Datenbanken II A: DB-Entwurf

Chapter 8: Gespeicherte Prozeduren

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2020/21

http://www.informatik.uni-halle.de/~brass/dd20/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Sie sollten einige Vorteile von im Server gespeicherten und ausgeführten Prozeduren und Funktionen nennen können.

  "Stored procedures". Damit sollten Sie auch für ein konkretes Projekt entscheiden können, ob server-seitige Prozeduren dort verwendet werden sollten.

- Sie sollten einfache Funktionen für PostgreSQL in den Sprachen SQL und PL/pgSQL erstellen können.
- Sie sollten auch Tabellenfunktionen in Anfragen verwenden können.

Inhalt

Einleitung (1)

- Bei allen großen DBMS ist es möglich, Prozeduren zu definieren, die in der Datenbank gespeichert werden, und vom Server ausgeführt werden ("Stored Procedures").
- Man kann auch Programmcode definieren, der bei bestimmten Ereignissen (z.B. Einfügung eines Tupels) automatisch ausgeführt wird ("Trigger").
- Jedes System hat dafür eine eigene Programmiersprache (manche auch mehrere zur Auswahl): Die Sprachen bieten u.a. die üblichen Kontrollstrukturen (`if`, `while`, etc.), Variablen, und natürlich SQL-Anweisungen.
- Es gibt einen Standard, SQL/PSM ("Persistant Stored Modules") seit 1996. Der Standard ist recht ähnlich zu Oracle's PL/SQL, das wiederum von ADA inspiriert war.

Einleitung (2)

- Sprachen verschiedener Systeme sind:

DBMS	Sprache(n)
Oracle	PL/SQL, Java
PostgreSQL	PL/pgSQL, PL/Perl, PL/Python
MS SQL Server	Transact-SQL
MySQL	SQL/PSM
DB2	SQL PL, Java

- Wenn die Sprachen größtenteils auch ziemlich ähnlich sind, erhöht der Einsatz serverseitiger Programmierung sicherlich den Portierungsaufwand bei einem Wechsel zu einem anderen DBMS.

- Während SQL deklarativ ist, sind diese Sprachen imperativ.

Einleitung (3)

Vorteile von "Stored Procedures":

- Sicherung komplexer Integritätsbedingungen:
  - Wie man in der objektorientierten Programmierung üblicherweise die Attribute einer Klasse nur über Methoden der Klasse ändern kann,
  - kann man hier die Zugriffsrechte so vergeben, dass Updates von Tabellen nur über diese Prozeduren möglich sind.

    Dagegen würde man Lese-Zugriffe auf die Tabellen weiter normal mit SQL-Anfragen erlauben.
- Das SQL-Interface der Programmiersprache ist deutlich besser als bei unabhängig entwickelten Programmiersprachen, z.B. werden die SQL-Typen auch für Variablen verwendet.

  Die DB-Programmierung ist daher einfacher als in Java mit JDBC.

Einleitung (4)

Vorteile von "Stored Procedures", Forts.:

- Wenn man komplexe Berechnungen auf dem Server ausführt, spart man die Netzwerk-Kommunikation mit dem Client.

  Es müssen nicht alle Daten, die für die Berechnung notwendig sind, über das Netz geschickt werden, sondern nur das Ergebnis. Häufig ist der Server auch mit RAM und CPU Cores gut ausgestattet, während die Clients sehr unterschiedlich und teilweise eher schwach sind.

- Die serverseitigen Prozeduren mit ihren Abhängigkeiten werden im Systemkatalog ("Data Dictionary") nachgewiesen.

  Dagegen weiss das DBMS nichts von den Anwendungsprogrammen.
  Oracle hat Tabelle `ALL_DEPENDENCIES`, in PostgreSQL gibt es nur den Quellcode.

- Die serverseitigen Prozeduren und ggf. die darin enthalten SQL-Anweisungen können im Server in vorcompilierter (und optimierter) Form gespeichert werden.

Einleitung (5)

Definition neuer Datentypen/Erweiterungen der Sprache SQL:

- Benutzerdefinierte Funktionen können auch in SQL-Anfragen genutzt werden.
- Damit kann man also die Sprache SQL erweitern.
- Dies ist insbesondere für moderne objektrelationale Systeme wichtig. Hier kann man auch eigene Datentypen definieren.

  Zu Datentypen gehören immer auch Operationen.
- Mit Triggern kann man die Semantik von Updates verändern.

  Z.B. auch programmierbare, komplexe Änderungsrechte (es wird ein Rollback ausgelöst, wenn die Bedingung für den Update nicht erfüllt ist).
  Die Trigger sind soetwas wie "Hooks", an denen man eigenen Code in das System einklinken kann.

Einleitung (6)

Natürliche Entwicklung:

- Ein Ziel einer Datenbank ist die zentrale Speicherung aller Daten einer Firma:
  - um Redundanzen und Inkonsistenzen zu vermeiden,
  - um Daten zwischen verschiedenen Anwendungen zu teilen,
  - um die Entwicklung neuer Anwendungen zu vereinfachen,
  - um die Administration zu vereinfachen,
  - um Kosten zu senken.
- Nun möchte man auch Prozeduren zentral speichern.

  Auf dem Weg von einer Daten- zur einer Wissensbank. Dafür wären aber deklarative Programmiersprachen, z.B. Datalog aus der logischen Programmierung, eine interessante Alternative.

Einleitung (7)

Vergleich mit Unterprogramm-Bibliothek:

- Eine Alternative wäre, Java-Klassen mit JDBC-Aufrufen zu erstellen, die von verschiedenen Anwendungsprogrammen genutzt werden können.
- Dieser Programmcode existiert nur außerhalb der Datenbank, deswegen kann er nicht in SQL-Anweisungen und Triggern benutzt werden.
- Diese Klassen und ihre Abhängigkeiten von DB-Tabellen sind nicht im Data Dictionary vermerkt.

  Wenn das DB-Schema geändert wird, werden eventuelle Probleme erst bemerkt, wenn die kritischen SQL-Anweisungen tatsächlich ausgeführt werden (ggf. also viel später).

Beispiel-Datenbank

STUDENTEN

SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	···
102	Michael	Grau	NULL
103	Daniel	Sommer	···
104	Iris	Winter	···

BEWERTUNGEN

SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
102	H	2	9
102	Z	1	10
103	H	1	5
103	Z	1	7

AUFGABEN

ATYP	ANR	THEMA	MAXPT
H	1	ER	10
H	2	SQL	10
Z	1	SQL	14

Inhalt

CREATE FUNCTION: Einführung (1)

- In PostgreSQL deklariert man Funktionen, die in SQL-Anfragen und Updates aufgerufen werden können.

  [https://www.postgresql.org/docs/9.5/sql-createfunction.html]

  `CREATE PROCEDURE` gibt es in PostgreSQL erst ab Version 11.

  [https://www.postgresql.org/docs/11/sql-createprocedure.html]

- Beispiel:

```
CREATE FUNCTION inc(n INTEGER) RETURNS INTEGER
       AS
       'SELECT n+1;'
       LANGUAGE SQL;
```

- Aufrufe der Funktion sind in jedem Term in SQL-Anweisungen (Anfragen oder Updates) möglich, am einfachsten so:

```
SELECT inc(3);
```

CREATE FUNCTION: Einführung (2)

- In PostgreSQL, user-defined functions can be written in different languages:
  - SQL [https://www.postgresql.org/docs/9.5/xfunc-sql.html]
  - C [https://www.postgresql.org/docs/9.5/xfunc-c.html]
  - PL/pgSQL [https://www.postgresql.org/docs/9.5/plpgsql.html]
  - PL/Python [https://www.postgresql.org/docs/9.5/plpython.html]
  - PL/Perl [https://www.postgresql.org/docs/9.5/plperl.html]
  - PL/Tcl [https://www.postgresql.org/docs/9.5/pltcl.html]
- Es gibt auch noch weitere Sprachen, die nicht zur Kern-Distribution gehören.

  U.a. Java. [https://www.postgresql.org/docs/current/external-pl.html]

CREATE FUNCTION: Einführung (3)

- Der Rumpf der Funktion, also der eigentliche Programmcode, wird als String-Konstante definiert.
- Das kann ein Problem sein, wenn der Programmcode selbst String-Konstanten enthält (`'` muss verdoppelt werden).
- PostgreSQL hat eine spezielle Syntax für String-Konstanten (nicht standard-konform, aber für diesen Zweck praktisch):

  $\$\langle Tag \rangle\$$ ... $\$\langle Tag \rangle\$$

- Das Tag kann leer sein, z.B. wäre `$$abc$$` der String `'abc'`.

