DB II B: DBMS-Implementierung

Übung 3: Systemkatalog, Zugriffsrechte

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2021/22

http://www.informatik.uni-halle.de/~brass/dbi21/

Inhalt

Blatt 1, Aufgabe a) (1)

Data Dictionary Abfrage: Tabellen mit gegebener Spalte

- Schreiben Sie eine SQL-Anfrage, die alle eigenen Tabellen ausdruckt, die eine Spalte „DEPTNO“ enthalten.
- Lösung:

```
SELECT table_name
FROM   user_tab_columns
WHERE  column_name = 'DEPTNO'
```

- Statt `user_tab_columns` kann man auch `cols` schreiben.

  Die Groß-/Kleinschreibung von Tabellen und Spaltennamen ist egal, außer, wenn sie als Zeichenkette auftreten (wie `DEPTNO` hier). Oracle wandelt alle Tabellen- und Spaltennamen (außer Delimited Identifier `"..."`) intern in Großbuchstaben um. PostgreSQL wandelt sie dagegen in Kleinbuchstaben um. Beide Varianten erreichen das Ziel, dass die Bezeichner nicht case-sensitiv sind.

Blatt 1, Aufgabe a) (2)

- Duplikatelimination ist hier überflüssig:

```
SELECT DISTINCT TABLE_NAME
FROM   COLS
WHERE  COLUMN_NAME = 'DEPTNO'
```

Die Kombination von `TABLE_NAME` und `COLUMN_NAME` ist effektiv ein Schlüssel von `COLS`. Da `COLS` eine Sicht ist, ist der Schlüssel nirgendwo deklariert. Aber eine Tabelle kann eben nur eine Spalte mit einem bestimmten Namen haben. Wenn man mit den Ausgabespalten und anderen eindeutig bestimmten Spalten einen Schlüssel von jeder Tupelvariable hat, kann es keine Duplikate geben (mehr Details in „Einführung in Datenbanken“).

Blatt 1, Aufgabe a) (3)

- LIKE und * entsprechen nicht der Aufgabenstellung:

```
SELECT *
FROM   user_tab_columns
WHERE  column_name LIKE '%DEPTNO%'
```

Achten Sie auf die Details der Aufgabenstellung, halten Sie die Spezifikation genau ein. Die Tabelle hat auch ziemlich viele Spalten, so dass `SELECT *` wirklich nicht schön ist.

- Bitte alles, was nicht SQL ist, als Kommentar markieren:

```
a)      -- Richtig waere so: -- a)
SELECT TABLE_NAME
FROM   COLS
WHERE  COLUMN_NAME = 'DEPTNO'
```

Blatt 1, Aufgabe a) (4)

- Dies ist ziemlich kompliziert:

```
SELECT table_name FROM cat
WHERE  LOWER(table_type) = LOWER('TABLE')
AND    EXISTS(SELECT * FROM cols
              WHERE  cols.table_name
                     = cat.table_name
              AND    LOWER(cols.column_name)
                     = LOWER('DEPTNO'))
```

Die Unteranfrage ist überflüssig, ein Join hätte es auch getan. Bei `TABLE_TYPE` kann man sicher sein, dass Oracle nur die Variante mit Großbuchstaben verwendet. Beim Spalten-Name wurde genau nach `DEPTNO` gefragt.
Es ist formal korrekt, dass in der Aufgabe „Tabelle“ stand. Wenn ich nicht ausdrücklich „Basistabelle“ sage, meine ich vermutlich immer „Tabelle oder Sicht oder Synonym . . . “.

Blatt 1, Aufgabe a) (5)

- Dies liefert auch fremde Tabellen:

```
SELECT TABLE_NAME
FROM   ALL_TAB_COLUMNS
WHERE  COLUMN_NAME = 'DEPTNO'
```

- Aufgabenstellung: „Schreiben Sie eine SQL-Anfrage, die alle eigenen Tabellen ausdruckt, die eine Spalte „DEPTNO“ enthalten.
- Wenn man wirklich auch Tabellen anderer Nutzer will, sollte man auch die Spalte OWNER mit ausdrucken.

  Man interessiert sich ja eigentlich nicht für den Tabellen-Namen, sondern die Tabelle. Ohne OWNER würde die Anfrage auch Duplikate liefern.

