p A$
              wenn also A funktional abh ist von der Kombination der übrigen am Beziehungstyp beteiligten Entitätstypen (hier B und C). (bzw. die Abbildung von B,C nach A ist eine partielle Funktion)
            - ansonsten ein Großbuchstabe
        - bsp: [Professoren] —1—<lesen>—N—[Vorlesungen] bedeutet: Ein Professor ließt N Vorlesungen. Eine Vorlesung wird von maximal einem Professor gelesen.
    - (min,max)-Notation
        - auf der anderen Seite geschrieben
        - steht bei E_i (min_i, max_i), so gibt es mindestendens min_i Tupel der Art (…, e_i, …) und höchstens max_i Tupel der Art (…, e_i, …) ε R

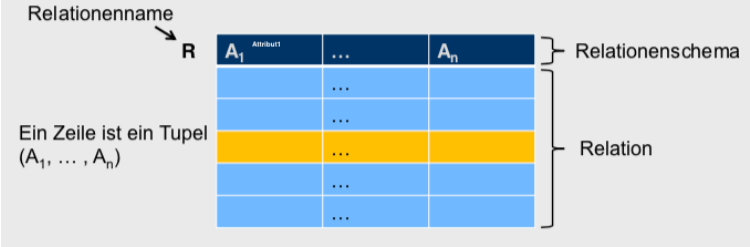
        - [Pflanze] —(0,1)—<ist in>—(0,*)—[Blumentopf] Eine Pflanze ist in maximal einem Blumentopf. Ein Blumentopf beinhaltet beliebig viele Pflanzen.
    - Spezielle Konzepte
        - Schwache/existenzabhängige Entitätstypen
            - Beziehung zw. “starken” und schwachem Typ ist immer 1:N (oder selten auch1:1).
            - Schlüssel der schwachen Entität ist nur in Kombination mit dem Schlüssel des Entitätstyp, von dem sie abhängig ist eindeutig
            - Schwacher Entitätstyp und beziehungstyp doppelt umrandet
                - bsp: Raum liegt in Gebäude, Schlüssel für Raum ist GebNr und RaumNr
        - Generalisierung/Spezialisierung
            - is-a Beziehung mit Pfeilen in richtung der Generalisierung (von Sub- zu Supertyp)
        - Aggregation
            - Teil-von (is-part-of) Beziehung
            - Komponenten bilden zusammen Aggregat
        - Relation/Tupelmenge (aus mengenlehre)
        - Jede Teilmenge eines Produktes A_1 x A_2 x … x A_n heißt Relation über A_1 … A_n
        - bsp:
            - Namen: {Janis; Bob; Dana}
            - MatrNrs: {563; 512; 501, 555}
            - Geschlechter: {m; w}
            - Eine Relation ist Teilmenge des kartesischen Produkts
              R ⊆ Namen x MatrNrs x Geschlechter
            - Beispielrelation R_B = {(Janis, 563, m), (Bob, 512, m), (Bob, 501, m), (Dana, 555, w)}
            - Relationen können als Tabelle dargestellt werden.

Name	MatrNr	Geschlecht
Janis	563	m
Bob	512	m
Bob	501	m
Dana	555	w

        - Auspägung des Beziehungstyps R stellt eine Teilmenge des kkarrtesischen Produktes der an der Beziehung beteiligten Entitätstypen dar: R ⊆ E_1 x E_2 x … x E_n
            - n ist Grad der Beziehung
            - Ein Element (e_1, e_2, e_3, …, e_n) nennt man eine Instanz des Beziehungstyps, wobei e_i ε E_i für alle 1 <= i <= n gelten muss

- Implementationsentwurf (Entwurf des Implementierungsschemas)
  (2. Schritt: Datenbankentwurf (Entwurf des Implementierungsschmas) (Von Konzeptuellem Schema nach Implementierungsschema))
    - Relationales Schema
        - Definitionen: Seien D_1, D_2, …, D_n Domönen (Wertebereiche)
          Relation: R ⊆ D_1 x … x D_n (z.b. Kunden ⊆ integer x string x string
          Tupel: Eintrag in Relation
          Schema: legt Struktur der gespeicherten Daten fest
            Schreibweise: Kunden: {[Kundennummer: integer, Name: string, Vorname: string]}
              […] Tupelkonstruktor, {…} Mengenkonstruktor
        - Tabellen
            - sowohl Entitäten, als auch Beziehungen werden als Relationen (Tabellen) repräsentiert
            - Zeile = Datensatz
            - Eigenschaften von Tabellen
                - Einträge in Spalte (Attributwerte von selben Attribut) haben denselben typ (oder NULL)
                - Alles Zeilen sind verschieden (keine mehrfach)
                - Tabellennamen sind eindeutig

- Bedeutung von Spalte durch Attributname gekenzeichnet



  - Schema: Beschreibung der Struktur
    - Name der Tabelle = Name der Relation
    - Spaltenüberschrift = Bezeichnung des Attributs
  - Ausprägung: der aktuelle Zustand der Datenbasis
  - Schlüssel: minimale Menge von Attributen, deren Wertte ein Tupel eindugtig identifizieren
  - Primärschlüssel: wir unterschstrichen
    - Einer der Schlüsselkandidaten wird als Primärschlüssel ausgewählt
- Vorgehen:
  - Für jeden Entitätstypen eine Relation (Tabelle) anlegen
    - Notation:
      <RelationsName>:{[<Attributname$_1$>:<Attributtyp$_1$>, …,
      <Attributname$_n$>:<Attributtyp$_n$>]}
      Studenten: {[MatrNr: integer, Name: string, Semester: integer]}
      - Bestandteile des Primärschlüssels unterstrichen
      - wir dürfen den Datentyp weglassen
  - Für (N:M) Beziehungen Relation anlegen
    - Fremdschlüssel: ein(e) Attribut(kombination), welche auf Primärschlüssel einer anderen/der gleichen Relation verweist
  - 1:N, N:1 Beziehungen können zusammengefasst werden indem die Relation der der 1 gegnüberliegenden Seite einen Fremdschlüssel auf die andere relation enthällt. (Die Relation die die andere eindeutig bestimmt enthällt ihren Fremdschlüssel)
    —> (Nur) Relationen mit demselben Schlüssel können zusammengefasst werden
- Gute Relationale Schema
  - Redundante Daten bei Füllung von Tabelen vermeiden
  - Vermeiden von NULL-Einträgen (soweit möglich)
    - wenn z.b. ein Fremdschlüssel meist NULL ist lieber die Beziehung als separate Relation modellieren. Bsp: [Mensch]—[istAbgeordneter]—[Land]
  - Unter berücksichtigung der Regeln mäglichst kleine Tabellenzahl

- Netzwerk Schema
- Objektorieintiertes Schema (mit halbautomatischer Transformation möglich)

- 3. Schritt: Optimierung der Datenbank auf Speicherebenene
  - Indexstrukturen z.B. Suchbäume, Hashverfahren
  - Replikation
  - Physische Ebene ist unabhängig von der Implementationsebene

- **Gründe** für (Datenintegration durch) **Datenbanken** (genormte zentrale Datenhaltung mit einheitlichen Schnittstellen für):
  - große Mengen von Daten effizient verarbeiten
  - Speichereffizenz
  - Reduktion der Entwicklungskosten
    - Daten sind unabhängig von Darstelllung und den verwendeten Rechner
  - Vermeidung von Redundanzen
  - Vermeidung von Inkonsistenzen —> konsistenz (Korrektheit der in der DB gespeicherten Daten)