  Das "Tag" ist nur wichtig, wenn man solche Konstanten schachteln muss.

  Z.B. wird ein mit `$body$` geöffneter String nur mit `$body$` wieder geschlossen.

- String-Konstanten in SQL (egal ob `'...'` oder `$$...$$`) können Zeilenumbrüche enthalten.

CREATE FUNCTION: Einführung (4)

- In PostgreSQL werden Funktionsdeklarationen typischerweise mit der $$-Notation für den Rumpf geschrieben:

```
CREATE FUNCTION inc(n INTEGER) RETURNS INTEGER
       AS $$
       SELECT n+1;
       $$
       LANGUAGE SQL;
```

- Der Rumpf einer Funktion in der Sprache SQL ist eine Folge von SQL-Anweisungen (außer `COMMIT` u.ä.).

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

- Der Rückgabewert der Funktion ist das Ergebnis der letzten SQL-Anweisung (bzw. die erste Zeile des Ergebnisses).

  Man kann auch mengenwertige Funktionen deklarieren, s.u.

CREATE FUNCTION: Einführung (5)

- Der Rumpf wird vollständig geparst, bevor er ausgeführt wird. Man kann `CREATE TABLE` Anweisungen in den Rumpf schreiben, aber die Tabellen dort noch nicht verwenden.
- Parameter kann man nur einsetzen, wo Datenwerte (Terme) erwartet werden, nicht für Tabellen- oder Spalten-Namen.
- Die Reihenfolge von `AS` ⟨Body⟩ und der Angabe der Sprache sowie vielen weiteren Optionen (s.u.) ist beliebig:

```
CREATE FUNCTION inc(n INTEGER) RETURNS INTEGER
       LANGUAGE SQL
       AS $$
       SELECT n+1;
       $$;
```

[https://www.postgresql.org/docs/current/sql-createfunction.html]

CREATE FUNCTION: Einführung (6)

- Der Rumpf der Funktion kann auch Updates enthalten:

```
CREATE FUNCTION abgabeschluss(a INTEGER)
       RETURNS VOID
AS $$
       INSERT INTO BEWERTUNGEN(SID, ATYP, ANR,
                               PUNKTE)
       SELECT SID, 'H', a, 0 as PUNKTE
       FROM STUDENTEN
       WHERE SID NOT IN (SELECT SID
                         FROM   BEWERTUNGEN
                         WHERE  ATYP = 'H'
                         AND    ANR  = a)
$$ LANGUAGE SQL;
```

Wer die Hausaufgabe a nicht abgegeben hat, bekommt 0 Punkte.

CREATE FUNCTION: Einführung (7)

- Die Parameter-Namen haben die gleiche Syntax wie Bezeichner in SQL (z.B. Spalten-Namen).

  Wenn man keinen Parameter-Namen angibt (nur den Datentyp), heißen die Parameter `$1`, `$2`, u.s.w. In diesem Fall gibt es keine Namenskonflikte.

- Hätte man den Parameter "`anr`" genannt, wäre in der Unteranfrage nicht klar, ob der Parameter oder die Spalte von `BEWERTUNGEN` gemeint ist.

- Bei der Sprache "SQL" für die Funktions-Implementierung gewinnt in diesem Fall der Spalten-Name.

  Bei PL/pgSQL ist es konfigurierbar, Default ist dort eine Fehlermeldung.
  [https://www.postgresql.org/docs/current/plpgsql-implementation.html]

- Notfalls schreibe man: "⟨Funktion⟩ . ⟨Parameter⟩".

  Möglicher Stil: Präfix "`p_`" für alle Parameter, und nicht in Spaltennamen.

CREATE FUNCTION: Einführung (8)

- Wenn die Funktion schon deklariert ist, führt eine Neudeklaration zu einer Fehlermeldung.

  Wenn man die Funktionsdeklaration in eine Datei geschrieben hat, die man mit `\i` in `psql` einliest, wird es bei Änderungen umständlich.

- Man kann aber festlegen, dass die Funktionsdeklaration in diesem Fall überschrieben wird:

  ```
  CREATE OR REPLACE FUNCTION ...
  ```

  Mit dem Risiko, dass man eventuell unabsichtlich eine Funktion überschreibt.

- In PostgreSQL können Funktionen überladen werden. Wenn man Anzahl oder Typen der Parameter ändert, definiert man eine zusätzliche Funktion!

- Das Löschen einer Funktion geht nur mit Parameter-Typen:

  ```
  DROP FUNCTION abgabeschluss(INTEGER);
  ```

Funktionen im Systemkatalog

- In der Kommandoschnittstelle `psql` kann man sich Funktionen u.a. mit folgenden Kommandos listen lassen:
  - `\df`: Eigene Funktionen (kurze Ausgabe)
  - `\df+`: Eigene Funktionen (lange Ausgabe)
  - `\dfS+`: System-Funktionen (lange Ausgabe)
- Funktionen sind in der Tabelle `pg_proc` eingetragen (im Schema `pg_catalog`).

  [https://www.postgresql.org/docs/9.2/catalog-pg-proc.html]

  Die Spalte `proname` ist der Name der Funktion, `prosrc` der Quellcode.

  Die Spalte `prorettype` ist der Ergebnistyp, allerdings als OID des Eintrags in `pg_type` (darin steht dann `typname`). Den Wert in der Spalte `proargtypes` kann man decodieren mit `oidvectortypes(proargtypes)`.

Parameter-Typen, Typ-Umwandlungen (1)

- Beispiel-Anfrage (leider ziemlich sinnlos):

```
SELECT inc(CAST(SID AS INTEGER)), Punkte
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND ANR = inc(1)
```

- Die Funktion `inc` ist mit Parameter-Typ `INTEGER` deklariert.
- Die Spalte `SID` ist als `NUMERIC(3)` deklariert.
  Dieser Typ wird nicht automatisch nach `INTEGER` konvertiert.
  Ohne `CAST` gibt es eine Fehlermeldung.

  "`No function matches the given name and argument types.`"
  Vermutlich ist die Ursache, dass der Typ in PostgreSQL einfach "`NUMERIC`" ist (Festkommazahl mit Nachkommastellen) und der "Type Modifier" `(3)` wie eine Integritätsbedingung behandelt wird. Deswegen können bei der Typ-Konvertierung ganzzahlige `NUMERIC`-Typen nicht von solchen mit Nachkommastellen unterschieden werden.

Parameter-Typen, Typ-Umwandlungen (2)

- Automatische Typ-Umwandlungen (Casts) kann man mit folgender Abfrage auflisten lassen:

```
SELECT s.typname SOURCE, t.typname TARGET
FROM   pg_cast c, pg_type s, pg_type t
WHERE  c.castsource = s.oid
AND    c.casttarget = t.oid
AND    c.castcontext = 'i' -- implicit
```

  Statt des Joins mit `pg_type` kann man z.B. auch schreiben:
  `c.castsource = 'NUMERIC'::regtype` (nutzt eine spezielle Typ-Umwandlung in den OID-Typ, `::` ist nur eine PostgreSQL-Abkürzung für den `CAST`).
  [https://www.postgresql.org/docs/current/catalog-pg-cast.html]
  [https://www.postgresql.org/docs/current/catalog-pg-type.html]
  [https://www.postgresql.org/docs/current/datatype-oid.html]

- In `psql` kann man z.B. "`\dC NUMERIC`" benutzen.

Parameter-Typen, Typ-Umwandlungen (3)

- Man kann `inc` mit `NUMERIC`-Argument deklarieren:

```
CREATE FUNCTION inc(n NUMERIC(3)) -- schlecht
       RETURNS NUMERIC(3)         -- s.u.
       AS $$
       SELECT n+1;
       $$ LANGUAGE SQL;
```

- Tatsächlich wird der Zusatz "(3)" hier einfach ignoriert, z.B.
  - `SELECT inc(3.5)` $\longrightarrow$ `4.5`
  - `SELECT inc(1000)` $\longrightarrow$ `1001`
- Man sollte hier also nur `NUMERIC` schreiben (ohne genauere Angabe), um keine falschen Erwartungen zu erwecken.

Parameter-Typen, Typ-Umwandlungen (4)

- Dagegen wird `INTEGER` automatisch in `NUMERIC` umgewandelt.