Blatt 1, Aufgabe a) (6)

- Warum so kompliziert?

```
SELECT table_name
FROM   all_tab_columns
WHERE  column_name LIKE 'DEPTNO'
AND    OWNER IN ('SCOTT')
```

- Ich ziehe einen halben Punkt ab, wenn `LIKE` ohne Wildcards (`%` oder `_`) verwendet wird. Schreiben Sie „`=`“.

  Nur, wenn Sie unbedingt die „Non-Padded“ Vergleichs-Semantik brauchen, wäre `LIKE` ohne echtes Muster zu verwenden.

- Statt dem `IN` wäre auch `=` möglich.

  Vielleicht sollte man `USER` statt des festen Nutzernamens verwenden. Aber wenn man `user_tab_columns` nutzt, ist das schon eingebaut.

Blatt 1, Aufgabe a) (7)

- Es funktioniert auch:

```
SELECT TABLE_NAME
FROM   USER_TAB_COLS
WHERE  COLUMN_NAME = 'DEPTNO'
```

  Zuerst habe ich in `all_synonyms` geschaut, ob beide Namen vielleicht auf die gleiche Sicht verweisen. Aber es gibt `SYS.USER_TAB_COLS` und `SYS.USER_TAB_COLUMNS`. Man kann aus `DICT_COLUMNS` herausfinden, dass die beiden Tabellen unterschiedlich viele Spalten haben (wie?), `USER_TAB_COLUMNS` hat 36, `USER_TAB_COL` hat 43. Können Sie die Differenz-Spalten bestimmen?

- Schauen Sie in die Oracle Database Reference, Kapitel 3, „Static Data Dictionary Views“ für diese beiden „Tabellen“.

  [https://docs.oracle.com/en/database/oracle/oracle-database/18/administration.html]

Blatt 1, Aufgabe b) (1)

Data Dictionary Abfrage: Anzahl Spalten

- Schreiben Sie eine SQL-Anfrage, die für alle eigenen Tabellen Tabellenname und Anzahl Spalten ausdruckt. Sortieren Sie das Ergebnis nach der Anzahl Spalten, größte zuerst.
- Lösung:

```
SELECT table_name, count(*) AS num_columns
FROM   user_tab_columns
GROUP  BY table_name
ORDER  BY num_columns DESC
```

- Ein Student hatte `COUNT(column_name)`.

  Nicht falsch, aber unnötig. Was sind die Unterschiede? Was wird suggeriert, was nicht zutrifft?

Blatt 1, Aufgabe b) (2)

- Auch möglich (Term unter `ORDER BY`):

```
SELECT TABLE_NAME, COUNT(COLUMN_NAME)
FROM   COLS
GROUP  BY TABLE_NAME
ORDER  BY COUNT(COLUMN_NAME) DESC
```

- Man darf Spalten im `ORDER BY` auch über die Position ansprechen, das gilt aber als veraltet:

```
SELECT TABLE_NAME, COUNT(*)
FROM   COLS
GROUP  BY TABLE_NAME
ORDER  BY 1 DESC
```

Blatt 1, Aufgabe b) (3)

- Ich wusste gar nicht, dass es eine Tabelle COL gibt:

```
SELECT TNAME AS "Tabelle",
       COUNT(TNAME) AS "Spalten"
FROM   COL
GROUP  BY TNAME
ORDER  BY COUNT(TNAME) DESC
```

- Das Handbuch sagt: „COL is included for compatibility. Oracle recommends that you not use this view.“
- Wo haben Sie das her? Halten Sie sich an die in der Vorlesung genannten Tabellen.

  Oder informieren Sie sich gründlich, und klären Sie mich dann auch auf über eventuelle neue Features.

Blatt 1, Aufgabe b) (4)

- Wenn man unbedingt sicherstellen möchte, dass es eine echte Tabelle ist (und keine Sicht):

```
SELECT table_name, COUNT(*) AS col_count
FROM   user_tables JOIN user_tab_columns
       USING (table_name)
GROUP  BY table_name
ORDER  BY col_count DESC
```

- Funktioniert wohl, entspricht aber nicht der Aufgabe:

```
SELECT c.table_name, max(column_id) as n
FROM   user_tab_columns c
group  by c.table_name
order  by n desc
```

Warum Tupelvariable c, wenn nur eine Tabelle? Warum manche Schlüsselworte groß, andere klein?