- Erweiterung der Zugriffsmöglichkeiten (verknüpfung von Daten)
- Ermöglichung des Mehrbenutzerbetriebs (paraleles Arbeiten mehrerer Benutzer auf den gleichen Daten)
- Vermeidung von Datenverlust (Datensicherheit)
- Vermeidung von Integritätsverletzungen
- Beschränkungen des Datenzugriffs (Datenschutz)

- **Datenbanksystem (DBS)**: ein System zur elektronischen Datenverwaltung
  - Aufgaben:
    - große Datenmengen effizient, widerspruchsfrei und dauerhaft speichern,
    - benötigte Teilmengen in unterschiedlichen, bedarfsgerechten Datstellungsformen für Benutzer und Anwendungsprogramme bereitstellen
  - DBS besteht aus zwei Teilen:
    - Datenbankmanagementsystem (DBMS)
      - Software zur Verwaltung von Datenbanken (Verwaltungssoftware)
      - organisiert intern die struktuierte Speicherung der Daten und kontrolliert alle lesenden und schreibenden Zugriffe auf die DB
      - Beispiele:
        - (Objekt-) Relationale DBMS
          - kommerziel: Oracle 11g, IBM DB2, Microsoft SQL-Server
          - Freie: MySQL, PostgreSQL, Ingres, FireBird
        - …
        - NoSQL
          - MongoDB, …
    - Datenbank (DB)
      - Strukturierter Datenbestand (z.b. Tabellen) (Menge der zu verwaltenden Daten)

- Prinzipien: **Die neun Codd'schen Regeln**
  (gewährleisten zuverlässigen und effizienten Datenbankbetrieb; beschreiben, was DB unterstützen muss um tatsächlich relational zu sein)
  1. Integration: einheitliche, nichtredundante Datenerwaltung
  2. Operationen: Speichern, Suchen. Ändern
  3. Katalog: Zugriffe auf Datenbankbeschreibungen im Data Dictionary
  4. Benutzersichten
  5. Integritätssicherung: Korrektheit des Datenbankinhalts
  6. Datenschutz: Ausschluss unauthorisierter Zugrifffe
  7. Transaktionenn: mehrere DB-Operationen als Funktionseinheit
  8. Synchronisation: parallele Transaktionen koordinieren
  9. Datensicherung: Wiederherstellung von Daten nach Systemfehlern
- Grundprinzip der Datenabstraktion
  - Externe Schicht: Definition von Teilmengen von Daten, die für spezielle Nutzergruppen sichtbar sein sollen
  - Logische Ebene: Schema zur Definition der Daten, die in DBS abgelegt werden
  - Physische Ebene: Festlegung der Speicherstrukturen

- Integrität in Datenbanken
  - Automatische Überprüfugn semantischer Integritätsbedingungen:
    - Eigenschaften, die aus dem konzeptuellen Modell abgeleitet werden können (Beziehungen, Kardinalitäten, existenzabhängigkeit, …)
    - Integritätsbedingungen: Schlüssel, Beziheungskardinalitäten, Attributdomänen, Inklusion bei Generalisierung
    - statische Integritätsbedingungen
      (Bedingungen an den Zustand der Datenbasis)
      - Befüllungdes Attributs: … NOT NULL …
      - Wertebereichseinschränungen
        - … CHECK <AttrName> BETWEEN <$wert_1$ > AND <$wert_2$>
      - Aufzählungstypen
        - … CHECK <AttrName> IN (<$wert_1$>, <$wert_2$>, …, <$wert_n$>)
    - dynamische Integritätsbedinungen
      (Bedingungen an Zustandsübergänge)
    - Vorteile

        - Konsitenz des Datenbestands wird von DB und nicht von jedem einzelnen AWP sichergestellt
        - Konsistenzcheck der Daten ist einfach abschaltbar (z.b. für Perfomaz sinnvoll bei Datenimport)
    - Einhaltung Referentieller Integrität
        - Integritätsbedinungen müssen überprüft werden bei
            - Einfügen neuer Datensätze
            - Ändern/Löschen von Datensätzen
        - Strategien bei Verletzung der Integritätsbedinungen
            - Defult: Zurückweisen der Änderungsoperation
            - Propagieren der Änderungen: CASCADE
            - Verweise auf Nullwert setzen: SET NULL
- Erhaltung der Konsistenz der Daten
    - Schutz vor unerlaubter Manipulation von Daten
    - Schutz vor Inkonsistenzen bei Mehrbenutzerbetrieb und Systemfehlern

SQL (Structured Query Language)
- Tabelle Erzeugen
  <CREATE TABLE <tab-name>(
    <att_1-name> <att_1-typ> <$constraint_{1,1}$> <$constraint_{1.2}$> … <$constraint_{1.n}$>,
    <att_2-name> <att_2-typ> <$constraint_{2,1}$> … ,
    <att_n-name> <att_n-typ> <$constraint_{n,1}$> …
  );
  - att-typ (Datentyp): Definiert Wertebereiche und zulässige Operationen
    - integer
    - numeric(p,s) = decimal(p,s)
        - per default (precision, scale) = (18, 0)
            - precision: Max Anzahl an Ziffern (Vor und nach Komma)
            - scale: Maximale Anzahl an Nachkommastellen
            - bsp: 123,56 has precision = 5, scale = 2
    - char(n) = character(n)
        - Zeichenkette mit fester länge n (kürzere mit Leerzeichen aufgefüllt)
        - wenn Werte immer feste länge haben perfomanter als varchar
    - varchar(n) = character varying(n)
        - Zeichenkette mit variabler Länge (Höchstgrenze: n)
    - text
        - Zeichenkette ohne Höchstgrenze
    - date
        - ‘yyyy-mm-dd’
    - daetime
        - ‘yyyy-mm-dd hh-mm-ss’
  - constraint: Bezeichner einer Bedinung, die dur DBS automatisch überprüft wird
    - PRIMARY KEY, NOT NULL
    - UNIQUE
      —> Kandidatenschlüssel
    - CHECK (Bedingung)
        - <att-name> <att-typ> CHECK (att-name …
            - in (<$wert_1$>, <$wert_2$>, …, <$wert_n$>)
            - between <min> and <max>
        - bsp: Note NUMERIC(2,1) CHECK (Note between 0.7 and 5.0)

    - REFERENCES <tabname>(<keyattribute>)
        - ON DELETE SET NULL
        - ON DELETE CASCADE
        - ON UPDATE SET NULL
        - ON UPDATE CASCADE
        - bsp: <att_x-name> <att_x-typ> REFERENCES <tabname>(<keyattribut>) ON DELETE CASCADE
  - bsp: CREATE TABLE kunden(nr integer primary key, name varchar(20), adresse varchar(20));
  - Primärschlüssel
    - als constraint

    - PRIMARY KEY (att_1, att_2)
  - Referenzielle Integrität / Fremdschlüssel
    - als constraint
    - FOREIGN KEY (<att-name>) REFERENCES <tabname>(<keyattribute)
    - bsp:
      userID INTEGER,
      FOREIGN KEY (userID) REFERENCES Users(userID) ON DELETE SET NULL

- Tabelle löschen
  DROP TABLE <tab-name>;

- Einfügen von Datensätzen
  INSERT INTO <tab-name> (<att_1>, <att2>, …, <att_n>) VALUES (<val_1>, <val_2>, …, <val_n>);
  - bsp: INSERT INTO kunden (nr, name, adresse) values (1,‘Hans Wurst‘, ‘Berliner Strasse 20’);
  - Sind Attribute nicht angegeben wird NULL eingefügt
  - auch mit verschachtelter Abfrage möglich
    INSERT INTO hören
      SELECT MatrNr, VorlNr
      FROM Studenten, Vorlesungen
      WHERE Titel= ‘Logik’;
    (fügt der tabelle hören hinzu, dass jeder Student Logik hört)

