Datenbank-Programmierung

Chapter 2: Benutzung von PostgreSQL

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2018/19

http://www.informatik.uni-halle.de/~brass/dbp19/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Den PostgreSQL Datenbank Server starten und stoppen
- SQL-Befehle mittels `psql` an die Datenbank schicken
- `psql`-Skripte schreiben

Inhalt

1. Einführung
2. Administratives Grundwissen
3. psql Kommandoschnittstelle
4. SQL-Skripte
5. Ausgabe-Formatierung

Geschichte des Systems (1)

- POSTGRES wurde von Michael Stonebraker (+ Team) an der University of California at Berkeley ab 1985 entwickelt.

  Michael Stonebraker hatte zuvor das INGRES-System entwickelt (ebenfalls in Berkeley), eines der beiden ersten relationalen Datenbanksysteme (das andere war System R, Forschungsprototyp von IBM). Michael Stomebraker bekam 2014 den Turing Award, eine Art Nobelpreis für Informatik [https://amturing.acm.org/byyear.cfm].

- Ein Prototyp wurde auf der SIGMOD Konferenz 1988 gezeigt, Version 1 wurde 1989 fertig gestellt.

- POSTGRES war Vorreiter der objektrelationalen Datenbanken.

  Michael Stonebraker gründete zusammen mit anderen die Firma Illustra Information Technologies, die eine kommerzielle Version von POSTGRES herausbrachte. Sie wurde 1997 von Informix gekauft (Informix 2001 von IBM).

Geschichte des Systems (2)

- Das POSTGRES Projekt endete 1994.
- 1994 entwickelten Andrew Yu und Jolly Chen (Studenten in Berkeley) eine Version mit SQL-Unterstützung (statt der Datenbank-Sprache QUEL/POSTQUEL).

  Die erste Version erschien 1995 ("Postgres95"). Die freie Weiterentwicklung war möglich, weil POSTGRES unter einer großzügigen Open Source Lizenz herausgegeben wurde (MIT Lizenz).

- Der Name "PostgreSQL" wurde 1996 eingeführt.

  Die erste PostgreSQL Release war 6.0, erschienen am 29.01.1997.

- Im Ranking von DB-Engines.com belegt PostgreSQL den vierten Platz (hinter Oracle, MySQL, Microsoft SQL Server).

  [https://db-engines.com/de/ranking]

  Siehe auch [https://www.datanyze.com/market-share/databases].

Verfügbarkeit

- Am 13. Februar 2020 wurden gleichzeitig folgende Versionen veröffentlicht: 12.2, 11.7, 10.12, 9.6.17, 9.5.21, 9.4.26.

  Die älteren Versionen erhalten im wesentlichen nur Fehlerbehebungen.

  Für 9.4 gibt es keine weiteren Updates, 9.5 wird ab 2/2021 nicht mehr gepflegt.

- Folgende Plattformen werden mit fertigen Installations-Paketen unterstützt [https://www.postgresql.org/download/]:
  - Linux (Red Hat/CentOS/Fedora, Debian, Ubuntu, SUSE)
  - BSD (FreeBSD, OpenBSD)
  - Windows
  - macOS
  - Solaris

- Der Quellcode ist auch frei verfügbar.

  [https://www.postgresql.org/ftp/source/] [https://git.postgresql.org/gitweb/]

Nutzung von PostgreSQL (1)

- PostgreSQL ist ein Client-Server-System.

  Der Server besteht aus mehreren Prozessen und einem “Shared Memory” Bereich.

- Die Kommunikation zwischen Client und Server erfolgt über eine Netzwerk-Verbindung.

  Wenn Client und Server auf dem gleichen Linux-Rechner laufen, wird ein “UNIX-Domain-Socket” zur Kommunikation verwendet (Inter-Prozess-Kommunikation über temporäre Datei).
  Unter Windows wird aber auch in diesem Fall TCP/IP benutzt.

- Die normale Kommandozeilen-Schnittstelle ist das Programm `psql`.

  Dies ist nur ein Client, der SQL Befehle entgegennimmt und an den Server schickt, dann das Ergebnis abholt und ausgibt. Natürlich gibt es einige Formatierungsmöglichkeiten, Abkürzungen, Einstellungen etc. (s.u.).

Nutzung von PostgreSQL (2)

- Eine bekannte graphische Schnittstelle is `pgAdmin`.

  [https://www.pgadmin.org/]

- Es ist in erster Linie zur Administration gedacht, erlaubt aber auch, SQL-Anfragen zu stellen.

  Und enthält einen Browser für alle Datenbank-Objekte in einer Baum-Ansicht (Expand/Collapse).

- Es gibt einen Treiber für die JDBC-Schnittstelle zur Kommunikation mit der Datenbank aus Java-Programmen.