Blatt 1, Aufgabe b) (5)

- Gruppierung nach `OWNER` ist überflüssig:

```
SELECT TABLE_NAME, COUNT(COLUMN_NAME)
FROM   ALL_TAB_COLUMNS
WHERE  OWNER = 'SCOTT'
GROUP  BY OWNER, TABLE_NAME
ORDER  BY COUNT (COLUMN_NAME) DESC
```

Warum überhaupt `ALL_TAB_COLUMNS` verwenden statt `USER_TAB_COLUMNS`?

- `HAVING` ist hier überflüssig:

```
select table_name, count(table_name)
from   (select * from all_tab_columns
        where owner = 'SCOTT')
group  by table_name
having count(table_name) > 0
order  by count(table_name) desc
```

Blatt 1, Aufgabe c) (1)

Blatt 1, Aufgabe c)

- Schreiben Sie eine SQL-Anfrage, die alle eigenen Tabellen ausdruckt, die einen Fremdschlüssel auf die Tabelle `DEPT` enthalten.
- Lösung:

```
SELECT f.table_name
FROM   user_constraints f, user_constraints k
WHERE  f.CONSTRAINT_TYPE = 'R' -- foreign key
AND    k.table_name = 'DEPT'
AND    f.r_constraint_name = k.constraint_name
```

- Der referenzierte Constraint ist sicher ein Schlüssel, die folgende Bedingung ist daher überflüssig:

```
AND k.constraint_type IN ('P','U')
```

Blatt 1, Aufgabe c) (2)

- Dies ist falsch, es würde nach einem Fremdschlüssel-Constraint suchen, den man „DEPT“ benannt hat:

```
SELECT TABLE_NAME
FROM   USER_CONSTRAINTS
WHERE  CONSTRAINT_TYPE = 'R'
AND    CONSTRAINT_NAME = 'DEPT'
```

Oracle Dokumentation

Blatt 1, Aufgabe d)

- Schauen Sie sich die Oracle 18c Dokumentation an:

  [https://docs.oracle.com/en/database/oracle/oracle-database/18/]

  Finden Sie das Handbuch „Database Concepts“ (unter „Administration“). Beantworten Sie folgende Frage: „Wie viele Kapitel hat dieses Handbuch und welches Kapitel beschäftigt sich mit dem Data Dictionary?“

Inhalt

PostgreSQL über Adminer Webschnittstelle (1)

- Wir haben einen PostgreSQL 14.0 Server laufen.
- Man kann darauf mit der Web-Schnittstelle Adminer zugreifen:

  [https://dbs.informatik.uni-halle.de/edb?pgsql=db&username=student_gast&db=postgres&ns=]

  Man kann Daten des Login-Bildschirms in der URL vorbelegen („`ns`“ steht für „Namespace“, d.h. Schema).
- Die Eingaben im Login-Formular des Adminers sind:
  - System: PostgreSQL
  - Server: `db`
  - Username: `student_gast`
  - Password: (Wird in Übung bekannt gegeben)
  - Database: `postgres`

Datenbanken (1)

- Ein "Database Cluster" in PostgreSQL besteht aus mehreren Datenbanken, die von einer Instanz des Datenbank-Servers verwaltet wird.

  Bei anderen Datenbanksystemen wäre ein "Cluster" eher eine Gruppe von Datenbank-Servern (mehrere Rechner, um Performance und Zuverlässigkeit zu steigern). Die GUI "`pgAdmin`" zeigt "Servers" an, darunter "Databases".

- Ein "Database Cluster" entspricht einem Verzeichnis im Dateisystem, in dem alle Datenbanken gespeichert sind.

  Z.B. `/usr/local/pgsql/data`, `/var/lib/pgsql/data`,
  `C:\Program Files (x86)\PostgreSQL\9.5\data`
  Wenn man `psql` als Nutzer `postgres` startet ("`sudo -u postgres psql`"), kann das Verzeichnis mit "`show data_directory;`" angezeigt werden.
  Unter Linux findet man es auch in "`ps -ef | fgrep postgres`" als Wert der Option `-D` des ersten Postgres Prozesses ("Postmaster").