  Das macht Sinn: Jeder `INTEGER`-Wert lässt sich in dem uneingeschränkten Festkomma-Zahltyp (`NUMERIC`) darstellen.

- Gleiches gilt für `VARCHAR`: Als Parameter-Typ kann man es auch nur ohne die Längen-Angabe verwenden.

  Man kann die Maximallänge zwar hinschreiben, aber sie wird einfach ignoriert.

- Konstanten in SQL ohne Nachkommastellen sind vom Typ `INTEGER`, mit Nachkommastellen vom Typ `NUMERIC`.

  Bei sehr großen Konstanten wählt PostgreSQL `BIGINT`, dann `NUMERIC`.

  Der Standard verlangt nur einen "implementation-defined exact NUMERIC type".

  In PostgreSQL zeigt `pg_typeof(`*e*`)` den Typ des Ausdrucks *e*.

- String-Konstanten haben in PostgreSQL den Typ `unknown`, was besonders großzügige Konvertierungen erlaubt.

Nullwerte

- Funktionen können auch für Nullwerte aufgerufen werden:

```
SELECT inc(NULL);
```

  Wenn es mehrere überladene Varianten von `inc` geben sollte, muss man über `CAST(NULL AS` T`)` den Typ explizit angeben.

- Man kann Funktionen als `STRICT` deklarieren, dann werden sie nicht aufgerufen, falls eins der Argumente `NULL` ist. In diesem Fall ist das Ergebnis automatisch `NULL`.

  Ausführlich heißt die Option "`RETURNS NULL ON NULL INPUT`".

```
CREATE FUNCTION inc(n INTEGER) RETURNS INTEGER
       LANGUAGE SQL
       AS $$
       SELECT n+1;
       $$ STRICT;
```

  Die Reihenfolge von `AS $$...$$` und `LANGUAGE SQL`, `STRICT` u.s.w. ist egal.

Rechte, Sicherheit

- Normalerweise werden Funktionen mit den Rechten des Nutzers ausgeführt, der sie aufgerufen hat.

  Das erfordert Vertrauen, dass die Funktion auch das tut, was man denkt.

- Im `CREATE FUNCTION`-Befehl kann man aber angeben:

  `SECURITY DEFINER`

  Das ist eine Option wie `LANGUAGE SQL`. Alle Optionen werden hintereinander (nur durch Leerzeichen getrennt) angegeben. Default: `SECURITY INVOKER`.

- Dann werden die Befehle im Funktionsrumpf mit den Rechten des Nutzers ausgeführt, der die Funktion definiert hat.

- Man sollte sicherstellen, dass niemand einem Funktionen über den Suchpfad "unterschieben" kann.

  Da es überladene Funktionen gibt, könnte das eine typmäßig besser passende Funktion sein, als die, die man eigentlich aufrufen wollte.

Seiteneffekte (1)

- Wenn man Funktionen unter `SELECT` aufruft, ist klar definiert, welche Funktionsaufrufe stattfinden:

```
SELECT abgabeschluss(ANR)
FROM   AUFGABEN
WHERE  ATYP = 'H'
```

- Es ist aber auch möglich, dass eine Funktion Updates enthält, und einen Funktionswert liefert (d.h. Ergebnistyp nicht `VOID`).
- Dann kann man die Funktion auch unter `WHERE` aufrufen.
- Bei mehreren mit `AND` verknüpften Bedingungen entscheidet aber der Optimierer über die Auswertungs-Reihenfolge (und wertet meist nur aus, was nötig ist).
- Dann ist undefiniert, welche Funktionsaufrufe stattfinden.

Seiteneffekte (2)

- Deshalb sollten Funktionen mit Seiteneffekten (die also den Datenbank-Zustand verändern) ausschließlich unter `SELECT` benutzt werden.
- Oracle trennt klar Funktionen und Prozeduren:
  - Funktionen können in Anfragen benutzt werden und haben keine Seiteneffekte.
  - Prozeduren müssen explizit aufgerufen werden. Sie können nicht in Anfragen benutzt werden.

    In SQL*Plus gibt es dafür den `EXECUTE` Befehl. Man kann auch einen anonymen PL/SQL-Block schreiben (ein Stück Programmcode). Dort ist es ein normaler Methodenaufruf (wie in Java).
- In PostgreSQL gibt es nicht einmal eine Warnung.

Hinweise für den Optimierer

- Man kann dem Optimierer mitteilen, ob mehrfache Aufrufe einer Funktion wegoptimiert werden dürfen:
  - `IMMUTABLE`: Funktionsaufruf für Konstanten darf schon bei Compilierung ausgewertet werden, Ergebnis ändert sich nie.

    Die Funktion `inc` fällt in diese Klasse, `inc(1)` wird immer 2 sein.

    Solche Funktionen dürfen nicht einmal lesend auf die DB zugreifen.
  - `STABLE`: Mehrfache Aufrufe mit den gleichen Argumenten in einer Anfrage sind unnötig. Der DB-Zustand wird nicht geändert.

    Solche Funktionen dürfen nur lesend auf den DB-Zustand zugreifen.
  - `VOLATILE`: Beliebige DB-Zugriffe. Dies ist der Default.

    [https://www.postgresql.org/docs/current/xfunc-volatility.html]
- Auch Angaben zur Kostenschätzung sind möglich: `COST`, `ROWS`.

Transaktionen

- Die SQL-Befehle im Rumpf einer Funktion werden als Teil der Transaktion ausgeführt, in der die aufrufende `SELECT`-Anfrage läuft.

  Das erklärt, warum man kein `COMMIT` in eine Funktion schreiben kann.

- Wenn man den Autocommit-Modus nicht mit `BEGIN TRANSACTION` ausgeschaltet hat, ist jede Top-Level Anfrage eine eigene Transaktion.

- Wenn man aber z.B. zwei `INSERT`-Anweisungen im Rumpf einer Funktion hat, und die zweite verursacht einen Fehler (z.B. Verletzung eines Schlüssels), wird auch die erste zurückgenommen.

  Es gilt hier also auch "ganz oder garnicht".

Tabellen-Funktionen (1)

- Tabellen-Funktionen sind Funktionen, die eine Tabelle als Funktionswert liefern. Sie können unter `FROM` benutzt werden:

```
select * from generate_series(1,3);
```

generate_series
1
2
3

[https://www.postgresql.org/docs/current/functions-srf.html]

- Tabellen-Funktionen sind recht ähnlich zu Sichten, aber man übergibt hier als Argumente, was sonst in einer Selektions- oder Join-Bedingung außen stehen würde.

  Die Syntax ist etwas kürzer und vermutlich für Programmierer ansprechender.

  Es gibt aber für beide Konstrukte etwas, was mit dem anderen nicht geht (s.u.).

Tabellen-Funktionen (2)

- Im Prinzip entspricht die Tabellenfunktion einer Sicht:

  ```
  GENERATE_SERIES_VIEW(START, STOP, VALUE)
  ```

  `START` und `STOP` sind die Eingabe-Argumente, `VALUE` die Ausgabe.

- Beispiel-Abfrage (geht so nicht):

  ```
  SELECT VALUE
  FROM   GENERATE_SERIES_VIEW
  WHERE  START = 1 AND STOP = 3
  ```

- Im Beispiel geht das nicht, weil die Sicht unendlich groß wäre, und erst durch die Selektion der Grenzen auf einen endlichen Bereich eingeschränkt wird.

- In SQL kann man jede Sicht vollständig auflisten:

  ```
  SELECT * FROM GENERATE_SERIES_VIEW
  ```

Tabellen-Funktionen (3)

- Im Beispiel ist es die Schwäche der Sicht, dass man Spalten wie `START` und `STOP` nicht als "Eingabespalten" festlegen kann, für die ein Wert beim Aufruf bekannt sein muss.
- Es ist sonst aber gerade die Stärke des relationalen Modells, dass man jede Spalte als Eingabe (mit einer Bedingung unter `WHERE`) oder Ausgabe (unter `SELECT`) benutzen kann.

  Man hat gewissermaßen mit jeder Funktion auch die Umkehrfunktion definiert, im relationalen Modell aber leider nur für endliche Funktionen.
- Deduktive Datenbanken (aus der logischen Programmierung) vereinheitlichen und verknüpfen die Vorteile beider Ansätze.

  In der auf Prolog basierenden Sprache Datalog wird (wie im Bereichskalkül) die Notation "Prädikat(Argumente)" verwendet, und Argumente können Konstanten oder Variablen sein. Prädikat-Extensionen können unendlich sein.