  [https://jdbc.postgresql.org/]
  Auch ODBC wird unterstützt: [https://odbc.postgresql.org/]

- Zu den unterstützten Programmiersprachen gehören u.a. Java, C/C++, PHP, Perl, Python.

Inhalt

1. Einführung
2. Administratives Grundwissen
3. psql Kommandoschnittstelle
4. SQL-Skripte
5. Ausgabe-Formatierung

Starten und Stoppen des Servers (1)

- Wenn man `psql` aufruft, und der Server läuft nicht, bekommt man die Fehlermeldung "`could not connect to server`".
- Man kann auch schauen, ob `postgres` Prozesse laufen.

  Unter Linux z.B.: `ps -ef | fgrep postgres`

  Unter Windows mit dem Taskmanager (Ctrl+Alt+Delete).
- Man kann server-basierte Datenbanken so installieren,
  - dass der Server beim Start des Betriebssystems automatisch mit gestartet wird, oder

    Bequem, aber das Hochfahren des Betriebssystems dauert etwas länger, und einige Systemresourcen sind belegt, auch wenn man die Datenbank nicht nutzt.
  - dass man ihn manuell starten muss (bei Bedarf).

Starten und Stoppen des Servers (2)

- Bei vielen Datenbanken ist es sehr wichtig, sie vor dem Shutdown des Betriebssystems ordnungsgemäß herunterzufahren.

  Sonst kann beim nächsten Hochfahren ein Recovery nötig sein, weil nicht alle veränderten Hauptspeicherinhalte geschrieben wurden. Dabei können die Daten aller beendeten Transaktionen wieder hergestellt werden (wenn alles richtig funktioniert), aber das Hochfahren dauert dann wesentlich länger.

- Linux schickt vor dem endgültigen Shutdown allen Prozessen ein `SIGTERM` Signal, und der PostgreSQL Server reagiert darauf (fährt geordnet herunter).

  Allerdings erst, wenn sich alle laufenden Sitzungen beendet haben. Eventuell könnte das dem Betriebssystem zu lange dauern.

Starten und Stoppen des Servers (3)

- Das Programm `pg_ctl` (“postgres control”) dient u.a. zum Starten und Stoppen des Servers.

- Es kann nur vom Administrator verwendet werden, d.h. unter Linux muss man der Nutzer `postgres` sein.

- Dann kann man eventuell einfach “`pg_ctl start`” eingeben.

  Wenn sich die Daten bei `/var/lib/pgsql/data` befinden oder die Umgebungsvariable `PGDATA` entsprechend gesetzt ist.

- Mit Nutzerwechsel und mehr Parametern sieht es so aus:

```
sudo -u postgres \
pg_ctl -D /data/pg -l /data/logfile -w start
```

  Mit `-D` wird das Hauptverzeichnis von Postgres angegeben, mit `-l` eine Logdatei für Fehlermeldungen (sonst Terminal, von dem `pg_ctl` aufgerufen). Die Option `-w` bedeutet, dass `pg_ctl` wartet, bis der Server gestartet ist.

Starten und Stoppen des Servers (4)

- Entsprechend kann man den Server geordnet herunterfahren mit "pg_ctl stop".
- Mit Nutzerwechsel und mehr Parametern:

```
sudo -u postgres \
pg_ctl -D /data/pg -m fast -w stop
```

Mit "-m fast" ("shutdown mode fast") kann man verlangen, dass eventuell laufende Sitzungen abgebrochen werden.
Der Default ("-m smart") ist, zu warten, bis sich alle laufenden Sitzungen freiwillig beenden.
Die dritte Möglichkeit ist "-m immediate", das würde die Prozesse sofort beenden, und damit aber ein Recovery beim nächsten Start nötig machen, weil ein eventuell veränderter Hauptspeicher-Inhalt nicht mehr gesichert wird.

- "pg_ctl status" zeigt an, ob der Server aktuell läuft.

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]