Datenbanken (2)

- Nachdem ein DB-Cluster angelegt wurde (mit `initdb` im Rahmen der Installation), werden darin drei Datenbanken angelegt: `postgres`, `template0` und `template1`.

  "`SELECT datname from pg_database;`" oder "`\l`" in `psql`.

  Die Datenbank `postgres` gibt es ganz von Anfang an, man kann sich mit ihr verbinden, um andere Datenbanken anzulegen (oder wenn man sonst keinen Datenbank-Namen kennt). Sie scheint im wesentlichen leer zu sein.

- Nutzer-Datenbanken werden normalerweise durch Clonen von `template1` angelegt.

  Man kann die Template-Datenbank aber auch explizit angeben:

  ```
  CREATE DATABASE dbname TEMPLATE template0;
  ```

  Es gibt zwei Template-Datenbanken, weil man `template1` lokal modifizieren kann (z.B. würden dort angelegte Tabellen automatisch mitkopiert), aber `template0` das Original bleiben soll. Direkt nach der Installation sind beide Template-Datenbanken gleich.

Datenbanken (3)

- Am einfachsten heißt die Datenbank wie der Betriebssystem-Nutzer.

  Wenn man die Kommandozeilen-Schnittstelle `psql` ohne explizite Angabe von Benutzer und Datenbank aufruft, wird der Benutzername des aktuellen Betriebssystem-Nutzers für beide Angaben eingesetzt.
  Wenn mein Login also `brass` ist, wäre es gut, einen Datenbank-Nutzer `brass` anzulegen, dem die Datenbank `brass` gehört.

- Wenn man PostgreSQL frisch installiert hat, gibt es natürlich noch keine Datenbank für bestimmte Nutzer.

- Man kann dann zuerst den Betriebssystem-Nutzer als Datenbank-Nutzer mit dem Programm `create_role` anlegen, anschliessend die Datenbank mit `createdb`.

  Beides geht auch direkt mit SQL-Befehlen, siehe nächste Folie.

Datenbanken (4)

- Nutzer und Datenbank unter Linux mit SQL anlegen:
  - Kommandoschnittstelle als Administrator aufrufen:
    `psql -U postgres`

    Wenn das nicht klappt (siehe nächste Folie), muss man der Betriebssystem-Nutzer `postgres` werden: `sudo -u postgres psql`
  - Nutzer anlegen ("Role": Oberklasse von Nutzer und Gruppe):
    `CREATE ROLE brass LOGIN CREATEDB;`

    Das `CREATEDB` Recht wäre nicht nötig.

    Siehe: [https://www.postgresql.org/docs/12/sql-createrole.html]
  - Datenbank anlegen:
    ```
    CREATE DATABASE brass OWNER brass
           ENCODING 'UTF8';
    ```
  - `psql` verlassen:
    `\q`

Nutzer-Authentifizierung

- PostgreSQL hat seine eigene Liste von Nutzern.

  Datenbank-Nutzer haben zunächst nichts mit Betriebsystem-Nutzern zu tun.

- DB-Nutzer können sich eventuell auf dem Server-Rechner gar nicht einloggen (kein Betriebssystem-Account), aber schon mit dem DB-Server verbinden.

- Ob ein Password oder eine andere Authenifizierung verlangt ist, steht in der Datei `pg_hba.conf`.

  "hba": "host based authentication". Steht im "`data`" Verzeichnis.

  [https://www.postgresql.org/docs/12/client-authentication.html]

- Falls da für lokale Logins "`trust`" eingetragen ist, wird kein Passwort verlangt (egal für welchen Nutzer).

  Ggf. gilt das sogar für den Administrator `postgres`.

Tablespaces

- Ein Tablespace ist ein Verzeichnis auf dem Server, in dem Tabellen gespeichert werden (physischer Container).
- Am Anfang hat eine PostgreSQL Installation zwei Tablespaces:
  - `pg_default`: Für alle normalen Tabellen.
  - `pg_global`: Tabellen, die zu mehreren Datenbanken gehören.

    Z.B. die Liste aller Datenbanken: `pg_database`.
- Man kann zusätzliche Tablespaces anlegen.

  ```
  CREATE TABLESPACE space1 LOCATION '/mnt/sda1/postgresql/data';
  ```

  Das ist eher etwas für Experten und verkompliziert spätere Backups.
- Man kann eine Tabelle einem Tablespace zuordnen:
  `CREATE TABLE r(a int) TABLESPACE space1;`

Schemata (1)

- Eine Datenbank kann mehrere Schemata enthalten.
- Jede neu angelegte Datenbank enthält ein Schema "`public`".