Tabellen-Funktionen (4)

- Man kann Tabellen-Funktionen auch selbst schreiben:

```
CREATE FUNCTION GELOEST(SID NUMERIC)
       RETURNS TABLE(ANR INTEGER)
       AS $$
           SELECT CAST(B.ANR AS INTEGER)
           FROM   BEWERTUNGEN B
           WHERE  B.ATYP = 'H'
           AND    B.SID = GELOEST.SID
       $$
       LANGUAGE SQL
       STRICT
       STABLE;
```

Liefert die Nummern der Hausaufgaben, die ein Student mit gegebener `SID` gelöst hat. Da der Parameter hier heißt wie eine Spalte, werden mit der Tupelvariable und dem Funktionsnamen die Referenzen eindeutig gemacht.

Tabellen-Funktionen (5)

- Aufruf der Tabellen-Funktion z.B.:

```
SELECT A.ANR, A.THEMA
FROM   GELOEST(103) G, AUFGABEN A
WHERE  G.ANR = A.ANR AND A.ATYP = 'H'
```

- Seit PostgreSQL 9.3 gibt es "Lateral Joins", die es erlauben, Spalten von Tabellen weiter links unter `FROM` in Unteranfragen oder Funktionsaufrufen weiter rechts zu verwenden:

```
SELECT G.ANR
FROM   STUDENTEN S, LATERAL GELOEST(S.SID)
WHERE  S.NACHNAME = 'Weiss'
```

[https://www.postgresql.org/docs/current/queries-table-expressions.html]

Ohne das Schlüsselwort `LATERAL` gibt es den Fehler "function expression in FROM cannot refer to other relations of same query level".

Mehr zu Datentypen (1)

- Den Typ einer existierenden Tabellenspalte kann man ansprechen mit ⟨Tabelle⟩.⟨Spalte⟩`%TYPE`.
- Es gibt in PostgreSQL auch
  - Aufzählungs-Typen ("Enumerations"),
  - Intervalle,
  - Record-Typen (Zeilen, "composite types"),
  - Arrays.

    [https://www.postgresql.org/docs/current/datatype.html]
    [https://www.postgresql.org/docs/current/extend-type-system.html]
- Man kann eigene Typen mit `CREATE TYPE` definieren.

  [https://www.postgresql.org/docs/current/sql-createtype.html]
  [https://www.postgresql.org/docs/9.5/sql-createdomain.html]

Mehr zu Datentypen (2)

- Tabellen-Funktionen können auch mit SETOF definiert werden:

```
CREATE FUNCTION GELOEST(SID NUMERIC)
       RETURNS SETOF INTEGER
```

- Für jede Tabelle ist automatisch ein Typ mit gleichem Namen für die Tabellenzeilen definiert. Z.B. Selektionsfunktion:

```
CREATE FUNCTION GUT(PCT INTEGER)
       RETURNS SETOF BEWERTUNGEN
       AS $$ SELECT *
             FROM   BEWERTUNGEN B
             WHERE  B.PUNKTE >=
                    (SELECT (A.MAXPT * PCT)/100
                     FROM   AUFGABEN A
                     WHERE  A.ATYP = B.ATYP
                     AND    A.ANR  = B.ANR)
       $$ LANGUAGE SQL STRICT STABLE;
```

Mehr zu Datentypen (3)

- Beispiel (Bewertungen mit mindestens 80% der Punkte):

```
SELECT * FROM GUT(80)
```

gut
(101,H,1,10.0)
(101,H,2,8.0)
(101,Z,1,12.0)
(102,H,1,9.0)
(102,H,2,9.0)

- Es sind hier also Werte des Zeilen-Typs (Record).

  [https://www.postgresql.org/docs/9.4/rowtypes.html]

- Mit folgender Anfrage bekommt man die Spalten einzeln:

```
SELECT X.* FROM GUT(80) X;
```

  Natürlich kann man auch auf einzelne Spalten zugreifen, z.B. mit `X.SID`.

Mehr zu Parameter-Listen (1)

- Man kann Parameter als `IN`, `OUT`, `INOUT` oder `VARIADIC` deklarieren ("Parameter Modes"). `IN` ist der Default.

  `VARIADIC` kann beliebig viele Eingabeparameter von einem Typ als Array akzeptieren. [https://www.postgresql.org/docs/9.5/xfunc-sql.html]

- Wenn es nur einen Ausgabe-Parameter gibt, ist dessen Typ der Rückgabe-Typ der Funktion.

- Bei mehreren Ausgabe-Parametern ist der Record-Typ mit diesen Parametern der Rückgabe-Typ der Funktion.

  Wenn man Ausgabe-Parameter hat, sollte man die `RETURNS`-Klausel weglassen. Man darf sie hinschreiben, aber sie muss dann zum durch die Ausgabe-Parameter festgelegten Typ passen.

Mehr zu Parameter-Listen (2)

- Man kann Default-Werte für Eingabe-Parameter angeben.
- Beispiel:

```
CREATE FUNCTION ANZ_GUT(IN PCT INTEGER = 90,
                        OUT ANZ BIGINT)
        AS $$ SELECT COUNT(DISTINCT B.SID) AS ANZ
              FROM   BEWERTUNGEN B
              WHERE  B.PUNKTE >=
                     (SELECT (A.MAXPT * PCT)/100
                      FROM   AUFGABEN A
                      WHERE  A.ATYP = B.ATYP
                      AND    A.ANR  = B.ANR)
        $$ LANGUAGE SQL STRICT STABLE;
```

Statt "=" kann man auch das Schlüsselwort "DEFAULT" schreiben.

- Beispiel-Aufruf: `SELECT ANZ_GUT();` Ergebnis: `BIGINT`-Wert.

Funktions-Aufrufe

- Bei Funktionsaufrufen kann man nur Werte für die `IN`- und `INOUT`-Parameter angeben.

  Die `OUT`-Parameter werden nur für die Berechnung des Rückgabetyps verwendet. Sie zählen nicht mit zur Signatur der Funktion. Auch in PL/pgSQL kann man keine Variablen für `OUT`-Parameter übergeben. Das ist anders als in Oracle's PL/SQL, dort muss man Variablen für `INOUT`- und `OUT`-Parameter angeben.

- Sei eine Funktion mit zwei Parametern gegeben:

  ```
  CREATE FUNCTION F(A INTEGER, B INTEGER) ...
  ```

  - Aufruf in der üblichen "positional notation": `F(1,2)`
  - Aufruf in der "named notation": `F(b => 2, a => 1)`

    [https://www.postgresql.org/docs/9.5/sql-syntax-calling-funcs.html]

    Vor PostgreSQL 9.5 musste man ":=" statt "=>" schreiben.

    Das wird aus Gründen der Abwärtskompatibilität auch noch akzeptiert.

    In Oracle's PL/SQL war es aber schon immer "=>".

Inhalt

PL/pgSQL: Einführung

- PL/pgSQL ("SQL Procedural Language") ist eine imperative Programmiersprache mit den üblichen Kontrollstrukturen, die besonders eng mit SQL verknüpft ist:
  - Gleiches Typsystem
  - Variablen können Nullwerte annehmen
  - Wertausdrücke (Terme) wie in SQL-Anfragen
  - `IF`-Anweisungen mit SQL-Bedingungen (wie unter `WHERE`)
  - Spezielle Schleifen für Verarbeitung von Anfrage-Ergebnissen
- Nur eine von mehreren ladbaren Sprachen.

  [https://www.postgresql.org/docs/12/server-programming.html]

  [https://www.postgresql.org/docs/12/plpgsql.html]

Beispiel: Fibonacci-Funktion (1)

```
 (1)  CREATE OR REPLACE FUNCTION
 (2)         fib(n INTEGER) RETURNS BIGINT
 (3)     LANGUAGE plpgsql
 (4)     STRICT
 (5)     AS $$
 (6)     BEGIN
 (7)         IF n < 0 THEN
 (8)             RAISE NOTICE 'Invalid argument';
 (9)             RETURN 0;
(10)         ELSIF n = 0 OR n = 1 THEN
(11)             RETURN 1;
(12)         ELSE
(13)             RETURN fib(n-1) + fib(n-2);
(14)         END IF;
(15)     END
(16)     $$;
```

Beispiel: Fibonacci-Funktion (2)

Anmerkungen:

- Groß-/Kleinschreibung ist wie in SQL egal.