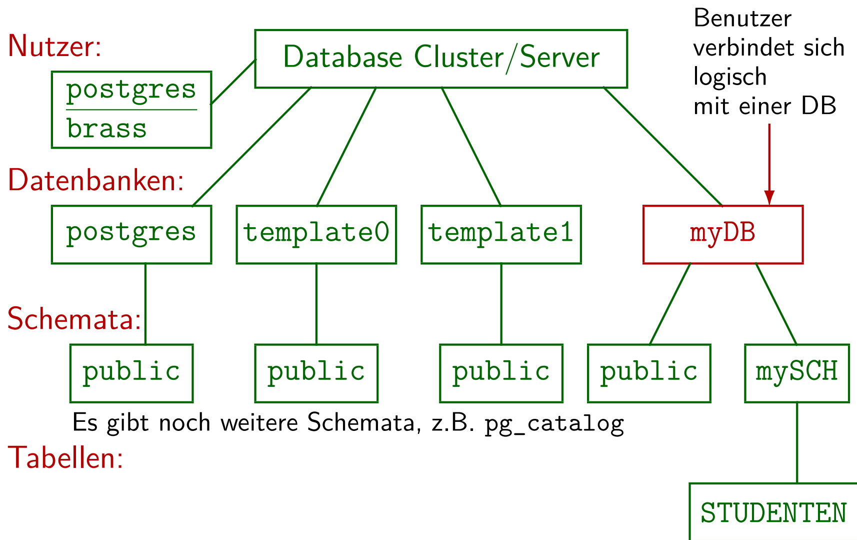
  [https://www.postgresql.org/docs/12/sql-grant.html]

- 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



Inhalt

1. Einführung
2. Administratives Grundwissen
3. psql Kommandoschnittstelle
4. SQL-Skripte
5. Ausgabe-Formatierung

Verbindung zum Server (1)

- Wenn man nicht explizit einen Server angibt, verbindet sich `psql` über einen Unix-Domain Socket oder TCP/IP zu einem Server auf dem lokalen Rechner.

  Auf UNIX/Linux-Rechnern ist der Default "Unix-Domain Socket". Das kann scheitern, obwohl ein Zugriff über TCP/IP zu `localhost` möglich wäre. Man sollte probehalber die Verbindungsdaten explizit angeben, wie unten gezeigt. Der TCP/IP Port ist normalerweise 5432 (aber konfigurierbar).

- Verbindung zu Port 5432 auf dem Rechner `localhost`, Datenbank `brass`:

```
psql -h localhost -p 5432 brass
```

  "`brass`" ist hier der Name der Datenbank (alternativ: `-d brass`). Man kann danach noch den Nutzernamen angeben, alternativ mit der Option `-U`. Nach `-h` kann man auch einen Socket angeben, z.B. `/var/run/postgresql`. In `psql` zeigt `\conninfo` die Verbindung zum Server an.
  [https://www.postgresql.org/docs/12/app-psql.html]

Verbindung zum Server (2)

- Zusammenfassung (Daten zur Verbindung mit DB-Server):

Parameter	Option	Default	Env.-Var.
Host-Rechner	`-h`	(Socket)	`PGHOST`
Port	`-p`	5432	`PGPORT`
Datenbank	`-d`	(wie OS-Nutzer)	`PGDATABASE`
DB-Nutzer	`-U`	(wie OS-Nutzer)	`PGUSER`

  Die Daten können über Umgebungsvariablen gesetzt werden, wenn man sie nicht bei jedem Aufruf von `psql` eingeben will.
  [https://www.postgresql.org/docs/12/libpq-envars.html]

- Ggf. wird zusätzlich ein Passwort verlangt.

  Man kann die Daten (inkl. Passwort) auch in eine Datei `.pgpass` schreiben.
  [https://www.postgresql.org/docs/12/libpq-pgpass.html]

psql Nutzung: Das Wichtigste

- Man kann SQL-Befehle über mehrere Zeilen verteilt eingeben und muss sie dann mit einem Semikolon "`;`" abschließen.

  Der Prompt enthält den Datenbank-Namen, z.B. `testdb=>`.

  Ggf. offene Klammern oder Anführungszeichen werden im Prompt angezeigt.

- Außerdem gibt es Befehle, die von `psql` selbst ausgeführt werden. Sie beginnen mit einem Rückwärts-Schrägstrich "\" und werden direkt ausgeführt, sobald Enter gedrückt wird.
- `\h`: Hilfe zu SQL-Anweisungen.
- `\?`: Hilfe zu `psql`-Befehlen.
- `\d`: Liste aller Tabellen/Sichten etc.
- `\d TAB`: Schema der Tabelle `TAB` anzeigen ("describe").
- `\q`: `psql` beenden ("quit").

psql vs. SQL

- Es ist wichtig, den Unterschied zwischen `psql`-Befehlen und SQL-Befehlen zu verstehen:
  - `psql`-Befehle beginnen mit `\` und erstrecken sich bis zum Ende der Zeile.
  - Sie werden sofort ausgeführt (von `psql`, also dem Client).
  - Alles andere hält `psql` für einen SQL-Befehl.
  - Es sammelt ggf. mehrere Zeilen in einem Puffer, bis zu einem Semikolon "`;`" (Prompt `->` statt `=>` für Fortsetzung).

    Die Eingabe kann auch mit speziellen psql-Befehlen beendet werden, wie z.B. `\r` ("reset"), was die Eingabe abbricht und den Puffer leert.
  - Beim "`;`" wird der SQL-Befehl an den Server geschickt, das Ergebnis geholt, und angezeigt.