  Jeder, der sich mit der Datenbank verbinden kann, kann dieses Schema nutzen.
  Man kann das Schema mit "`DROP SCHEMA public`" löschen, wenn gewünscht.
- Man kann ein Schema anlegen mit

  ```
  CREATE SCHEMA name;
  ```

  Es gibt auch eine Variante, bei dem man das Schema einem Nutzer als Besitzer zuordnet: "`CREATE SCHEMA name AUTHORIZATION nutzer;`".
- Man kann eine Tabelle in einem Schema mit folgender Notation ansprechen: "`Schema.Tabelle`", z.B.:

  ```
  SELECT * FROM public.STUDENTEN;
  ```

Schemata (2)

- Es gibt einen Suchpfad für Datenbank-Schemata. Dieser wird benutzt, wenn ein Schema nicht explizit angegeben ist:

```
SELECT * FROM STUDENTEN;
```

- Man kann den Suchpfad in `psql` anzeigen mit

```
SHOW search_path;
```

- Der Default ist: `"$user",public`.
- Das erste existierende Schema wird benutzt, um Tabellen anzulegen, für die nicht explizit ein Schema angegeben ist.

  Solange es kein Schema mit dem Namen des Nutzers gibt, wird also das `public` Schema verwendet.

- Der Suchpfad kann mit folgendem Befehl verändert werden:

```
SET search_path TO myschema,public;
```

Schemata (3)

- Man kann eine Tabelle in einem Schema anlegen mit:

```
CREATE TABLE schema.name ( ... );
```

- Durch den Suchpfad und das `public`-Schema braucht man sich nicht unbedingt mit Schemata zu befassen.

  Allerdings zeigen Werkzeuge wie `pgAdmin` die Schemata an, auch bei Anfragen an den Systemkatalog wird man öfters Schemata sehen. Deswegen ist es gut, das Konzept zu kennen.

- Eine mögliche Struktur ist, dass es nur eine Datenbank gibt, und darin ein Schema pro Nutzer.

  Das Schema heißt so wie der Nutzer. Man kann in PostgreSQL keine Anfragen stellen, die auf Tabellen in mehreren Datenbanken zugreifen, wohl aber Anfragen an Tabellen verschiedener Schemata.

Schemata (4)

- `\dn` in `psql` listet alle Schemata.

  Das "`n`" steht wohl für "Namespace".

- Jede Datenbank hat ein Schema `pg_catalog` für die System-Tabellen ("Data Dictionary").

- Dieses Schema ist implizit ganz vorn in jedem Suchpfad.

  Sofern es nicht explizit enthalten ist. Bei Bedarf könnte man es also weiter hinten einfügen. Es ist aber wichtig, dass über den Suchpfad mir niemand andere Objekte (z.B. auch Funktionen) unterschieben kann, als die, die ich beabsichtige, zu nutzen. Das Schema `public` steht normalerweise jedem offen.

- Die dort enthaltenen Tabellen beginnen mit "`pg_`".

  Man sollte solche Namen für eigene Tabellen vermeiden.

- Beispiel: `pg_catalog.pg_tables`.

Zugriffsrechte

- Es gibt später noch ein eigenes Kapitel über Zugriffsrechte.