  Es wäre wahrscheinlich geschickt, sich in PostgreSQL an die Kleinschreibung zu gewöhnen (intern werden alle Bezeichner in Kleinschreibung abgebildet). In Oracle umgekehrt.

- Das Schlüsselwort "`STRICT`" bedeutet, dass es für einen Nullwert als Eingabe auch einen Nullwert als Ausgabe gibt (ohne dass der Rumpf ausgeführt wird).

  Ansonsten bekommt man bei "`select fib(NULL);`" einen Stack-Overflow aufgrund von Endlos-Rekursion: "ERROR: stack depth limit exceeded".
  Man kann natürlich mit "`IF n IS NULL`" den Nullwert auch explizit behandeln.

- Alle solchen Optionen für Funktionen können auch am Ende (nach dem Rumpf) angegeben werden.

Beispiel: Geld vom Konto abheben (1)

- Es sei eine Datenbank einer (sehr kleinen) Bank betrachtet.

KONTO		
NR	STAND	INHABER
101	20.00	Lisa Weiss
102	150.00	Iris Winter

PROTOKOLL			
ID	ZEIT	NR	BETRAG
1	2020-06-25 14:00:00.000000	101	-10.00

Die Tabelle `PROTOKOLL` dient der Generierung von Kontoauszügen.

`ID` ist vom Typ `SERIAL` (`INTEGER` mit `DEFAULT nextval('Sequenz')`).

- Es soll eine Prozedur für Barauszahlungen geschrieben werden, die Kontoüberziehungen vermeidet.

Beispiel: Geld vom Konto abheben (2)

```
(1)  CREATE OR REPLACE FUNCTION
(2)     Abhebung(n INTEGER, b NUMERIC) RETURNS VOID
(3)     -- n: Kontonummer, b: Betrag
(4)     AS $$
(5)     DECLARE
(6)        s KONTO.STAND%TYPE;
(7)        -- Variable vom gleichen Typ
(8)        -- wie Spalte STAND in Tabelle KONTO
(9)     BEGIN
(10)       SELECT STAND INTO s FROM KONTO
(11)          WHERE NR = n FOR UPDATE;
(12)       IF s < b THEN
(13)          RAISE NOTICE 'Stand zu niedrig';
(14)       -- Fortsetzung auf naechster Folie
```

Beispiel: Geld vom Konto abheben (3)

```
(15)          ELSE -- Kontostand ausreichend
(16)              s := s - b;
(17)              UPDATE KONTO SET STAND = s
(18)                  WHERE NR = n;
(19)              INSERT INTO PROTOKOLL
(20)                      (ZEIT, NR, BETRAG)
(21)                  VALUES
(22)                      (CURRENT_TIMESTAMP, n, -b);
(23)          END IF;
(24)      END
(25)      $$ LANGUAGE PLPGSQL;
```

Beispiel: Geld vom Konto abheben (4)

Anmerkungen:

- Manche Fehler in SQL-Anfragen (wie nicht existierende Tabellen) werden erst bei Ausführung der SQL-Anfrage gemeldet, nicht beim `CREATE FUNCTION`.

  Ich halte das für eine schlechte Lösung. Es erfordert gründlicheres Testen. Bei Ausführung der Funktion wird die Optimierung von Anfragen gespart, die in einem nicht betretenen `IF`-Zweig stehen.

- Im Beispiel sollte eine Zuweisung an eine Variable gezeigt werden. Sonst hätte man das Update auch

  ```
  UPDATE KONTO SET STAND = STAND - b WHERE NR = n;
  ```

  schreiben können.

  Das "`SELECT ... FOR UPDATE`" wäre trotzdem nötig, sonst könnte der Kontostand negativ werden, wenn sich eine andere Transaktion zwischen den Test und die Verringerung schiebt.

Blöcke

- PL/pgSQL Programmcode steht in Blöcken:

```
<<⟨Label⟩>>
DECLARE
    ⟨Deklarationen⟩
BEGIN
    ⟨Anweisungen⟩
END ⟨Label⟩
```

  Viele moderne Sprachen sind blockstrukturiert. In Java sind Blöcke die zusammengesetzten Anweisungen `{...}`. Ein Unterschied von PL/pgSQL zu Java ist also, dass die Deklarationen klar von den Statements getrennt sind.

- Den "Label" kann man weglassen.
- Ebenso kann der `DECLARE`-Teil entfallen.
- Ein minimaler Block ist also: `BEGIN` ⟨Anweisungen⟩ `END`

Deklarationen (1)

- Eine einfache Variablendeklaration hat die Form:

  ⟨Variable⟩ ⟨Datentyp⟩ ;

  [https://www.postgresql.org/docs/current/plpgsql-declarations.html]

- Man kann der Variablen gleich einen Wert zuweisen:

  ⟨Variable⟩ ⟨Datentyp⟩ := ⟨Term⟩ ;

  Statt ":=" kann man auch "`DEFAULT`" schreiben. Variablen werden immer initialisiert. Wenn man keinen Wert angibt, werden sie auf `NULL` gesetzt.

- Man kann die Zuweisung von Nullwerten verbieten:

  ⟨Variable⟩ ⟨Datentyp⟩ `NOT NULL` := ⟨Term⟩ ;

  In diesem Fall muss man natürlich eine explizite Initialisierung vornehmen.

- Man kann Konstanten deklarieren:

  ⟨Variable⟩ `CONSTANT` ⟨Datentyp⟩ := ⟨Term⟩ ;

Deklarationen (2)

- Beispiel:

```
kontostand INTEGER NOT NULL := 0;
```

- Spezielle Typ-Angaben:
  - ⟨Tabelle⟩.⟨Spalte⟩%TYPE: Typ einer Tabellenspalte.
  - ⟨Variable⟩%TYPE: Typ einer anderen Variablen.
  - ⟨Tabelle⟩%ROWTYPE: Record-Typ für Tabellenzeilen.

    In PostgreSQL ist das "%ROWTYPE" nicht nötig, weil für jede Tabelle automatisch ein Typ gleichen Namens eingeführt wird. Mit "%ROWTYPE" wäre es aber z.B. kompatibel zu Oracle (und vielleicht klarer).

- Man kann einen anderen Namen für eine Variable einführen:

```
⟨NeuerName⟩ ALIAS FOR ⟨AlterName⟩;
```

Das geht auch mit Parametern (war vor Version 8.0 wichtig weil sonst nur $\$i$).

Wertausdrücke/Terme

- Die Wertausdrücke (Expressions, Terme) der Sprache sind exakt wie in SQL.

  In PostgreSQL führt die SQL Engine die Wertausdrücke aus (als "Prepared Statements" mit den PL/pgSQL Variablen als Parameter).

- Boolesche Ausdrücke (z.B. nach `IF`) entsprechen also `WHERE`-Bedingungen, u.a. mit: `=`, `<>`, `<`, `>`, `<=`, `>=`, `LIKE`, `BETWEEN`, `IS NULL`, `AND`, `OR`, `NOT`. Auch `IN`, `EXISTS`!

- Allgemein sind Wertausdrücke in PL/pgSQL alles, was man in der `SELECT`-Liste als Wert einer Ergebnisspalte schreiben kann.

  Für Zahlen kann man also +, -, *, / verwenden, für Zeichenketten || (Konkatenation). Man kann natürlich alle Datentyp-Funktionen benutzen und eigene Funktione aufrufen und sogar Unteranfragen als Terme schreiben. Diese dürfen wie üblich nur maximal einen Wert liefern.

Zuweisungen, Funktions-Aufrufe

- Eine Zuweisung hat die Form:

  ⟨Variable⟩ := ⟨Term⟩;

  In Oracle wären IN-Parameter schreibgeschützt, und von OUT-Parametern könnte man den Wert nicht abfragen. In PostgreSQL ist beides möglich.

- Wenn man einen Funktionsaufruf nur wegen der Seiteneffekte ausführen will, muss man in PostgreSQL Folgendes schreiben:

  PERFORM ⟨Funktion⟩(⟨Argumentliste⟩);

- Selbst eine Funktion mit VOID Ergebnis kann nicht einfach wie in Java aufgerufen werden:

  ⟨Funktion⟩(⟨Argumentliste⟩); -- Syntaxfehler

  Ebenso geht nicht ein SELECT, dessen Ergebnis nicht verwendet wird (selbst wenn das Ergebnis VOID ist): SELECT ⟨Funktion⟩(⟨Argumentliste⟩);
  In Oracle gibt es dagegen kein PERFORM, sondern den üblichen Prozeduraufruf.