- Verändern von Datensätzen
  UPDATE <tab-name> SET <att-name> = <newval>
    WHERE <condition>
  - bsp: UPDATE kunden SET adresse=‘Franz-Mehring-Platz’ WHERE nr=1

- Löschen von Datensätzen
  DELETE FROM <tab-name> WHERE <bed>
  - bsp: DELETE FROM kunden WHERE nr=1

- Abfragen (queries)
  SELECT <att_1>, <att_2>, …, <att_n>
    FROM <tab_1>, <tab_2>, …, <tab_n>
    WHERE <bed>
  - Abarbeitungsreihenfolge:
    - FROM, WHERE, GROUP BY, HAVING, SELECT (evt. mit Aggregationsfunktion)

  - SELECT
    - Duplikateliminierung: DISTINGT
      - bsp: SELECT DISTINCT rang FROM Professoren
    - Umbenennung von Tabellen
      - SELECT s.Name, h.VorlNr FROM Studenten s, hoeren h WHERE s.MatrNr = h.MatrNr;
    - Sortieren: ORDER BY attr1 [<Richtung>, attr2 <Richtung,…];
      - Richgungen:
        - DESC = descending = absteigend
        - Default: ASC = ascending = aufsteigend
      - bsp: SELECT PersNr, Name, Rang FROM Professoren ORDER BY Rang DESC, Name ASC;
  - Case Anweisung (Anzeige der Note als Text)
    - SELECT MatrNr, (CASE WHEN Note < 1.5 THEN ‘sehr gut’
      WHEN Note < 2.5 THEN ‘gut’
      WHEN Note < 3.5 THEN ‘befriedigend’
      WHEN Note <= 4 THEN ‘ausreichend’
      ELSE ‘Nicht bestanden’ END)
      FROM pruefen

  - FROM

  - WHERE
    - Arbeitet auf Tupelebene, wird vor Gruppierung abgearbeitet (Daher keine Aggregation außer in Subqueries)

    - Vergleiche mit LIKE
        - % : beliebig viele/kein beliebiges Zeichen
        - _ genau ein beliebiges Zeichen
        - SELECT * FROM Studenten WHERE Name LIKE 'J%i_' = Studenten, deren Name mit J beginnt und i als vorletzten Buchstabe hat
    - Mengenvergleiche
        - IN / NOT IN
            - Welcher Professor liest keine Vorlesung?
              SELECT Name FROM Professoren WHERE PersNr NOT IN
                (SELECT gelesenVon FROM Vorlesungen);
            - Welche Vorlesung hat keinen Nachfolger?
              SELECT Titel FROM Vorlesungen WHERE VorlNr NOT IN
                (SELECT Vorgaenger FROM voraussetzen);
        - ALL
            - Wie sind die Namen der Studenten, die am längsten studiert haben?
              SELECT Name FROM Studenten WHERE Semester >= ALL
                (SELECT Semester FROM Studenten);
    - a BETWEEN 1 AND 4
        - enpricht a >= 1 AND a <= 4
        - entpsricht a IN (1,2,3,4)
- Verknüpfung von unterschiedlichen Daten
- Mengenoperationen
    - Schemagleichheit erforderlich
    - Vereinigung zweier Tabellen (gleiches Schema, Datensätze von beiden Tabellen enthalten)
        - UNION eliminiert Duplikate, UNION ALL nicht
        - bsp: (SELECT name FROM Assistenten) UNION (SELECT name FROM Professoren) UNION (SELECT name FROM Studenten)
- Aggregatfunktionen
    - Menge von Tupeln wird zusammengefasst zu einem Wert
    - DISTINCT : nur verschiedene werte werden berücksichtigt, kann angewendet werden, wenn argument nicht * (mehrere Spalten nicht unterstützt). (bsp siehe count)
    - Mit GROUP BY auf jede Gruppe, statt auf gesamte Tabelle angewendet.
        - GROUP BY
            - Aufteilung der einträge in Gruppen anhand der Attributwerte
            - Pro Gruppe ein Ergebnistupl
                - Daher müssen alle Attribute aus SELECT-Klausel außer den aggregierten auch in der GROUP BY Klausel stehen
                    - hätte man z.b. zusätzlich zu den Attributen aus GROUP BY ein weiteres Attribut in SELECT-Klausel, so ist nicht eindeutig was dessen Wert ist
                        - MySQL nimmt beliebigen Wert, PostgreSQL schmeißt Fehler
    - COUNT(*)
        - Zählt die Anzahl der Datensätze (Zeilen) (unabhängig von Wert)
        - Wieviele Studenten gibt es
          SELECT COUNT(*) FROM Studenten
    - COUNT(attribut)
        - Zählt die Anzahl der Datensätze, bei denen der Wert des angegebenen Attributs nicht NULL ist.
        - COUNT(DISTINCT attribut) Zählt alle Einträge, wobei die Duplikate eliminert werden.
            - Wieviele verschiedene Noten wurden vergeben?
              SELECT COUNT(DISTINCT Note) FROM pruefen
    - AVG(attribut)
        - Durchschnittswert des Attributs
        - DUrchschnittssemesterzahl aller Studenten
          SELECT avg(Semester) FROM Studenten
        - Durchschnittliche Prüfungsnote der einzelnen Studenten
          SELECT s.MatrNr, s.Name, AVG(Note) FROM Studenten s, pruefen p WHERE s.MatrNr = p.MatrNr GROUP BY s.MatrNr, Name
    - MAX(attribut)
        - Maximaler Wert des angegebenen Attributs
        - Semesteranzahl des Studenten, der am längsten studiert hat
          SELECT MAX(Semester) FROM Studenten

    - Wie sind die Namen der Studenten, die am längsten studiert haben? (vgl. ALL)
      SELECT Name FROM Studenten WHERE Semester >= (SELECT MAX(Semester) FROM Studenten);
  - MIN(attribut)
  - SUM(attribut)
- HAVING
  - Für Überprüfungen auf Gruppenebene (die also einen Aggregationsoperator verwenden)

- Behandlung von NULL-Werten
  - Boolsche Ausdrücke haben drewertige Logik (Nicht nur true, false, sondern auch unknown).
  - Unknown, wenn mindestens eines der Argumente von Vergleichsoperator NULL ist und das andere nicht schon das ergebnis bestimmt (z.b. and(true, unknown) = unknown):

not	
true	false
unknown	unknown
false	true

and	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

or	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

  - in WHERE-Bedinungung werden nur TUpel weitergereicht für die das Prädikat true ist. Tupel, deren Bedinung zu unknown ausgewertet ist werden also nicht ins Ergebnis aufgenommen.
  - Bei Gruppierung wird NULL als eigenständiger Wert aufgefasst und in eine eigene Gruppe eingeordnet

- Geschachtelte Anfragen
  - in SELECT-Klausel
    - für jedes Ergebnistupel wird die Unteranfrage ausgeführt
    - erfordert Bennung der Unteranfrage mit AS
    - Unteranfrage ist korreliert (greift auf Attribute der umschließenden Anfrage zu)
    - SELECT PersNr, Name, (SELECT SUM(SWS) AS Lehrbelastung FROM Vorlesungen
      WHERE gelesenVon=PersNR)
      FROM Professoren;
  - in FROM-Klausel
    - erfordert Benennung der Unteranfrage (ohne/mit AS-Stichwort)
    - Welche Studenten haben mehr als zwei vorlesungen gehört? (alt. mit HAVING statt verschachtelung)
      SELECT tmp.* FROM
      (SELECT s.MatrNr, s.Name, count(*) as anzVorl FROM Studenten s NATURAL JOIN
      hoeren h GROUP BY s.MatrNr, s.Name) tmp
      WHERE tmp.VorlAnzahl>2