    Nach der Ausführung steht die Anfrage weiter im Puffer.

Schema-Information (1)

- `\dt`: Liste aller Tabellen anzeigen (ohne Katalog-Tabellen)
- Alternativ mit Tabelle aus PostgreSQL Data Dictionary:

```
SELECT *
FROM   pg_catalog.pg_tables;
```

  Es reicht `pg_tables` zu schreiben, die Tabelle wird über den Schema-Suchpfad gefunden. Spalten: `schemaname`, `tablename`, `tableowner`, `tablespace`, `hasindexes`, `hasrules`, `hastriggers`.

- Oder mit "Information Schema" aus dem SQL Standard:

```
SELECT table_schema || '.' || table_name
FROM   information_schema.tables
WHERE  table_type = 'BASE TABLE'
AND    table_schema NOT IN
       ('pg_catalog', 'information_schema');
```

Schema-Information (2)

- Die internen Anfragen für `\d` und ähnliche Kommandos werden angezeigt, wenn man `psql` mit Option `-E` aufruft.

```
SELECT n.nspname as "Schema",
       c.relname as "Name",
       CASE c.relkind
            WHEN 'r' THEN 'table'
            WHEN 'v' THEN 'view'
            WHEN 'i' THEN 'index'
            WHEN 'S' THEN 'sequence'
            WHEN 's' THEN 'special'
            WHEN 'f' THEN 'foreign table' END as "Type",
       pg_catalog.pg_get_userbyid(c.relowner) as "Owner"
FROM   pg_catalog.pg_class c
       LEFT JOIN pg_catalog.pg_namespace n
                 ON n.oid = c.relnamespace
WHERE  c.relkind IN ('r','v','S','f','')
AND    n.nspname <> 'pg_catalog'
AND    n.nspname <> 'information_schema'
AND    n.nspname !~ '^pg_toast'
AND    pg_catalog.pg_table_is_visible(c.oid)
ORDER  BY 1,2;
```

Schema-Information (3)

\d STUDENTEN:

```
             Table "public.studenten"
  Column  |          Type          | Modifiers
----------+------------------------+-----------
 sid      | numeric(3,0)           | not null
 vorname  | character varying(25) | not null
 nachname | character varying(25) | not null
 email    | character varying(80) |
Indexes:
    "studenten_pk" PRIMARY KEY, btree (sid)
Check constraints:
    "studenten_sid_positiv" CHECK (sid > 0::numeric)
Referenced by:
    TABLE "bewertungen" CONSTRAINT "bew_ref_stud"
        FOREIGN KEY (sid) REFERENCES studenten(sid)
```

Schema-Information (4)

Weitere Informationen:

- Alle Tabellen abfragen, auch mit Katalog:
  ```
  \dt *
  ```
- Nur Tabellen eines bestimmten Schemas:
  ```
  \dt schema_name.*
  ```
- Liste aller Datenbanken mit Speicher-Größe:
  ```
  \l+
  ```
- Datenbank, mit der man aktuell verbunden ist:
  ```
  SELECT current_database();
  ```
- PostgreSQL-Version:
  ```
  SELECT version();
  ```

Editieren der Eingabe (1)

- psql verwendet die GNU Readline-Bibliothek zum Einlesen der SQL-Kommandos.

  [https://tiswww.cwru.edu/php/chet/readline/rluserman.html]

  [https://www.gnu.org/software/bash/manual/html_node/Bindable-Readline-Commands.html]

  [https://en.wikipedia.org/wiki/GNU_Readline]

- Dies gibt recht mächtige Editiermöglichkeiten für die Eingabe:
  - ←, →: Cursor in der aktuellen Zeile bewegen.

    Wenn die Cursortasten nicht funktionieren: `Ctrl+F` (forward), `Ctrl+B` (backward). `Alt+F`, `Alt+B`: Wortweise springen.
  - `Ctrl+A`, `Ctrl+E`: Zum Anfang/Ende der Zeile.
  - `Backspace`, `Delete`: Zeichen löschen links/rechts vom Cursor.
  - Normale Zeichen: Werden eingefügt.

Editieren der Eingabe (2)

- Editieren der Eingabe, Forts.:
  - `Ctrl+K`: Rest der Zeile löschen ("kill") →Clipboard.

    `Alt+D`: Rest des aktuellen Wortes löschen.
  - `Ctrl+U`: Von Cursor bis Anfang der Zeile löschen.

    `Ctrl+W`: Nach links bis zum nächsten Leerplatz (Whitespace) löschen.
  - `Ctrl+Y`: Gelöschen Text einfügen.