Prozeduren

- Lange Zeit hatte PostgreSQL nur "`CREATE FUNCTION`" (s.o.).

  In Analogie zu anderen Systemen kann man doch von "Stored Procedures" reden. Interessanterweise heisst die Tabelle im Systemkatalog auch `pg_proc`.

- Seit Version 11 hat PostgreSQL auch "`CREATE PROCEDURE`".

  Auf meinem Rechner läuft 2020 noch Version 9.2. Diese Version ist Standard in CentOS 7 (konservative Linux-Distribution, Support bis 2024).

- Prozeduren sind mit `CALL` aufzurufen, und nicht in `SELECT`-Anfragen zu verwenden.

  [https://www.postgresql.org/docs/current/xproc.html]
  [https://www.postgresql.org/docs/current/sql-createprocedure.html]
  [https://www.postgresql.org/docs/current/sql-call.html]

- Prozeduren können auch Transaktionen kontrollieren.

  Außerdem wird so die Kompatibilität zu Oracle und zum Standard verbessert.

Bedingte Anweisungen

- Eine `IF`-Anweisung wird in PL/pgSQL so geschrieben:

```
IF ⟨Bedingung⟩ THEN
    ⟨Folge von Anweisungen⟩
ELSIF ⟨Bedingung⟩ THEN
    ⟨Folge von Anweisungen⟩
  ⋮
ELSE
    ⟨Folge von Anweisungen⟩
END IF;
```

  Die `ELSIF`- und `ELSE`-Teile sind optional.

  [https://www.postgresql.org/docs/12/plpgsql-control-structures.html]

- Außerdem gibt es `CASE` wie in SQL.

```
CASE ... WHEN ... THEN ... ELSE ... END CASE
CASE WHEN ... THEN ... ELSE ... END CASE
```

Schleifen (1)

- Die `WHILE`-Schleife führt eine Folge von Anweisungen (Schleifenrumpf) so lange aus, wie eine Bedingung wahr ist:

```
WHILE ⟨Bedingung⟩ LOOP
    ⟨Folge von Anweisungen⟩
END LOOP;
```

Man kann vor dem `WHILE` einen Label in `<<...>>` schreiben, und den Label (ohne `<<...>>`) nach dem `END WHILE` wiederholen. Siehe auch `EXIT` unten.

- Z.B. Berechnung von $20! = 1 * 2 * 3 * \cdots * 20$ ("20 Fakultät"):

```
factorial := 1;
i := 1;
WHILE i <= 20 LOOP
    factorial := factorial * i;
    i := i + 1;
END LOOP;
```

Schleifen (2)

- Die FOR-Schleife führt eine Folge von Anweisungen (den Schleifenrumpf) einmal für jeden möglichen Wert einer ganzzahligen Laufvariable (in einem Intervall) aus:

```
FOR ⟨Variable⟩ IN [REVERSE] ⟨Start⟩..⟨Ende⟩ LOOP
    ⟨Folge von Anweisungen⟩
END LOOP;
```

  Nach dem Intervall kann man auch noch mit BY eine Schrittweite festlegen. Die Variable wird automatisch mit Typ integer deklariert und ist nur im Schleifenrumpf zugreifbar. FOR-Schleifen für Anfrageergebnisse: s.u.

- Beispiel (nochmals Berechnung von $20! = 1 * 2 * 3 * \cdots * 20$):

```
factorial := 1;
FOR i IN 1..20 LOOP
    factorial := factorial * i;
END LOOP;
```

Schleifen (3)

- Die LOOP-Anweisung führt eine Folge von Anweisungen (Schleifenrumpf) wiederholt aus, bis EXIT erreicht wird.

```
LOOP
    ⟨Folge von Anweisungen⟩
END LOOP;
```

- "EXIT;" beendet die innerste Schleife, in der es sich befindet.

  Diese Anweisung entspricht dem break in Java. Man darf EXIT auch in den anderen Schleifentypen benutzen (z.B. WHILE).

- Statt

```
IF ⟨Bedingung⟩ THEN EXIT; END IF;
```

  kann man auch schreiben:

```
EXIT WHEN ⟨Bedingung⟩;
```

Schleifen (4)

- Berechnung von 20! mit der LOOP-Anweisung:

```
factorial := 1;
i := 1;
LOOP
    factorial := factorial * i;
    i := i + 1;
    EXIT WHEN i > 20;
END LOOP;
```

Natürlich ist die FOR-Schleife der richtige Schleifentyp für die Berechnung der Fakultätsfunktion, da die Anzahl Schleifendurchläufe vorab bekannt ist.

- Man kann Schleifen mit einem Label markieren:

```
<<Name>> LOOP ...
```

und dann mit "`EXIT Name;`" diese (äußere) Schleife verlassen.

- `CONTINUE` beginnt sofort den nächsten Schleifendurchlauf.

RETURN

- Falls eine Funktion nicht den Ergebnistyp `VOID` hat, und keine Ausgabe-Parameter, muss man den Funktionswert mit

  `RETURN` ⟨Wertausdruck⟩`;`

  festlegen. Damit wird die Ausführung der Funktion beendet.

  Die Kontrolle kehrt zum Aufrufer zurück, deswegen der Name. Es ist ein Fehler, wenn die Kontrolle das `END` am Ende des Funktions-Rumpfes erreicht.

- In Funktionen mit `VOID` Ergebnis (oder `OUT` Parametern) kann man die Ausführung der Funktion beenden mit

  `RETURN;`

- Funktionen mit `SETOF` Ergebnis können mit

  `RETURN NEXT` ⟨Wertausdruck⟩`;`

  einen Wert zum Ergebnis hinzufügen.

  Die Ausführung endet dadurch nicht, erst mit `RETURN;`.

  Es gibt auch `RETURN QUERY` ⟨Q⟩ (Hinzufügen zum Ergebnis ohne Beenden).

SQL: Updates

- SQL-Anweisungen, die kein Ergebnis lieferen, z.B. `INSERT`, `UPDATE`, `DELETE`, sind auch Anweisungen in PL/pgSQL.

  [https://www.postgresql.org/docs/current/plpgsql-statements.html]

  In PL/pgSQL sind auch `CREATE TABLE` Anweisungen möglich, und im Gegensatz zur Implementierungssprache SQL kann man die Tabellen hier auch gleich verwenden (z.B. `INSERT`). `COMMIT` führt dagegen zum Fehler. In Oracle's PL/SQL wäre `CREATE TABLE` nicht möglich, `COMMIT` aber schon.

- Das Ergebnis einer Anfrage kann man dagegen nicht einfach ignorieren (siehe `PERFORM` oben).

- Dabei werden Variablen und Parameter an Stellen in der Anweisung, wo ein Datenwert (Konstante) stehen könnte, durch ihren Wert ersetzt ("Variablensubstitution").

  [https://www.postgresql.org/docs/current/plpgsql-implementation.html]

  An Stellen, wo Tabellennamen stehen müssen, findet keine Substitution statt.

SQL: SELECT INTO (1)

- In SQL-Anfragen, die genau eine Ergebniszeile liefern, kann man mit der `INTO`-Klausel angeben, in welche Variablen das Ergebnis gespeichert werden soll:

```
SELECT VORNAME, NACHNAME
INTO   vname, nname
FROM   STUDENTEN
WHERE  SID = snr;
```

  Hier wird der Wert der Variablen `snr` eingesetzt, die Anfrage ausgeführt, und das Ergebnis in die Variablen `vname` und `nname` geschrieben.

- Eigentlich ist es ein Fehler, wenn die Anfrage keine oder mehr als eine Ergebniszeile liefert.

  Bei PostgreSQL wird aber nur eine Exception (`NO_DATA_FOUND`, `TOO_MANY_ROWS`) erzeugt, wenn man `INTO STRICT` schreibt. Sonst wird die erste Ergebniszeile zugewiesen, bzw. Nullwerte, wenn es keine gibt. Oracle liefert die Fehler.

SQL: SELECT INTO (2)

- Man kann auch eine Variable vom Zeilentyp verwenden:

```
CREATE FUNCTION sname(sid NUMERIC) RETURNS TEXT
    LANGUAGE PLPGSQL AS $$
    DECLARE
        s STUDENTEN%ROWTYPE; -- Oder: RECORD
    BEGIN
        SELECT * INTO s FROM STUDENTEN
            WHERE STUDENTEN.SID = sname.sid;
        IF FOUND THEN -- Status-Variable
            RETURN s.VORNAME|| ' ' ||s.NACHNAME;
        ELSE
            RETURN 'Unbekannt: ' || sname.sid;
        END IF;
    END $$;
```

`RECORD` ist ein Wildcard-Typ für beliebige Zeilen ("pseudo type").