- psql (PostgreSQL interaktive terminal)
  \? : Anzeige aller Kommandos der psql-Konsole
  \h : Anzeige aller SQL-Befehle
  \h <befehlsname> : Hilfe zu Befehl
  \d : Anzeige aller Relationen
  \d <tabellenname>: Anzeige des Schemas der Tabelle

Relationale Algebra und Operatorbäume
- ermöglicht Relationen miteinander zu verknüpfen
- definiert Operationen, die sich auf Menge von Relationen anwenden lassen, welche wiederum Relationen als Ergebniss haben. Basis für SQL
- $\sigma_{\text{Selektionsprädikat}}$ Selektion (WHERE)
  - Filtert die Relation mit nach Tupeln die das Selektionsprädilkat erfüllen
  - Selektoionsprädikat ist ein Boolescher Ausdruck
    - arithmetische Vergleichsoperatoren =, <> bzw ≠, < , >, ≤, ≥
    - logische Operatoren ∨, ∧, ¬
    - bezieht sich immer nur auf eine Zeile

- $\sigma_{Name='Sokrates'}$(Professoren)

- ∏ Projektion (SELECT)
  - Extahiert die angegebenenn Attribute/Spalten einer Tabelle
  - $\prod_{Attributname1, Attribuname2, ...}$(Relationenname)
  - $\prod_{Rang}$(Professoren)
  - Eventuell auftretende Duplikate werden in RA entfernt, nicht aber in SQL (dafür SELECT DISTINCT)

- x Kreuzprodukt (,)
  - Elemente verknüpfen, Ergebnissrelation hat alle Attribute beider Relationen (Schema der Ergenisrelation sch(B x B) ist vereinigung der Attribute aus sch(A) und sch(B)
  - $A \times B = \{(a; b) \mid a \in A \wedge b \in B\}$
  - bsp: A= {a,b,c}, B ={1,2,3}, A x B = {(1,a),(1,b),(1,c),(2,a),(2,b),…,(c,3)}
  - Riesige Zwischenergebnisse, daher lieber Join wo möglich

- ⋈ Join / Verbund (INNER JOIN)
  - Theoretisch perfomanter, da weniger Zwischenergebnisse berechnet. Da Anfrageoptimierer der DB vor ausführen optimierende Umformungen vornimmt nicht notwendig.
    - EXPLAIN anfrage : gibt optimierten Operatorbaum der ANfrage aus
  - ⋈ bzw. A (NATURAL JOIN)
    - Einmal wird die doppelte Spalte eliminiert
    - $R(A_1, \ldots, A_m, B_1, \ldots, B_k)$
      $S(B_1, \ldots, B_k, C_1, \ldots, C_n)$
      $R \bowtie S = \prod_{A1, ..., Am, B1, ..., Bk, C1, ..., Cn}(\sigma_{R.B1=S.B1 \wedge \ldots \wedge R.Bk = S.Bk}(RxS))$
      SELECT * FROM R NATURAL JOIN S;
  - Theta Join: $\bowtie_{tab1.attr = tab2.attr}$ (… JOIN … ON …)
    - $R \bowtie_{Prädikat} S = \sigma_{Prädikat}(RxS)$ (Entspricht Kartesischen Produkt inklusive Selektion)
    - Das Selektionsprädikat ist ein beliebiges Prädikat
    - Wird auf Gleichheit von Attributen selektier, sagt man auch Equi-Join
      - ist veralgemeinerung von INNER JOIN … ON, bei welchem nur auf gleichheit der Attribute verglichen wird
    - bsp: Proffessoren $\bowtie_{Boss = Professoren.PersNr \wedge Professoren.Gehalt < Assistenten.Gehalt}$ Assistenten
      SELECT * FROM Professoren JOIN Assistenten
      ON Boss = Professoren.PersNr ∧ Professoren.Gehalt < Assistenten.Gehalt

- ⟕ LEFT OUTER JOIN bzw. LEFT JOIN
  - Elemente der linken Tabelle, die keinen JOIN-Partner haben bleiben erhalten
- ⟖ RIGHT OUTER JOIN bzw. RIGHT JOIN
  - Elemente der rechten Tabelle, die keinen JOIN-Partner haben bleiben erhalten

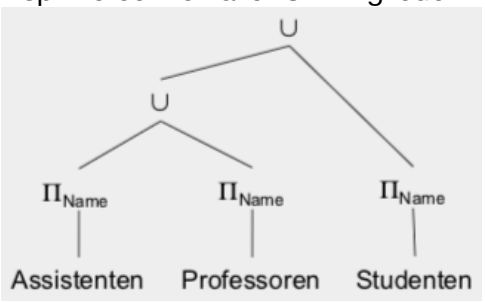
- ⟗ FULL OUTER JOIN bzw. FULL JOIN
  - in beiden Tabellen bleiben Elemente erhalten, die keinen JOIN-Partner haben
- ⋉ LEFT SEMI JOIN
  - Wie NATURAL JOIN, ergebnisstabelle hat jedoch nur die Spalten der Linken Tabelle.
    - SELECT DISTINCT lTable.* FROM lTable NATURAL JOIN rTable
  - (quasi SELECT * FROM leftTable WHERE EXISTS (SELECT * FROM rightTable WHERE leftTable.gemAttribut = rightTable.gemAttribut)

- $\rho_{RelationsnameNeu}(R_1)$, $\rho_{AttrNameNeu \leftarrow AttrNameAlt}(R_1)$, Umbenennung (von Relationen und Attributen)
  - benötigt wenn/bei
    - Unteranfrage (siehe geschachtelte anfragen)
    - Wenn nicht eindeutig aus welcher Tabelle Attribut ist, den Tabellennamen dazuschreiben. Wenn der Tabellenname nicht eindeutig ist (zwei mal die geliche Tabelle) oder er nicht Existiert (Tabelle, die Ergebinis einer Relation ist) ist Umbennennung erforderlich.
    - Schemagleichheit benötigt ist und die Attribute verschieden heißten (gelesenVon, PersNr)
  - bsp: Ermittlung indirekter Vorgänger 2. Stufe der Vorlesung 5216
    - $\prod_{v1.Vorgänger}(\sigma_{v2.Nachfolger=5216 \wedge v1.Nachfolger = v2.Vorgänger}(\rho_{v1}$ (voraussetzen) x ρ_{v2} (voraussetzen)))

- \ bzw. - Differenzmenge (EXCEPT)
  - Schemagleichheit erforderlich

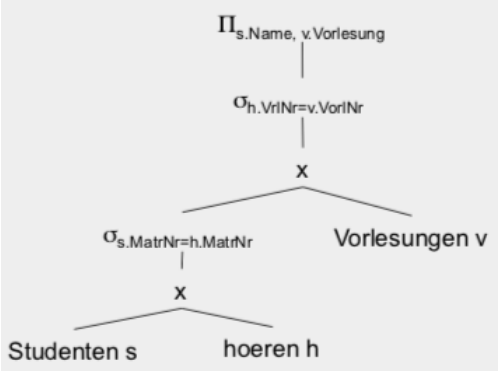
- R - S = D, bzw. R \ S = D entähllt alle Tupel aus A, die nicht in B enthalten sind
- $\prod_{MatrNr}$(Studenten) - $\prod_{MatrNr}$ (pruefen)
  (SELECT matrNr FROM Studenten) EXCEPT (SELECT matrNr FROM Pruefen)
- Nicht in allen DBS implementiert (in PostgreSQL ja, in MySQL nicht)