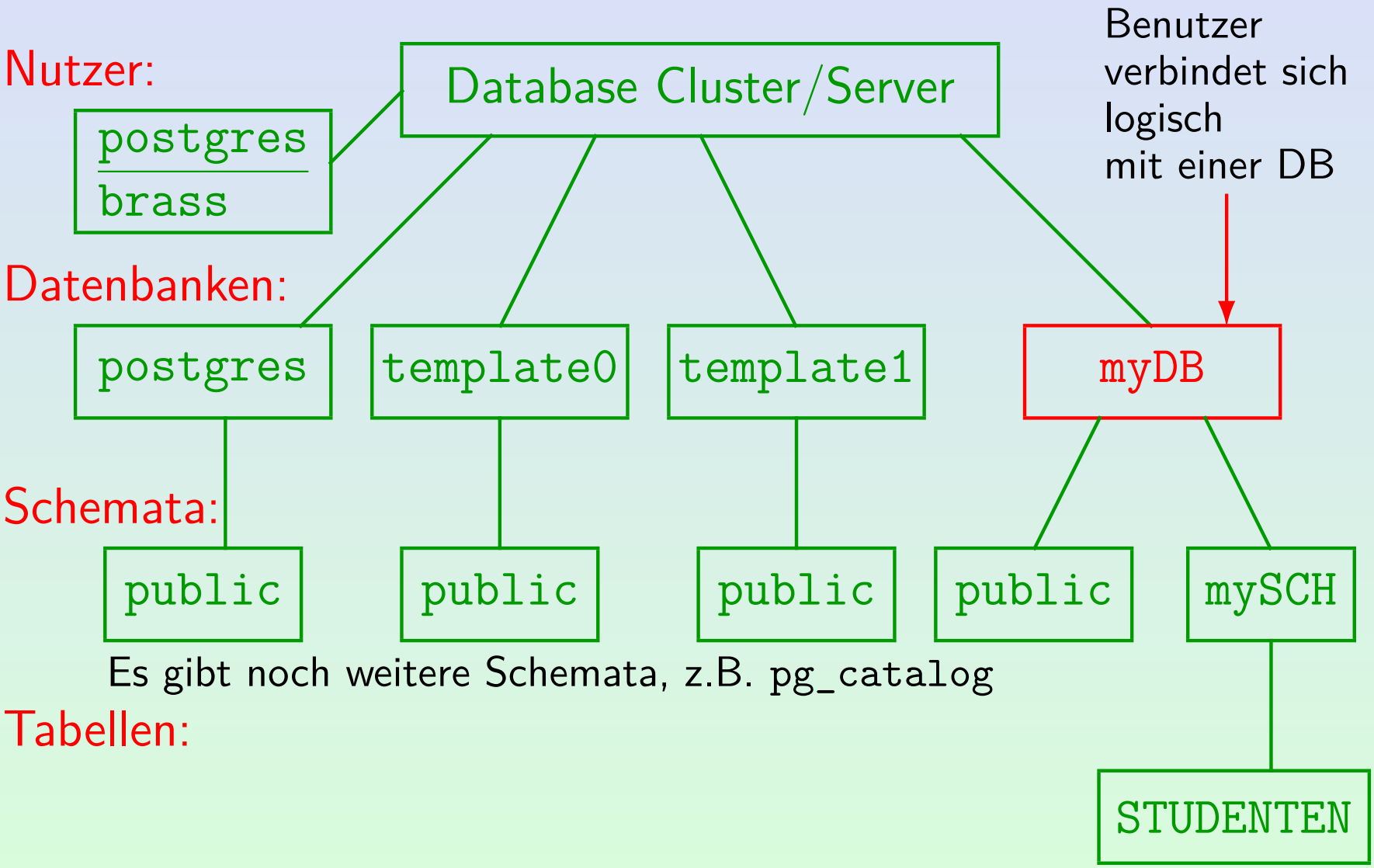
  [https://www.postgresql.org/docs/12/ddl-priv.html]

  [https://www.postgresql.org/docs/12/sql-grant.html]

- Für Datenbank-Schemata gibt zwei mögliche Rechte:
  - `USAGE`: Katalog-Information über Tabellen im Schema.
  - `CREATE`: Tabellen in Schema anlegen.
- Für das `public` Schema in einer neu angelegten Datenbank sind beide Rechte automatisch an die Nutzergruppe "`public`" vergeben, die alle Nutzer enthält.

  D.h. jeder, der sich mit der Datenbank verbinden kann, kann dort Tabellen anlegen, und sich die Namen aller existierender Tabellen anschauen. Man kann diese Rechte aber ändern (einschränken). Für Tabellen hat dagegen nur der Nutzer, der sie angelegt hat, alle Rechte (auch das kann man ändern).

Zusammenfassung/Überblick



Präsenzaufgabe: System-Katalog (1)

- Der Systemkatalog eines DBMS enthält „Metadaten“, also „Daten über Daten“, und insbesondere das DB-Schema.

  Das „Information Schema“ ist Teil des SQL-Standards.

  PostgreSQL hat daneben noch seinen eigentlichen „nativen“ Systemkatalog.