    "yank": reißen, zerren, herausziehen. Wenn vorher z.B. mehrere Worte mit `Alt+D` gelöscht wurden, werden sie jetzt alle eingefügt.
  - `Ctrl+X Ctrl+U`: Undo.

    Hier muss man beide Tasten nacheinander drücken. Nach der Anleitung geht auch `Ctrl+_`, aber bei mir wird das vom Terminalfenster abgefangen (und verkleinert den Font). `Alt+R` ("revert") macht alles rückgängig, löscht eine neue Eingabe also vollständig.

Editieren der Eingabe (3)

- Die Readline-Bibliothek speichert die vorigen Eingabezeilen.

  In `.psql_history` im Home-Verzeichnis sogar über eine Sitzung hinaus.

- Zugriff auf die Historie:
  - ↑: Zur vorigen Eingabezeile.

    Wenn Cursortaste nicht geht: `Ctrl+P` ("previous").
  - ↓: Zurück zur nächsten Zeile der Historie.

    Alternativ: `Ctrl+N` ("next").
  - `Ctrl+R`: Suche in Historie.

    "Reverse search history". Man kommt zum nächsten (frischesten) Eintrag, der den dann eingegebenen String entält. Durch mehrmaliges Drücken von `Ctrl+R` auch noch weiter. Wenn man das Kommando ausführen will, drückt man die Eingabetaste (Return/Enter). Wenn man die Suche abbrechen will und die Original-Zeile wiederherstellen, drückt man `Ctrl+G`.

Editieren der Eingabe (4)

- Die Einträge in der Historie sind nicht einzelne Zeilen, sondern ganze SQL-Kommandos (ggf. mehrere Zeilen).

  Das ist eigentlich logisch und praktisch. Leider ist GNU Readline nur zum Editieren einzelner Zeilen gemacht, wie etwa Kommandozeilen in einer Shell. Bei der Verwendung in `psql` stößt es an Grenzen. Z.B. kann man sich nicht mit "Cursor hoch" zur vorigen Zeile bewegen (damit kommt man zum vorigen Eintrag in der Historie). Die schnellste mir bekannte Bewegung ist wortweise mit `Alt+F` (forward) und `Alt+B` (backward). Vielleicht hilft es auch, mit `Ctrl+A` zum Anfang zu springen. Wenn man einen Zeilenumbruch einfügen will, muss man `Ctrl+V Ctrl+J` drücken (`Ctrl+V` erlaubt die Eingabe von Steuerzeichen, `Ctrl+J` ist der Linefeed, die `Return`-Taste wäre nur `Ctrl+M`).

- Dagegen arbeitet `psql` für die aktuelle Anfrage zeilenweise: Wenn man `Return` gedrückt hat, kann man nicht mehr mit ← oder `Ctrl+B` zurück.

Editieren der Eingabe (5)

- Den aktuellen SQL-Befehl (Puffer-Inhalt) kann man mit dem psql-Befehl `\e` in einem normalen Editor ändern.

  Man bekommt `vi` unter Linux und `notepad.exe` unter Windows. Mit den Umgebungsvariablen `PSQL_EDITOR` bzw. `EDITOR` kann man den Editor wählen.

  Falls unabsichtlich in `vi` gekommen: "`:q`" bzw. "`:q!`" beendet den Editor.

  Wahl des Editors in `psql` (nur für die aktuelle Sitzung): `\setenv EDITOR pico`

- `\e` geht auch mitten in einer unfertigen SQL-Anfrage.

  Natürlich nicht innerhalb `'...'`. Alle \-Kommandos werden sofort ausgeführt.

  Mit mit \e gespeicherte Datei wird wie eine Eingabe behandelt ("`;`": ausführen).

- `\g`: Aktuelle Anfrage (Puffer) (nochmals) ausführen ("go").
- `\p`: Puffer-Inhalt anzeigen ("print").
- `\w ⟨file⟩`: Eingabepuffer in Datei schreiben ("write").
- `\i ⟨file⟩`: Datei ausführen ("include").

Editieren der Eingabe (6)

- Beispiel (Tippfehler, Zeile schon beendet):

```
testDB=> select vname, nachname
testDB->
```

- Man kann jetzt Folgendes eingeben:
  - `;`: Gibt Fehlermeldung, aber Anfrage in Historie.

    Anschließend kommt man mit ↑ zu der Anfrage. Man kann jetzt den Readline-Zeileneditor nutzen, um die Anfrage zu korrigieren.
  - `\r`: Anfrage abbrechen und löschen.

    Sie ist dann nicht Teil der Historie, der schon geschriebene Anfang ist also weg. Man könnte Stücke mit der Copy&Paste-Funktion des Terminalprogramms in den nächsten Versuch übernehmen (umständlich).
  - `\e`: Editor aufrufen, anschließend "`;`".
  - `\w x.sql`: Anfrage in Datei speichern.