- ÷ Division
- ∪ Vereinigung (UNION)
  - Schemagleichheit erforderlich (gleiche Attributnamen und Attributtypen (Domönen)
  - R ∪ S = Relation, die die gleichen Attribute hat wie die Schemagleichen vereinigten Relationen R und S und alle Elemente aus R und S enthällt
  - Bsp: Berechnen aller Unimitglieder



  $\prod_{Name}$(Assistenten) ∪ $\prod_{Name}$(Professoren) ∪ $\prod_{Name}$(Studenten)
  (SELECT name FROM Assistenten) UNION (SELECT name FROM Professoren) UNION (SELECT name FROM Studenten)
  - UNION eliminiert Duplikate, UNION ALL nicht

- ∩ Schnittmenge/Mengendurchschnitt (INTERSECT)
  - Schemagleichheit erforderlich
  - $R_1 \cap R_2$ = S enthällt alle Elemente, die sowohl in R, als auch in S sind
  - $\prod_{Attribute}$ (Tabelle) ∩ $\prod_{Attribute}$(Tabelle)
    (SELECT … FROM …) INTERSECT (SELECT … FROM ….)
  - Nicht in allen DBS implementiert (in PostgreSQL ja, in MySQL nicht)
  - bsp: Personalnummern der Professoren, die eine Vorlesung halten und Rang C4 haben
    - $\prod_{PersNr}$ ($\rho_{PersNr<-gelesenVon}$(Vorlesungen)) ∩ $\prod_{PersNr}$($\sigma_{Rang=C4}$(Professoren)))
    - (SELECT gelesenVon as PersNr FROM Vorlesungen)
      INTERSECT
      (SELECT PersNr FROM Professoren WHERE rang='C4')
- Bsp: Welcher Professor ließt Ethik?
  - $\prod_{p.Name}$($\sigma_{p.PerNr=v.gelesenVon \wedge v.Titel = Ethik}$(Professoren p x Vorlesungen v))
  - SELECT name FROM professoren, vorlesungen WHERE persNr=gelesenVon AND titel='Ethik'
- Bsp: Welche Studenten hören welche Vorlesungen



  - Inline: $\prod_{s.Name, v.Vorlesung}$($\sigma_{s.MatrNr=h.MatrNr}$($\sigma_{h.VorlNr = v.VorlNr}$ (Studenten s x hoeren h) x Vorlesungen v)

- **Qualität von Relationenschemata**
  - Schlechte Relationenschemata
    - Update-Anomalien
    - Einfüge-Anomalien
    - Löschanomalien
  - Korrektheitskriterien für die Zerlegung (Dekomposition) von Relationenschemata
    - Verlustlosigkeit
      - Die ursprünglichen Relationenausprägungen des Schemas R enthaltenen Informationen müssen aus den Ausprägungen der neuen Relationenschemata R_1, R_2, …, R_n rekonstruierbar sein.
        - Auch wenn die rekonstruierte Tabelle mehr tupel entällt wird der Zustand als Verlust bezeichnet.
      - Kriterien für Verlüstlosigkeit
        - $R = R_1 \cup R_2 \cup \ldots \cup R_n$
        - Für jede mögliche (gültige) Ausprägung der Relation R gilt
          - gültigeAusprägung(R) = R_1 JOIN R_2 JOIN … JOIN R_n
        - Hinreichende Bedingung:
          - $(R_1 \cap R_2) \rightarrow R_1$ oder $(R_1 \cap R_2) \rightarrow R_2$

    - Abhängigkeitserhaltung
      - Die für R geltenden funktionalen Abhängigkeiten müssen auf die Schema R_1, R_2, …, R_n übertragbar sein.

  - Normalformen (als Gütekriterium)
    - Funktionale Abhängikeit
      Seien $a \subseteq R$, $b \subseteq R$
      $a \rightarrow b$ genau dann wenn $\forall$ r, s $\in$ R mit r.a = s.a $\Rightarrow$ r.b = s.b
    - 1. Normalform
      - Def: Jedes Attribut der Relation muss einen atomaren Wertebereich haben (d.h. zusammengesetzte, megenwertige oder geschachtelte Wertebereiche (relationenwertige Attributwertebereiche) sind nicht erlaubt)
      - Nutzen: Macht abfragen effizienter / möglich (bsp. nach Nachname sortieren, wenn Vor und Nachname in einem Attribut Name ist nicht möglich)
      - Vorgehen: Aufspalten der Relation
        - Nicht atomare Attribute aufspalten und Teil davon dem Primärschlüssel hinzufügen
        - bsp: Rezeptverwaltung R: {[ID,Rezeptname, Kategorie, Zutaten]}
          R_B = {(1, Goulasch, Fleischgericht, {1. Rindfleich 500g, 2. Schweinefleisch 500g, …})
          —> R: {[ID,Rezeptname, Kategorie, ZutatNr, Zutat, Anz, Einheit]}
    - 2. Normalform
      - Def: Relation R mit zugehörigen Funktionalen Abhängigkeiten ist in zweiter Normalform, wenn:
        - in 1. Normalform
        - jedes Nichtschlüssel-Attribut voll funktional abhängig ist von jedem Kandidatenschlüssel der Relation (bei uns zur vereinfachung: … vom Primärschlüssel)
          - Bzw. Jedes nicht-primäre Atribut ist jeweils von gesamten Primärschlüssel (bzw. eig. …) abhängig, nicht nur von Teil
      - Nutzen: Jede Relation modelliert nur einen Sachverhalt (“monothematische” Relationen).
        —> Nur noch logisch/sachlich zusammengehörige Information in einer Relation
        —> Redundanz und dadurch auch Gefahr von Inkonsistenzen reduziert
      - Vorgehen:
        - Funktionale Abhängigkeiten (FDs) ermitteln (die, bei denen auf rechter Seite ein Nicht-Schlüsselattribut steht für II. NF relevant)
        - Verbotene FDs auslagern indem linke Seite Schlüssel der neuen Relation ist und Rechte Seite nicht-Schlüssel
        - bsp: R: {[ID,Rezeptname, Kategorie, ZutatNr, Zutat, Anz, Einheit]}
          —> R: {[ID, Rezeptname, Kategorie]}, S: {[ID, ZutatNr, Zutat, Anz, Einheit]}
        - alternativ: ist die Relation in 1. NF und besteht der Primärschlüssel aus nur einem Attribut, so liegt automatisch die 2. NF vor. Man kann also einfach neue ID einführen

- 3. Normalform
  - Def: Relation R mit zugehörigen FDs ist in 2. NF, wenn:
    - in 2. Normalform
    - jedes Nichtschlüssel-Attribut von keinem Schlüsselkandidaten P (bei uns vereinfacht: von Primärschlüssel) transitiv abhängt (also P —> A, A —> B, damit P —> B)
      - bzw. Ein Nichtschlüsselattribut darf nicht von einer Menge abhängig sein, die ausschließlich aus Nichtschlüsselattributen besteht (Darf also nur von einem Kandidatenschlüssel (/dem Primärschlüssel) abhängen.
      - impliziert: Ist die Relation in 2. NF und besitz außer Primärschlüsel höchstens ein weiteres Attribut, so liegt die Tabelle in 3. Nf vor
  - Nutzen: Relationen des Schemas sind zuverlässig monothematisch (verbliebene Thematische Durchmischungen in Relation werden behoben)
  - Vorgehen: Verbotene FDs auslagen, indem linke Seite Schlüssel der neuen Relation ist und Rechte Seite nicht-Schlüssel
    - bsp: R: {[ID, Rezeptname, Kategorie]}, S: {[ID, ZutatNr, Zutat, Anz, Einheit]} —> R: {[ID, Rezeptname]}, S: {[ID, ZutatNr, Zutat, Anz, Einheit]}, T: {[Rezept, Kategorie]}