SQL: Status-Abfragen

- PostgreSQL setzt in vielen Fällen die spezielle boolesche Variable `FOUND`, die anzeigt, dass mindestens eine Zeile geliefert/verarbeitet wurde, z.B. nach
  - `SELECT INTO`: Zeile gefunden oder nicht.
  - `INSERT`, `UPDATE`, `DELETE`: Mindestens eine Zeile betroffen.
  - `FOR` über einem Anfrageergebnis: Die Schleife wurde mindestens einmal durchlaufen.

    Die Variable wird nach dem Ende der Schleife gesetzt.

  Die Variable ist lokal in jeder Funktion, initialisiert auf `false`.
- Man kann eine Variable auf die Anzahl der von der letzten SQL-Anweisung gelieferten bzw. bearbeiteten Zeilen setzen:

```
GET DIAGNOSTICS i = ROW_COUNT;
```

SQL: SELECT mit Schleife (1)

- Wenn eine Anfrage mehrere Ergebnis-Zeilen liefern kann, muss man die in einer Schleife verarbeiten.

  Das ist der sogenannte "impedance mismatch": SQL ist deklarativ und mengenorientiert, die umgebene Programmiersprache ist meist imperativ und kann nur einzelne Werte auf einmal verarbeiten. Deduktive Datenbanken sind mit dem Anspruch angetreten, Deklarativität und Mengenorientierung auch auf den Programm-Anteil einer Anwendung auszudehnen.

- Man kann Schleifen über Anfrageergebnissen schreiben:

```
FOR ⟨Variable(n)⟩ IN ⟨Anfrage⟩
LOOP
    ⟨Folge von Anweisungen⟩
END LOOP;
```

  Wie immer ist <<⟨Label⟩>> vor der Schleife möglich, und der gleiche ⟨Label⟩ dann nach dem `END LOOP`. Die Anfrage muss man natürlich ohne ";" am Ende schreiben. Variablen (durch "," getrennt) müssen vorher deklariert sein.

SQL: SELECT mit Schleife (2)

- Beispiel (Studenten-Namen als CSV-String):

```
CREATE FUNCTION namen_csv() RETURNS TEXT
    LANGUAGE PLPGSQL AS $$
    DECLARE
        csv text; v text; n text;
    BEGIN
        csv := '';
        FOR v,n IN SELECT VORNAME, NACHNAME
                   FROM   STUDENTEN
                   ORDER  BY NACHNAME, VORNAME
        LOOP
            csv := csv || v || ',' || n || E'\n';
        END LOOP;
        RETURN csv;
    END $$; -- Anmerkungen siehe nächste Folie
```

SQL: SELECT mit Schleife (3)

Anmerkungen:

- `E'...'` ist die PostgreSQL-spezifische erweiterte Syntax für String-Literale mit "Escape-Sequenzen" wie in C/Java.

  [https://www.postgresql.org/docs/current/sql-syntax-lexical.html]

  `E'\n'` ist also ein Zeilenumbruch. Dagegen ist `'\n'` eine Zeichenkette aus einem Rückwärts-Schrägstrich und einem "n". D.h. Standard-SQL interpretiert "\" nicht, erlaubt aber Zeilenumbrüche in `'...'`.

- Der PostgreSQL-spezifische Typ `TEXT` entspricht `VARCHAR` ohne festgelegte Begrenzung der Länge.

  Nach dem SQL-Standard kann man zwar den Typ `CHAR` ohne Längenangabe verwenden, aber nicht den Typ `VARCHAR(`*n*`)`. Es ist so oder so nicht portabel.

- Statt der einzelnen Variablen pro Ergebnis-Spalte kann man auch eine Variable `r` vom Pseudotyp "`RECORD`" verwenden, und dann z.B. mit `r.VORNAME` auf die Werte zugreifen.

SQL: SELECT mit Cursor (1)

- Klassisch verwendet man einen "Cursor", um über ein Anfrage-Ergebnis mit mehr als einer Zeile zu iterieren.
- Der Cursor muss im `DECLARE`-Abschnitt deklariert werden:

  `⟨Cursor-Name⟩ CURSOR FOR ⟨Anfrage⟩;`
- Dann öffnet man den Cursor (führt im Prinzip Anfrage aus):

  `OPEN ⟨Cursor-Name⟩;`
- Nun holt man Ergebniszeilen in einer Schleife in Variablen (d.h. setzt die Variablen auf die Werte der nächsten Zeile):

  `FETCH ⟨Cursor-Name⟩ INTO ⟨Variable(n)⟩;`
- Die Variable `FOUND` zeigt an, ob das erfolgreich war.
- Am Ende schließt man den Cursor (gibt Resourcen frei):

  `CLOSE ⟨Cursor-Name⟩;`

SQL: SELECT mit Cursor (2)

```
CREATE FUNCTION namen_csv() RETURNS TEXT AS $$
    DECLARE
        csv text := ''; v text; n text;
        c CURSOR FOR SELECT VORNAME, NACHNAME
                     FROM   STUDENTEN
                     ORDER  BY NACHNAME, VORNAME;
    BEGIN
        OPEN c;
        LOOP
            FETCH c INTO v, n;
            EXIT WHEN NOT FOUND;
            csv := csv || v || ',' || n || E'\n';
        END LOOP;
        CLOSE c;
        RETURN csv;
    END $$ LANGUAGE PLPGSQL;
```

SQL: SELECT mit Cursor (3)

Warum einen Cursor verwenden?

- Man kann die aktuell gelesene Zeile ändern (oder löschen):

```
UPDATE STUDENTEN SET ...
       WHERE CURRENT OF c;
```

  Die Anfrage sollte "`FOR UPDATE`" enthalten.

- Cursor lassen auch Sprünge in der Tupelmenge zu.

  [https://www.postgresql.org/docs/9.2/plpgsql-cursors.html]

  In der Cursor-Deklaration sollte man "`SCROLL CURSOR`" statt "`CURSOR`" schreiben, wenn man sich auch rückwärts bewegen will. Man kann beim `FETCH` noch eine Position angeben (der Default ist `NEXT`), außerdem gibt es einen Befehl `MOVE`.

- Falls komplizierte Kontrollstruktur, nicht einfache Schleife.
- Man kann Cursor zwischen Funktionen weitergeben.

  Der Typ `refcursor` passt für alle Cursor. Ggf. effizienter als z.B. großes Array.

SQL: SELECT mit Cursor (4)

Details:

- Auch `FOR`-Schleife über `SELECT` verwendet intern einen Cursor.

  Er ist die DB-Schnittstelle zur Verarbeitung von Anfragen, Zugriff auf Ergebnisse.

- Noch offene Cursor werden am Transaktionsende geschlossen.

- Bei großen Anfrageergebnissen wird die Anfrage erst beim `FETCH` nach und nach ausgewertet.

  [https://www.postgresql.org/docs/current/protocol-overview.html]

  Die Schritte der Anfrageauswertung sind:

  (1) Parse: überführt den Text der Anfrage in ein Prepared Statement

  (2) Bind (beim `OPEN`): Fügt Werte für Parameter ein, Ergebnis heißt "Portal"

  (3) Execute (ggf. mehrfach): Berechnet Ergebnisse, unterbricht Arbeit ggf. nach vorgegebener Anzahl Tupel.

- Ein offener Cursor kann eine effiziente Repräsentation einer großen Tupelmenge sein (nicht "materialisiert").

SQL: SELECT mit Cursor (5)

Parameter eines Cursors:

- Alle Variablen, die vor der Cursor-Definition mit der Anfrage deklariert sind, werden implizit Parameter des Cursors.

  Es findet im Prinzip wie sonst auch eine Variablen-Substitution statt, aber hier werden erst die Werte der Variablen beim `OPEN` verwendet.

- Man kann bei der Cursor-Deklaration Parameter angeben:

```
c CURSOR (max_sid integer) FOR
        SELECT ... WHERE SID <= max_sid;
```

  Es werden aber effektiv auch alle weiteren Variablen, die vor dieser Zeile deklariert sind, und in der Anfrage vorkommen, Parameter der Anfrage.

- Beim `OPEN` gibt man dann die Werte für die Parameter an:

```
OPEN c(103);
```

  Auch von allen anderen Variablen, die in der Anfrage zu substituieren sind, gelten die Werte, die beim `OPEN` in den Variablen stehen.