    Editieren mit wählbarem Programm: `\! vi x.sql`, ausführen: `\i x.sql`

Kommando-Vervollständigung

- Wenn man eine SQL-Anfrage, die Tabulator-Zeichen enthält, mit Copy&Paste in `psql` kopiert, funktioniert es meist nicht.
- Grund ist, dass die GNU Readline Bibliothek auch Kommando-Vervollständigung enthält: Wenn man z.B. `sel`⟨`Tab`⟩ tippt, erscheint automatisch `select`.
- Leider funktioniert das ziemlich schlecht.

  Es versteht sehr wenig von der SQL-Syntax und nichts vom aktuellen Schema.

- Man kann es abschalten in der Konfigurationsdatei `.inputrc` (der Readline-Bibliothek) im Home-Verzeichnis.

  Man schreibt folgenden Text in die Datei `.inputrc`:

```
$if psql
set disable-completion on
$endif
```

Ausgabe langer Tabellen

- Falls das Anfrage-Ergebnis länger als der Bildschirm ist, benutzt `psql` normalerweise ein Programm wie `more`, um die Ausgabe nach jeweils einer Seite anzuhalten.

  [http://man7.org/linux/man-pages/man1/more.1.html]

  - Leertaste: Eine Seite weiter.
  - Eingabetaste ("Return"): Eine Zeile weiter.
  - `q`: Abbrechen.
  - /*s*: Springen zu Vorkommen von Zeichenkette *s*.

- Mit "`\pset pager off`" kann man die Nutzung eines solchen Programms abschalten (ganze Ausgabe auf einmal).

  Gilt nur für die aktuelle Sitzung, aber man kann eine Datei `.psqlrc` im Homeverzeichnis anlegen, die immer beim Start gelesen wird. Mit der Umgebungsvariable `PAGER` kann man ein Programm wählen.

Inhalt

1. Einführung
2. Administratives Grundwissen
3. psql Kommandoschnittstelle
4. SQL-Skripte
5. Ausgabe-Formatierung

SQL-Skripte (1)

- Diese Vorlesung heisst ja "Datenbank-Programmierung".
- Manche Aufgaben können durch Skripte implementiert werden, die von einer SQL-Kommandoschnittstelle wie `psql` ausgeführt werden.

  Ggf. noch mit einem Shellskript drum herum.
- So bekommt man zwar keine hübsche graphische Benutzerschnittstelle, aber die Methode eignet sich z.B. für
  - den wiederkehrenden Export von Daten in verschiedenen Formaten (z.B. CSV, JSON, XML, LaTeX),
  - einfache Berichte aus der Datenbank (Statistiken),
  - wiederkehrende administrative Aufgaben (z.B. Löschen/Archivieren alter Daten).

SQL-Skripte (2)

Erstes Beispiel:

- Man schreibt einen SQL-Befehl in eine Datei `lisa.sql`:

```
SELECT B.ANR, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.VORNAME  = 'Lisa'
AND    S.NACHNAME = 'Weiss'
AND    B.SID = S.SID AND B.ATYP = 'H'
ORDER  BY B.ANR;
```

- In `psql` kann man die Datei ausführen mit:

```
\i lisa.sql
```

- Auf der UNIX/Linux-Kommandozeile mit:

```
psql -f lisa.sql
```

"`-f`" wie "file". Ggf. sind außerdem weitere Optionen anzugeben, siehe Folie 27.

SQL-Skripte (3)

Variablen:

- Angenommen, das obige Skript soll für einen beliebigen Studenten funktionieren, der über Vorname und Nachname ausgewählt wird.

  D.h. Vorname und Nachname sind Parameter/Eingabewerte des Skripts.

- `psql` hat dafür Variablen:

```
\set Vorname  'Lisa'
\set Nachname 'Weiss'
```

  Da es eine Reihe von vordefinierten Variablen gibt, die nur aus Großbuchstaben bestehen, wird empfohlen, dass Nutzer-Variablen mindestens einen Kleinbuchstaben enthalten. Variablennamen sind case-sensitiv.

- In allen folgenden Kommandos (SQL und psql) wird dann `:Vorname` durch `Lisa` ersetzt (ohne Anführungszeichen).

SQL-Skripte (4)

Variablen, Forts.:

- In der SQL-Anfrage sind Anführungszeichen `'...'` nötig, dafür schreibt man `:'Vorname'`.

  Dagegen würde `':Vorname'` nicht funktionieren, da der Doppelpunkt im Innern von String-Konstanten keine besondere Bedeutung hat. Wenn man den Variablenwert in `"..."` braucht (für Spaltennamen), geht das auch: `:"Var"`.