- In der Tabelle `information_schema.columns` sind alle Spalten von allen Tabellen der Datenbank eingetragen.

  [https://www.postgresql.org/docs/9.3/infoschema-columns.html]

- Schreiben Sie eine Anfrage, die alle Spalten liefert von einem Typ `numeric(p,s)` mit Nachkommastellen ($s \neq 0$) und höchstens 4 Dezimalstellen ($p \leq 4$).

  Spalten: `data_type` („`numeric`“), `numeric_precision` (p), `numeric_scale` (s).

- Geben Sie `table_schema`, `table_name`, `column_name` und den Datentyp aus, z.B. `NUMERIC(4,1)`.

  Die String-Konkatenation `||` kann auch auf Zahlen angewendet werden.

Präsenzaufgabe: System-Katalog (2)

- Erwartetes Ergebnis:

table_schema	table_name	column_name	datentyp
student...	bewertungen	punkte	NUMERIC(4,1)
sakila_public	film	rental_rate	NUMERIC(4,2)
sakila_public	film_list	price	NUMERIC(4,2)
sakila_public	nicer_...	price	NUMERIC(4,2)

[https://dbs.informatik.uni-halle.de/edb?pgsql=db&username=student_gast&db=postgres&ns=]

Die Zugangsdaten unserer Installation stehen in StudIP, Reiter „Adminer“.

- Anfragen müssen nicht nur mit dem aktuellen Zustand funktionieren, sondern mit beliebigen DB-Zuständen.
- Sie können sich also nicht darauf verlassen, dass alle Ergebnis-Typen `numeric(p,s)` die Bedingung $p = 4$ erfüllen. Die Bedingung der Aufgabe ist $p \leq 4$.

Präsenzaufgabe: System-Katalog (1)

- Es sind alle Tabellen gesucht, die in Fremdschlüsseln die Tabelle `studenten` referenzieren.

  Es soll hier der „native“ Systemkatalog von PostgreSQL genutzt werden.

- Tabellen sind in `pg_catalog.pg_class` eingetragen.
  Wir benötigen nur die Spalten `relname` (Name der Tabelle) und `oid` (Interne Nummer, Object-ID).

  [https://www.postgresql.org/docs/9.1/catalog-pg-class.html]

- Constraints stehen in `pg_catalog.pg_constraint`.

  [https://www.postgresql.org/docs/9.1/catalog-pg-constraint.html]

  - `conrelid`: OID der Tabelle mit diesem Constraint.

    Den Namen dieser Tabelle sollen Sie ausgeben.

  - `contype`: Art des Constraints, z.B. `'f'` für Fremdschlüssel.
  - `confrelid`: OID der referenzierten Tabelle.

    Dies soll die Tabelle `studenten` sein.

Präsenzaufgabe: System-Katalog (2)

- Erwartetes Ergebnis:

relname
bewertungen

[https://dbs.informatik.uni-halle.de/edb?pgsql=db&username=student_gast&db=postgres&ns=]

Die Zugangsdaten unserer Installation stehen in StudIP, Reiter „Adminer“.

- `bewertungen( sid`$\rightarrow$`studenten, (atyp,anr)`$\rightarrow$`aufgaben, punkte)`
- Anfragen müssen nicht nur mit dem aktuellen Zustand funktionieren, sondern mit beliebigen DB-Zuständen.
- Das Schema „`pg_catalog`“ wird immer zuerst durchsucht, wenn es nicht explizit in `search_path` ist.

D.h. man kann sich den Schema-Präfix bei diesen Katalog-Tabellen sparen.

Präsenzaufgabe: System-Katalog (3)

Tipps:

- Wenn Sie

  ```
  SELECT * FROM pg_class
  ```

  eingeben, wird die Spalte `oid` nicht angezeigt.

  Sie ist eine spezielle interne Spalte (ein wenig versteckt).

- Es gibt diese Spalte aber. Z.B. funktioniert Folgendes:

  ```
  SELECT stud.oid
  FROM   pg_class stud
  WHERE  stud.relname = 'studenten'
  ```

- Beim Verbund müssen die Spaltennamen nicht identisch sein, z.B. funktioniert

  ```
  fk.confrelid = stud.oid
  ```

  wenn `fk` eine Tupelvariable über `pg_constraint` ist.