Verschiedene Anfragesprachen

- es gibt 3 mathematisch orientierte ANfragesprachen:
  - Alle 3 gelich mächtig aber nicht Turing-vollständig (es gibt Anfragen, die sich nicht mit ihnen ausdrücken lassen, z.b. beliebig viele Vorgänger-Beziehungen der Vorlesungen)
  - 1. Relationale Algebra
    - prozedurale Berechnungsvorschrift, vgl. Operatorbäume)
  - 2. Relationaler Tupelkalkül
    - Relationenkalkül (deklarativ), basierend auf dem mathematischen Prädikatenkalkül erster Stude, d.h. mit den Quantoren: Existenzquantor, Allquantor
    - Form: {t | P(t)} mit
      t: Tupelvariable
      P: Prädikat, dass erfüllt sein muss, damit t ins Ergebnis übernommen wird
    - Bsp: Berechne alle C4-Professoren:
      {p | p ∈ Professoren ∧ p.Rang = ‘C4}
      SELECT * FROM Professoren WHERE Rang=‘C4’
    - Bsp: Studenten, die min eine Vorlesung hören:
      {s | s ∈ Studenten ∧ ∃h ∈ hoeren(s.MatrNr=h.MatrNr) }
      SELECT * FROM Studenten s WHERE EXISTS (SELECT h.* FROM hoeren h WHERE s.MatrNr = h.MatrNr);
    - Bsp: Welche Studenten haben noch keine Vorlesung gehört?
      {s | s ∈ Studenten ∧ ¬∃h ∈ hoeren(s.MatrNr=h.MatrNr)}
      SELECT * FROM Studenten WHERE NOT EXISTS (SELECT * FROM hoeren WHERE s.MatrNr=h.MatrNr)
  - 3. Relationaler Domänenkalkül
    - Relationenkalkül (deklarativ), basierend auf dem mathematischen Prädikatenkalkül erster Stude, d.h. mit den Quantoren: Existenzquantor, Allquantor

Sicherheit (und Sichten)

- Sicherheitskonzepte in Datenbanken
  - Discretionary Access Control
  - Rechte und Rollen
  - Weitergabe von Rechten
- Schutzmechanismen
  - Identifikaiton und Authentisierung des Nutzers
    - durch Eingabe von Username und Passwort oder Smartcard
  - Autorisierung und Zugriffskontrolle

    - Menge von Regeln, die die erlaubten Arten des Zugriffs auf Sicherheitsobjekte (z.B. Tabellen, Attribute, Tupel) von Sicherheitssubjekten (z.b. Benutzer, Prozesse) definieren
  - Auditing
    - Zur Überprüfung von Richtigkeit und Vollständigkeit von Autorisierungsregeln wird über jede sicherheitsrelevante Operation Buch geführt

- Beispiel: Rechtevergabe Unix/Linux
  - Zugrifsrechte einzelnen Nutzern oder Grupen zugeordnet
    - Nutzer enthält Rechte über direkte Zuweisung oder über Gruppenzugehörigkeit
  - Definition von Gruppen (Sammlung von Nutzern)
  - Jede Datei enthält Rechteinformationen (rwx) über Besitzer, gruppenzugehörigkeit (genau eine), alle sonstigen nutzer
- Ebenen von Zugriffsrechten
  - können direkt vergeben werden
    - Zugriffsrechte auf DBMS-Ebene (Wer darf von wo an das DBMS verbinden)
    - Zugriffsrechte auf Dtenbankebene (wer darf welche Datenbank wie nutzen)
    - Zugriffsrechte auf Tabellenebene (Häufig in Verknüpfung mit einer Sicht)
    - Zugriffsrechte auf Attributebene
  - Werden mit Hilfe von SIchten modelliert
    - Zugriffsrechte innerhalb eines Attributs (abhängig z.b. von Werten)
- Explizite Autorisierung
  - Nimmt schnell großen Umfang an
  - erlaubt Zugriff auf Objekt
  - Superuser darf im System alles (u.a. als einziger den Besitzer einer Tabelle ändern)
  - Tabellen haben jew. “Besitzer” (ist Rolle/Nutzer)
  - Besitzer darf lesen, schreiben, ändern des Schemas (add column, drop column, rename,…)
  - Niemand außer Besitzer darf neu angelegte Tabelle lesen, schreiben oder löschen
  - Rechte zum Zugriff auf einzelne Objekte müssen erst weitergegeben werden
- Dirscretionary Access Control (DAC)
  - Zugriffsregeln der DAC geben zu Subjekt s mögliche Zugriffsarten auf Objekt o an. Formal als Quintupel: (o,s,t,p,f)
    - $o \in O$, Menge der Objekte (z.B. Datenbanken, Relationen, Attribute),
    - $s \in S$, Menge der Subjekte (z.B. Benutzer, Prozesse),
    - $t \in T$, Menge der Zugriffsrechte (z.B. T = {lesen, schreiben, löschen})
    - $p \in P$, Prädikat (z.B. Rang = ‘C4’ für die Relation Professoren)
    - $f \in F$, Boolescher Wert, der Angibt, ob s das Recht (o,t,p) an ein anderes Subjekt s’ weitergeben darf
  - Vorstellbar als Tabelle

	Nutzer 1	Nutzer 2	....
Tabelle 1	S U D G	-	…
Tabelle 2	S U D G	-	…
Attribut x	-	S	…
…			

  - Umsetzung von SQL
    - Subjekte sind sog. Rollen oder Nutzer
    - Im Prinzip sind Nutzer gleich Rollen (Nutzer können sich einloggen)
    - Anlegen eines Nutzers
      CREATE USER <username>;
      CREATE USER <username> PASSWORD ‘<password>’;
    - Löschen eines Nutzers
      DROP USER <username>;
    - Anlegen einer Rolle
      CREATE ROLE <rollenname>
    - Löschen einer Rolle
      DROP ROLE <rollenname>
    - Rollen ansehen
      SELECT rolname FROM pg_roles;
      oder: \du+
    - Weitergabe von Rechten
      GRANT SELECT | INSERT | UPDATE | DELETE | RULE | REFERENCES | TRIGGER [,…] |