- Variablen können auch auf der Kommandozeile gesetzt werden (z.B. wichtig für Shell-Skripte):

  ```
  psql -f hw.sql -v Vorname=Lisa -v Nachname=Weiss
  ```

- Das `psql`-Skript kann auch eine Nutzer-Eingabe verlangen:

  ```
  \prompt 'Bitte Vorname eingeben: ' Vorname
  ```

- Ausgaben in `psql`-Skripten gehen mit `\echo`:

  ```
  \echo :Vorname
  ```

SQL-Skripte (5)

SQL-Skript mit Variablen:

- Hausaufgaben-Punkte für wählbaren Studenten:

```
-- Beispiel-Skript
\prompt 'Bitte Vornamen eingeben:  ' Vorname
\prompt 'Bitte Nachnamen eingeben: ' Nachname
\echo Hausaufgaben von :Vorname :Nachname

SELECT B.ANR, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.VORNAME = :'Vorname'
AND    S.NACHNAME = :'Nachname'
AND    B.SID = S.SID
AND    B.ATYP = 'H' -- Hausaufgaben-Punkte
ORDER  BY B.ANR;
```

SQL-Skripte (6)

Details zu Ausgaben mit \echo:

- `\echo` druckt mehrere durch Leerplatz getrennte Argumente (Worte auf der Zeile) mit jeweils einem Leerzeichen getrennt.

  Unabhängig davon, wie viele Leerzeichen zwischen den Argumenten standen.

- Die Kontrolle über Leerplatz ist mit Stringkonstanten möglich:

  ```
  \echo 'X     Y'
  ```

  Dies gibt wirklich mehrere Leerzeichen zwischen `X` und `Y` aus.

  Die Anführungszeichen `'...'` werden nicht ausgegeben (`"..."` dagegen schon).

- Wenn man keinen Zeilenumbruch am Ende wünscht, kann man die Option `-n` als erstes Argument angeben:

  `\echo -n 's'` (mit *s* kursiv)

  Dies gibt exakt den String *s* aus und nichts sonst. Z.B. “`:`” direkt nach Namen:

  ```
  \echo -n 'Hausaufgaben von' :Vorname :Nachname
  \echo ':'
  ```

SQL-Skripte (7)

Variable auf Anfrage-Ergebnis setzen:

- Mit `\gset` (seit Ver. 9.3) werden Variablen auf das Ergebnis der Anfrage im Puffer gesetzt (darf nur eine Zeile liefern):

```
SELECT SUM(B.PUNKTE) AS "Gesamt"
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.VORNAME = :'Vorname'
AND    S.NACHNAME = :'Nachname'
AND    B.SID = S.SID
AND    B.ATYP = 'H' -- Hier kein ;
\gset
\r
```

- Die Variable `Gesamt` enthält nun die Gesamtpunktzahl.

  Die Variablen heißen wie die Ergebnisspalten (besser `"..."` verwenden).

  “`\gset x_`”: Präfix “`x_`” für Variablen. Falls 0 Zeilen: Variablen unverändert.

SQL-Skripte (8)

Ausgabe in Datei schreiben:

- `\o ⟨file⟩` ("output destination") leitet die Ausgabe der folgenden Anfragen in die Datei ⟨file⟩ um.

  Mit `\o |⟨cmd⟩` kann man die Ausgabe auch als Eingabe für ein beliebiges UNIX/Linux-Kommando verwenden ("pipe" wie in der Shell).

- `\qecho '⟨text⟩'` ("query echo") schreibt ⟨text⟩ dorthin, wohin auch Anfrage-Ergebnisse geschrieben werden (Datei).

  Dagegen schreibt der normale \echo-Befehl auch nach der Ausgabe-Umleitung mit \o weiter auf die Standard-Ausgabe (normalerweise den Bildschirm).

  Die Ausgabe kann von einem beliebigen Programm stammen: `` \qecho `⟨prog⟩` `` (wenn man "backticks" verwendet, wird der Linux-Befehl darin ausgeführt).

- Es ist möglich, in die Ausgabe-Datei SQL-Anweisungen zu schreiben, und diese anschließend mit `\i` auszuführen.

  Auf diese Art bekommt man auch eine Art von Schleifen.