SQL: SELECT mit Cursor (6)

Weitere Möglichkeiten:

- Man kann den Cursor auch als "`refcursor`" ohne Anfrage deklarieren, und die Anfrage dann beim `OPEN` angeben.

  Ein Cursor ohne Anfrage heißt "unbound".

- Bei `FETCH` kann man auch eine Variable vom Typ `RECORD` benutzen, statt Variablen für die einzelnen Spalten.

- Es gibt auch eine `FOR`-Schleife für Cursor:

```
FOR ⟨RecordVar⟩ IN ⟨Cursor⟩ LOOP
    ⟨Folge von Anweisungen⟩
END LOOP;
```

  Falls der Cursor Parameter hat, sind Werte dafür nach dem Namen des Cursors in `(...)` anzugeben (wie beim `OPEN`).

Fehlerbehandlung (1)

- Bei der Abarbeitung von Anfragen können Laufzeit-Fehler (Exceptions) auftreten, z.B.
  - `division_by_zero`: Division durch 0.
  - `numeric_value_out_of_range`: Wert zu groß für Datentyp.
  - `integrity_constraint_violation`: IB verletzt.
  - `unique_violation`: Schlüsselbedingung verletzt.
  - `check_violation`: `CHECK`-Bedingung verletzt.
  - `deadlock_detected`: Parallele Transaktionen warten zyklisch.
  - `insufficient_resources`: Platte voll etc.
  - `no_data_found`: `SELECT STRICT INTO` mit leerem Ergebnis.
  - `too_many_rows`: `SELECT STRICT INTO` mit > 1 Zeilen.

[https://www.postgresql.org/docs/9.2/errcodes-appendix.html]

Fehlerbehandlung (2)

- Normalerweise führen solche Exceptions zum Abbruch der umgebenden Anfrage.
- Außerdem wird die aktuelle Transaktion abgebrochen:
  - Jede weitere SQL-Anweisung gibt einen Fehler.

    "ERROR: current transaction is aborted, commands ignored until end of transaction block". Der Nutzer oder das Programm sollten keine weiteren SQL-Befehle in dieser Transaktion mehr eingeben. Falls das Programm den Fehler aber nicht mitbekommen hat, und einfach weiter durchläuft, stellt PostgreSQL sicher, dass alle weiteren Befehle der Transaktion ignoriert werden.
  - Man sollte jetzt die Transaktion mit `ROLLBACK` beenden.

    Im normalen Autocommit Modus geschieht das automatisch.
  - Gibt man `COMMIT` ein, bekommt man die Antwort `ROLLBACK`, und es werden alle Änderungen zurückgenommen.

Fehlerbehandlung (3)

- Man kann "Exception Handler" am Block-Ende schreiben:

```
DECLARE
    ⟨Deklarationen⟩
BEGIN
    ⟨Folge von Anweisungen A⟩
EXCEPTION
    WHEN ⟨Exceptions E1⟩ THEN
        ⟨Folge von Anweisungen H1⟩
    WHEN ⟨Exceptions E2⟩ THEN
        ⟨Folge von Anweisungen H2⟩
    ...
END;
```

Tritt in A eine Exception e auf, werden alle weiteren Anweisungen übersprungen, und stattdessen der erste Exception-Handler H_i ausgeführt, auf dessen Exception-Spezifikation E_i ($\rightarrow$ nächste Folie) die aufgetretene Exception e passt.

Fehlerbehandlung (4)

Exception-Auswahl:

- Die Exception-Spezifikation (nach `WHEN`) ist eine Liste von Fehlertypen wie `unique_violation`:

  $$\texttt{WHEN } e_1 \texttt{ OR } \ldots \texttt{ OR } e_n \texttt{ THEN } \ldots$$

  Vollständige Liste im Anhang A des PostgreSQL-Handbuchs.

- Man kann aber auch alle übrigen Exceptions abfangen:

  ```
  WHEN OTHERS THEN ....
  ```

  Nicht abgedeckt: `QUERY_CANCELLED`, `ASSERT_FAILURE` (soll man nicht abfangen).

- Es gibt auch "Oberklassen" in den Exceptions, die auf alle Exceptions ihrer Gruppe passen, z.B. würde

  ```
  WHEN integrity_constraint_violation THEN ...
  ```

  auch bei einer `unique_violation` greifen.

  Diese Exceptions sind im Anhang an einem Statuscode *XX*000 erkennbar.

Fehlerbehandlung (5)

- Alle Datenbank-Änderungen des Blocks, in dem die Exception aufgetreten ist, werden zurückgenommen.

  Das System setzt beim `BEGIN` eines Blocks mit Exception Handler einen "Savepoint" und macht automatisch einen Rollback bis zu diesem Punkt. Man hat deswegen mit geschachtelten Blöcken Einfluss darauf, wieviel an Änderungen zurückgenommen wird. Die Transaktion selbst endet nicht, wenn die Exception behandelt wird (Savepoints sind eine feinere Struktur innerhalb von Transaktionen). Variablen haben dagegen im Exception Handler den letzten Wert, den sie im Programm hatten.

- Bei Blöcken ohne (passenden) Exception Handler wird die Exception nach außen weiter gegeben.

  Das geht auch über Funktionsgrenzen hinweg. Entsprechend werden auch mehr Änderungen zurückgenommen. Am Ende steht der Transaktionsabbruch.

- Weitere Informationen zum Fehler (z.B. Tabelle, Spalte): `GET STACKED DIAGNOSTICS`: siehe Handbuch.

Fehlerbehandlung (6)

- Beispiel:

```
CREATE FUNCTION sname(sid NUMERIC) RETURNS TEXT
    LANGUAGE PLPGSQL AS $$
    DECLARE
        v TEXT; n TEXT;
    BEGIN
        SELECT VORNAME, NACHNAME
            INTO STRICT v, n
            FROM STUDENTEN
            WHERE STUDENTEN.SID = sname.sid;
        RETURN v || ' ' || n;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN 'Unbekannt: ' || sname.sid;
    END $$;
```

Mitteilungen ausgeben, Exceptions auslösen

- Beispiel für Mitteilung während der Funktions-Ausführung:

```
RAISE NOTICE 'Unbekannte SID';
```

[https://www.postgresql.org/docs/12/plpgsql-errors-and-messages.html]
`NOTICE` ist die Schwere des Fehlers. Mögliche Stufen sind: `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `EXCEPTION`. In der Konfiguration des Servers kann festgelegt werden, welche Schwere dem Benutzer angezeigt wird (`client_min_messages`), und welche in einer Datei auf dem Server protokolliert wird (`log_min_...`).
Nur bei `EXCEPTION` (ist Default) kommt es zum Abbruch der Transaktion.

- Der Text der Mitteilung muss eine String-Konstante sein, aber man kann mit "%" Parameter einbinden:

```
RAISE NOTICE 'Unbekannte SID: %', SID;
```

- Beispiel für Fehler (viele weitere Optionen im Handbuch):

```
RAISE EXCEPTION unique_violation
USING MESSAGE = 'SID ' || SID || ' ist vergeben.';
```

Literatur/Quellen

- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Edition, 1999. Section 10.5, "Programming Oracle Applications"
- Akhil Bhadwal: What You Need to Know About Stored Procedures [https://hackr.io/blog/stored-procedures]
- Wikipedia: PL/pgSQL [https://de.wikipedia.org/wiki/PL/pgSQL]
- PostgreSQL Documentation: CREATE FUNCTION [https://www.postgresql.org/docs/9.5/sql-createfunction.html]
- PostgreSQL Documentation: Extending SQL [https://www.postgresql.org/docs/9.5/extend.html]
- PostgreSQL Documentation: Query Language (SQL) Functions [https://www.postgresql.org/docs/9.5/xfunc-sql.html]
- PostgreSQL Documentation: PL/pgSQL — SQL Procedural Language [https://www.postgresql.org/docs/9.5/plpgsql.html]
- Michael Gertz: Oracle/SQL Tutorial, 1999. [http://www.mathcs.emory.edu/~cheung/Courses/377/Others/tutorial.pdf]
- Oracle Database Application Debeloper's Guide — Fundamentals — 10g Rel. 2 [https://docs.oracle.com/cd/B19306_01/appdev.102/b14251.pdf]
- Oracle Database: PL/SQL Language Reference, 11g Rel. 2 [https://docs.oracle.com/cd/E11882_01/appdev.112/e25519.pdf]