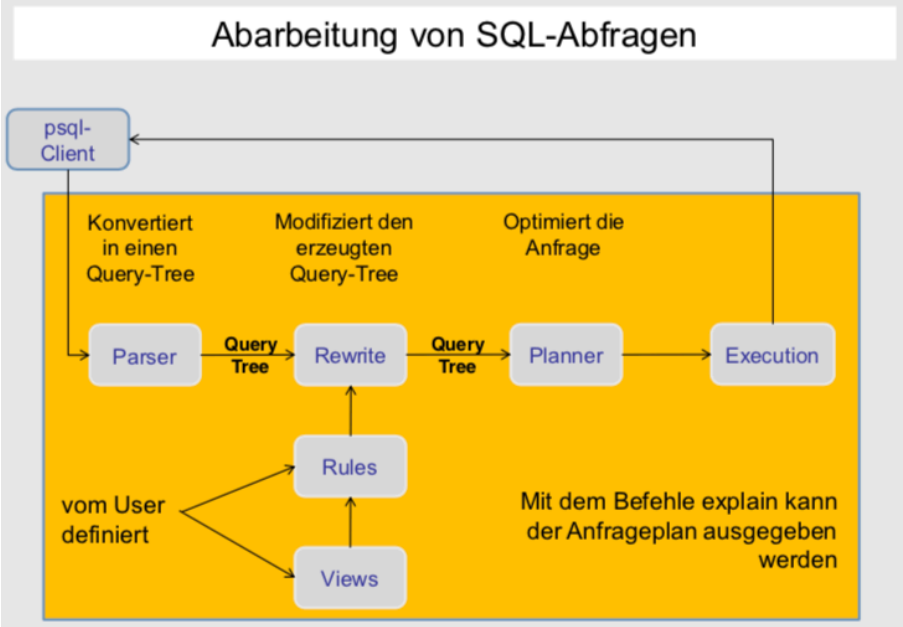
{ALL [PRIVILEGES] }
ON [TABLE] tablename [,…] TO {username | GROUP groupname | PUBLIC} [,…]
[WITH GRANT OPTION];
- Bsp.
  - GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO s0563168;
  - GRANT SELECT, INSERT ON Studenten TO gaertner WITH GRANT OPTION;
  - GRANT SELECT (MatrNr) ON Studenten to AllStudents;
- Rechte entziehen
  - S1 gibt Recht an S2 und S3. S2 gibt Recht an S3 weiter. S2 will Recht von S3 entziehen —> Fehler
  - Kaskadierend (rekursiv)
    - REVOKE SELECT, INSERT FROM gaertner CASCADE
    - Aufhebung der ursprünglichen Zuweisung und allen davon abhängigen Zuweisungen.
      - Bsp. S1 gibt Recht an S2, S2 gibt es weiter an S3. S1 entzieht S2 das recht. Es wird aufgrund von CASCADE auch von S3 entzogen
    - Schwierigkeit bei Zyklen
  - Nicht kaskadierend: RESTRICT
    - Falls das Recht weitergegeben wurde ist löschung nicht möglich
    - REVOKE SELECT, INSERT FROM gaertner RESTRICT
- Es wird dabei gespeichert von wem weitergegeben

- Knüpfung von Rechten an Rollen
  - Rolle (ähnlich Gruppe) erzeugen
    CREATE ROLE Prüfungsausschuss
  - Nutzer eine Rolle hinzufügen
    GRANT <rolle> TO <nutzer>
  - Vergabe des Rechts zum Verbinden zur Datenbank
    GRANZ CONNECZ ON DATABASE test TO Prüfungsausschuss
  - Vergabe der Berechtigung alle Einträge in pruefen zu sehen an Prüfungsausschuss
    GRANT SELECT, INSERT, UPDATE, DELETE ON pruefen TO Prüfungsausschuss
  - Vergabe des Rechts in der Tabelle Professoren PersNr, Name lesen zu dürfen
    GRANT SELECT(PersNr, Name) ON Professoren TO Prüfungsausschuss
    - Ist der Nutzer also mit Rolle Prüfungsausschuss authentiiziert, so erhällt er bei SELECT * FROM Professoren; die Spalten PersNr, Name
  - Weitergabe des Rechts die Tabelle Vorlesungen zu lesen an alle Nutzer
    GRANT SELECT ON Vorlesungen TO public
  - Rechte ansehen
    - Alle Rechte des aktuellen Nutzers an Tabelle pruefen anzeigen
      SELECT * FROM information_schema.table_privileges WHERE table_name='pruefen';

- Verknüpfung mit Sichten
  - häufig sinnvoll
  - GRANT SELECT ON sicht TO subjekt
- **Sicht**: Tabelle, die Daten aus einer/verschiedenen Tabelle/n für einen speziellen Zweck zusammenfasst.
  - Erzeugen
    CREATE [OR REPLACE] [TEMP | EMPORARY] VIEW name [(column_name [,…]]] AS query
  - Löschen
    DROP VIEW [IF EXISTS] name [, …] [CASCADE | RESTRICT]
  - Dient Datenschutz, Vereinfachung von Zusammenhängen, Aufbau statistischer Auswertung
  - Einmalig definiert, stehen dann während der Laufzeit zur (wie tabelle) Verfügung (im Gegensatz zu WITH-statement)
    - wird auch von \d angezeit
  - Intern definition der Sicht gespeichert, nicht als Tabelle.
  - Beispiel: note aus Prüfen verbergen
    CREATE VIEW pruefenSicht AS
      SELECT MatrNr, VorlNr, PersNr FROM pruefen

- Beispiel: Sicht, die beinhaltet, welche Studenten bei welchen Professoren welche VOrlesungen hören
  CREATE VIEW StudProfVl (Sname, Semester, Titel, Pname) AS
  SELECT s.name, s.Semester, v.Titel, p.Name
  FROM Studenten s, hören h, Vorlesungen v, Professoren p
  WHERE s.MatrNr = h.MatrNr AND h.VorlNr = v.VorlNr AND p.PersNr = v.gelesenVon
  - Für Laien dann einfach abragbar
  SELECT * FROM StudProfvl WHERE Pname='Gaertner';
- Änderbarkeit der Daten von Sichten
  - in SQL möglich, wenn
    - nur eine Basisrelation verwendet wird
    - der Schlüssel vorhanden ist
    - keine Aggregatfuntionen, Gruppierung und Duplikateliminierung benutzt wird
    - —> also fast nie
  - in PostgreSQL lässt sich mit Regel verändern unter welchen Bedinungen die Daten Sichten verändert werden können
    - Bsp: CREATE VIEW prof(PersNr, Name, Raum) AS SELECT PersNr, Name, Raum FROM Professoren;
      CREATE RULE "_insertProf" AS ON INSERT TO prof DO INSTEAD
      insert into Professooren(PersNr, Name, Raum) VALUES (new.PersNr, new.Name, new.Raum);

-



Transaktionen

- Def: Eine Folge von Datenverarbeitungsbefehlen (lesen, verändern, einfügen, löschen), die eine Datenbank von einem konsistenten ZUstand in einen anderen (oder den selben) konsistenten Zustand überführt. Dabei ist wesentlich, dass die Folge von Befehlen (logisch) ununterbrechbar, d.h. atomar ausgeführt wird
- Eigenschaften (entsprechen ACID-Paradigma, einem grundlegendem Konzept verteilter Systeme):
  - Atomacy (Atomarität):
    - Eine Transaktion ist die kleinste, nicht mehr weiter zerlegbare Einheit. Entweder werden alle Änderungen der Transaktion in die Datenbank geschrieben oder keine
  - Consistency (Konsistenz):
    - Eine Transaktion hinterlässt nach Beendigung einen konsistenten Datenbasiszustand. Andernfalls iwrd sie zurückgesetzt.
      - Zwischenzustände können inkonsistent sein
        - Hinzufügen von Datensätzen, deren Fremdschlüssel sich auf einen Datensatz bezieht, der erst ab dem nächsten Befehl existiert (führt normalerweise zu fehler) ist also in einer Transaktion möglich.

        - Dafür ist es notwendig bei Tabellendefinition den Fremdschlüssel als DEFERRABLE INITIALLY DEFERED (also auffschiebbar) zu markieren.
      - Bsp:
        CREATE TABLE Bestellungen(
          …
          Ausleiher INTEGER REFERENCES Mitarbeiter(MitarbeiterID) ON DELETE CASCADE DEFERRABLE INITIALLY DEFERRED,
          …
        );
        - Konsistenzbedingung 'Referenz' wird erst am Ende einer Transaktion überprüft
  - Isolation (Isolation):
    - Nebenläufig (parallel, gleichzeitig) ausgeführte Transaktionen dürfen sich nicht gegenseitig beeinflussen. Jede Transaktion muss - logisch gesehen - so ausgeführt werden, als ob sie die einzige aktive auf der Datenbank wäre.
  - Durability (Dauerhaftigkeit):
    - Nach Beenden der Transaktion (commit) werden die Daten dauerhaft in die Datenbank geschrieben.