Inhalt

1 Einführung

2 Administratives Grundwissen

3 psql Kommandoschnittstelle

4 SQL-Skripte

5 Ausgabe-Formatierung

Ausgabe-Formatierung (1)

- Das Standard-Ausgabeformat von Tabellen ist so:

```
sb=> select * from studenten;
 sid | vorname | nachname |        email
-----+---------+----------+---------------------
 101 | Lisa    | Weiss    | weiss@acm.org
 102 | Michael | Grau     |
 103 | Daniel  | Sommer   | daniel@gmx.de
 104 | Iris    | Winter   | irisw@gmail.com
(4 rows)
```

- Mit Rot hervorgehoben ist der Rahmen der Tabelle (erste zwei Zeilen der Ausgabe und Zusammenfassung am Ende).
- Diesen kann man abstellen mit: `\pset tuples_only on`

Ausgabe-Formatierung (2)

- Normal werden die Datenwerte mit Leerzeichen aufgefüllt, so dass die Werte einer Spalte übereinander stehen.
  Das ist das Ausgabeformat "`aligned`":

```
␣101␣|␣Lisa␣␣␣␣|␣Weiss␣␣␣␣|␣weiss@acm.org
␣102␣|␣Michael␣|␣Grau␣␣␣␣␣|␣
␣103␣|␣Daniel␣␣|␣Sommer␣␣␣|␣daniel@gmx.de
␣104␣|␣Iris␣␣␣␣|␣Winter␣␣␣|␣irisw@gmail.com
```

- Beim Format "unaligned" werden keine extra Leerzeichen eingefügt (`\pset format unaligned`):

```
101|Lisa|Weiss|weiss@acm.org
102|Michael|Grau|
...
```

`\pset fieldsep '\t'` ersetzt den "|" durch ein anderes Zeichen (z.B. TAB).

Ausgabe-Formatierung (3)

- Wenn die Tupel breiter als eine Terminal-Zeile sind, kann man "`\pset format wrapped`" probieren. Spaltenwerte werden dann teils über mehrere Zeilen verteilt.

  Das wird durch einen Punkt am Ende jeder Zeile eines aufgespaltenen Wertes und einen Punkt am Anfang der nächsten Zeile symbolisiert. Die Spalten bleiben aber mindestens so breit wie der Spalten-Name.

- Wenn viel zu breit: `\pset expanded on`

```
* Record 1
sid      101
vorname  Lisa
nachname Weiss
email    weiss@acm.org
* Record 2
...      ...
```

Ausgabe-Formatierung (4)

- `\pset format html` liefert Ausgaben im HTML-Format.

  Damit könnte man Anfrage-Ergebnisse in einfache Webseiten einbauen, ggf. sogar mit dem Skript die ganze Webseite generieren.

- `\pset format latex` liefert Ausgaben im LaTeX-Format.

  Damit kann man die Tabelle z.B. leicht in Abschlussarbeiten einbauen.

- `\pset footer off` stellt "`(`*n* `rows)`" am Ende ab.

  Dagegen würde `tuples_only` zusätzlich auch den Kopf entfernen.

- Man kann natürlich auch SQL-Anfragen schreiben, die LaTeX oder HTML generieren.

  Dazu würde man Strings mit `||` konkatenieren und ggf. mehrere Teile mit `UNION ALL` zusammenfügen. Spalten für die Ausgabe-Sortierung müssen beim `UNION` noch vorhanden sein, aber man kann es in eine Unteranfrage/`WITH` stecken, und dann nach einer Spalte sortieren, die nicht im `SELECT` auftaucht.

Ausgabe-Formatierung (5)

- Angenommen, man hat eine SQL-Anfrage, die nur eine Ausgabespalte hat, und man möchte genau die erzeugten Daten in eine Datei schreiben:

```
\pset tuples_only on
\pset format unaligned
\o ⟨File⟩
SELECT ...;
-- Bei Bedarf alles zurückschalten:
\o
\pset format aligned
\pset tuples_only off
```

`unaligned` ist auch bei einer Spalte wichtig: Sonst werden Zeilenumbrüche in den Daten mit "+" markiert (und Leerzeichen vor/nach jedem Wert gedruckt).

Für einige häufige Optionen gibt es auch kurze Spezialbefehle, z.B. schaltet `\t` die Option `tuples_only` immer um zwischen `on` und `off`.

Literatur/Quellen

- Wikipedia: PostgreSQL (englisch, deutsch)
  [https://en.wikipedia.org/wiki/PostgreSQL]
  [https://de.wikipedia.org/wiki/PostgreSQL]
- Homepage des Projekts
  [https://www.postgresql.org/]
- Deutsche PostgreSQL Homepage
  [http://www.postgres.de/]
- PostgreSQL Dokumentation: `psql`
  [https://www.postgresql.org/docs/9.5/app-psql.html]
- PostgreSQL Tutorial:
  [http://www.postgresqltutorial.com/]
  PSQL Commands: [http://www.postgresqltutorial.com/psql-commands/]
- pgAdmin (GUI zur Administration, auch SQL-Zugang)
  [https://www.pgadmin.org/]
- PostgreSQL Documentation 11: Chapter 37: The Information Schema
  [https://www.postgresql.org/docs/current/information-schema.html]