- Zweistufige Speicherhierarchie:
  DBMS-Puffer <—Einlagerung—Auslagerung—> Hintergrundspeicher

- Anforderungen (für transaktionssichere Datenbank sind die folgenden beiden grundlegenden Mechanismen umgesetzt werden:
  - Recovery: Behebung von eingetretenen (oft unvermeidbaren) Fehlersituation
    - Datenbank enthält Mechanismen zur Behandlung der Fehler: Log-Files, Spiegelung der DB, Regelmäßiger Backup, …
    - Fehlerklassifikaiton
      - 1. Lokaler Fehler in noch nicht festgeschrebener (committed) Transaktion
        (Führen zu scheitern der Transaktion, beeinflussen aber den Rest des Systems hinslichtlich Datenkonsistenz nicht)
        - Wirkung muss zurükgesetzt werden (lokales Undo, Muss in DBMS effizient implementiert sein, da häufig auftritt)
        - R1-Recovery
        - Bsp:
          - Anwendungsfehler
          - Expliziter Abbruch der Transaktion (abort) durch Benutzer
          - Systemgesteuerter Abbruch einer Transaktion (z.b. um Verklemmung (Deadlock) aufzulösen
      - 2. Fehler mit Hauptspeicherverlust
        - Abgeschlossene Transaktionen müssen erhalten bleiben (R2-Recovery) (globales UNDO)
        - Noch nicht abgeschlossene Transaktionen müssen zurückgesetzt werden (R3-Recovery)
        - —> Änderungenn müssen protokolliert werden.
      - 3. Fehler mit Hintergrundspeicherverlust
        - R4-Recovery
        - Bsp:
          - "head crash" der Festplatte
          - Externe Faktoren (Feuer, Erdbeben,…)
          - Fehler in Systemprogrammen (z.B. in einem Plattentreiber)
        - —> Durchführen von Archivkopien der materialiserten Datenbasis und Erzeugen eines Logarchivs mit allen vollzogenen Änderungen seit der letzten Archivkopie.
  - Synchronisation von mehreren gleichzeitig auf der DB ablaufenden Transaktionen

- "Definieren" einer Transaktion in klassischen DB-Systemen
  - begin of transaction: Markiert den Begin einer Transaktion
    - PostgreSQL: START TRANSACTION (alternativ BEGIN, aber von weniger DBS unterstützt)
  - commit: Leitet die Beendigung der Transaktion ein. Alle änderungen der Datenbasis werden durch diesen Befehl festgeschrieben
    - PostgreSQL: COMMIT (alternativ END, aber von weniger DBS unterstützt)
  - abort: Leitet Selbstabbruch der Transaktion ein. DBS stellt sicher, dass die Datenbasis wieder in den Zustand zurück versetzt wird, der vor Beginn der Transaktion existierte
    - in PostgreSQL: ABORT oder ROLLBACK

    - Bei Fehlern gilt COMMIT=END=ABORT=ROLLBACK
- Abbruch von Transaktionen: Commit, Abort, Fehler.
- Fehlerquellen
    - Stromausfahll, technischer Defekt (exogene Faktoren)
    - Fehler im Datenbanksystem
    - Fehler im Programmcode (SQL-Befehle)
    - Verletzung von INtegritätsbedingungen
    - Deadlocks
- Autocommit
    - Fast alle Transaktionen (außer z.b. CREATE TABLE) finden innerhalb einer Transaktion statt
    - Vermeindlich kleine änderungen können größere Änderungen durch das Auslösen von Triggern nach sich ziehen (z.b. CASCADE). Die Definition einer Transaktionn erleichtert das Zurücksetzen
    - sollte z.b. bei großen Datenimporten ausgeschaltet werden
        - bei Definition einer Transaktion automatisch und temporär - genau für diese Transaktion
        - Ausschalten des autocommits
          \set AUTOCOMMIT OFF (großschreibung relevant)

- **Mehrbenutzerbetrieb**:
    - Fehler:
        - Lost Update: Verloren gegangene Änderungen.
            - bsp: A überweist B geld. Während der neue Kontostand von A berechnet wird auf Basis des altne werts, schreibt eine andere Transaktion A eine Zinsgutschrift von 3% zu (ändert Kontostand). Dann schreibt Transaktion 1 den berechnetetn Kontostand in A, die 3% sind verloren.
        - Dirty Read: Wert wird gelesen, der so niemals in einem gültigen (transaktionskonsistenten) Zustand vorkommt.
            - bsp: Transaktion 1 schreibt wert in A, bricht dann später aber. Vor abbruch ließt Transaktion 2 den Wert aus A (und aktualisiert diesen basierend au dem gelesenen.)
        - Phantomproblem: Transaktionn bekommt für die selbe Anfrage unterschiedliche Antworten, da andere Transaktion zwischendurch den wert ändert.
            - bsp: Summe aller Kontostände 2 mal Kontostand berechnen, zwischendurch Konto mit kontostand != 0 hinzufügen.
        - **Serialisierbarkeit**
            - Grund: Zielkonflikt: Parallele Ausführung erhöht Leistungsfähigkeit des Systems signifikant. Transaktionen können sich gegenseitig beeinflussen, was zu fehlern führen kann
            - Die serialisierbare Ausführung einer Menge von transaktionen entspricht einer kontrolllierten, nebenläufigen, verzahnten Ausführung, wobei die Kontrollkomponente dafür sorgt, dass die (beobachtbare) Wirkung der nebenläufigen Ausführung einer möglichen seriellen Ausführung der Transaktion entspricht.
                - alt: Als Serialisierbar bezeichnet man eine Historie, wenn sie zum selben Ergebnis führt wie eine nacheinander (seriell) ausgeführte Historie über die selben Transaktionen.
            - #TODO Beispiele zum scheitern der Serialisierbarkeit Folie 32-36. (teils siehe Transaktionsverwaltung)
            - Transaktionsverwaltung in SQL92
                - SET TRANSACTION
                  [read only, | read write]
                  [ISOLATION LEVEL
                  read uncommitted |
                  read committed, |
                  repeatable read, |
                  seriazable]
                - read uncomitted: Die schwächste Konsistenzstufe. Dafar nur für read only-Transaktionen spezifiziert werden. Diese haben dann Zugiff auf noch nicht festgeschriebene Daten.
                    - bsp: T_1 macht read(A)…write(A), dann T_2 read(A)…, dann T_1 rollback
                - read committed: Transaktionen lesen nur festgeschriebene Werte. Allerdings können sie unterschiedliche Zustände der Datenbasis-Objekte zu sehen bekommen.

    - bsp: T_1 macht read(A), dann T_2 write(A), commit, dann T_1 read(A)
  - repeatable read: readcommitted, jedoch wird das problem des non repeatable read durch diese Konsistenzstufe ausgeschlossen. Es kann jedoch noch zu einem Phantomproblem kommen.
    - bsp: Änderungsaktion führt dazu, dass Tupel ein Selektionsprädikat erfüllen, das sie zuvor nicht erfüllten.
  - serializable (default): Die Konsistenzstufe fordert die Serialisierbarkeit.

- JDBC: Methoden der Klasse Connection mit bezug auf Transaktionen
  - void setAutocommit(boolean)
  - boolean getAutocommit()
  - void commit()
  - void rollback()
  - coid setTransactionIsolation(int)
  - int getTransactionIsolation()
  - Safepoint setSavepoint()
  - void rollback(Savepoint)
  - …

JDBC siehe Folien

#TODO Übungsaufgaben abstrahieren in sprache Entitätstypen/entitäten in beziehung mit,…
- und dafür dann lösung ebenfalls abstrahiert
- jew. beispiel mitaufschreiben
—> in klausur kann abgeschrieben werden