Datenbanksysteme

VU 184.686, WS 2022

PL/pgSQL

Johannes Fichte, Felix Winter

Institut für Logic and Computation
Technische Universität Wien

Acknowledgments

Die Folien sind eine Weiterentwicklung der Folien von Reinhard Pichler und Sebastian Skritek.

Weiterführende Informationen zu den behandelten Themen:
`https://www.postgresql.org/docs/current/static/plpgsql.html`

Ein Wort zum 4. Übungsblatt

Datenbankserver:

- `bordo.dbai.tuwien.ac.at`
- Zugriff ausschließlich mittels `ssh` von "überall" her möglich
- Accounts:
  - Alle TeilnehmerInnen erhalten einen Account `u<Matrikelnummer>`
  - Passwörter sind in TUWEL verfügbar
  - Passwort ändern: Befehl `passwd`
    (Neues Passwort nicht vergessen!!)
- Sie erhalten ein Home-Verzeichnis (Upload mittels scp)
- DB-Username = Linux-Username
- DB-Zugriff: `psql` (Authentifizierung über SSH-Account)

Warum PostgreSQL?

- OpenSource (minimale Lizenz Anforderungen)
- Lässt sich ohne Anmeldung leicht herunterladen
- Leicht auf allen gänigen Plattformen verfügbar
- Timestamp-based concurrency control
- Treiber/Interfaces für alle gängigen Programmiersprachen
- SQL Standard
- Verbreitung (OpenStreetMap, BASF, ISS, Instagram, TripAdvisor, NOAA, Guardian)
- Praxis: extrem leicht erweiterbar, für Standardanwendungen gut optimiert, ungeeignet für Analyse (spaltenbasierte Auswertung)

PL/pgSQL

1. Einführung
2. Struktur von PL/pgSQL Programmen
3. Variablen
4. Ausdrücke
5. Kontrollstrukturen
6. Cursors

Inhalt

Inhalt

SQL User Defined Functions

SQL erlaubt die Definition von Funktionen:

SQL User Defined Functions

SQL erlaubt die Definition von Funktionen:

Beispiel

```
CREATE FUNCTION neuesSemester() RETURNS void
AS $$
  UPDATE Studierende SET Sem=Sem+1;
  ...
$$ LANGUAGE SQL;
```

SQL User Defined Functions

SQL erlaubt die Definition von Funktionen:

Beispiel

```
CREATE FUNCTION neuesSemester() RETURNS void
AS $$
  UPDATE Studierende SET Sem=Sem+1;
  ...
$$ LANGUAGE SQL;
```

Aufruf mittels:

```
SELECT neuesSemester();
```

SQL vs. prozedurale Sprachen

SQL:

- Deklarative Programmiersprache (*was*, nicht *wie*)

SQL vs. prozedurale Sprachen

SQL:

- Deklarative Programmiersprache (*was*, nicht *wie*)
- Vorteile: Bessere Optimierungsmöglichkeiten, kompakt, ...

SQL vs. prozedurale Sprachen

SQL:

- Deklarative Programmiersprache (*was*, nicht *wie*)
- Vorteile: Bessere Optimierungsmöglichkeiten, kompakt, ...
- Nachteile: eingeschränkte Ausdruckskraft; "typische" Programmiersprachenkonstrukte wären oft wünschenswert

SQL vs. prozedurale Sprachen

SQL:

- Deklarative Programmiersprache (*was*, nicht *wie*)
- Vorteile: Bessere Optimierungsmöglichkeiten, kompakt, ...
- Nachteile: eingeschränkte Ausdruckskraft; "typische" Programmiersprachenkonstrukte wären oft wünschenswert

$\Rightarrow$ fast alle DBMS bieten **Prozedurale DB-Programmiersprachen**

Warum prozedurale DB-Sprachen?

- Performance:
  - Weniger unnötiges “hin- und her” zwischen Client und DB
  - (Netzwerk-) Traffic wird eingespart
  - Mehrfaches Parsen der selben Abfrage kann vermieden werden
- Anwendungslogik befindet sich an zentraler Stelle
- Genauere Zugriffsbeschränkungen möglich

Abwägung: Einsatz von Prozeduralen Sprachen

Perspektive: Datenbanken

- Anwendungslogik befindet sich an zentraler Stelle
- Schnellerer Zugriff

Perspektive: Software-Entwicklung

- Unklar wo die Anwendungslogik tatsächlich ist
- Datenbanken skalieren für Nutzerin womöglich nicht gut genug
- PLSQL kann schwieriger lesbar sein als gewohnter Anwendungscode (Zusätzliche Kenntnisse notwendig)
- Inhaltliche Trennung erschwert: Testbarkeit / Debugbarkeit kann leiden
- Programmiertools weniger komfortabel

Inhalt

PostgreSQL

- PostgreSQL ermöglicht es *user-defined functions* in beliebigen Sprachen zu schreiben.
- Der Source Code wird von PostgreSQL als Text behandelt und an den jeweiligen Adapter der Programmiersprache weitergegeben.
- Vier Sprachen standardmäßig unterstützt:
  PL/pgSQL, PL/Tcl, PL/Perl, PL/Python.
- Zusätzliche Sprachen können nachinstalliert werden.

- In dieser LVA: PL/pgSQL
  - Sehr ähnlich zu PL/SQL (Oracle)

Inhalt

PL/pgSQL-Programmteile in der Vorlesung

- Programme sind meistens nur auszugsweise auf den Folien wiedergegeben (nur die "wesentlichen" Teile).
- Durch die Auslassungen ist der Source Code auf den Folien eventuell nicht ablauffähig.

Quellen

- PostgreSQL Online Dokumentation:
  `https://www.postgresql.org/docs/current/`
- Folien dienen nur zum Überblick. Details findet man in der Online Dokumentation.

Inhalt

Inhalt

User-defined Functions (PostgreSQL)

```
CREATE [OR REPLACE] FUNCTION
  name ([ [argname] argtype [, ...] ])
    [ RETURNS rettype
     |RETURNS TABLE (colname coltype [, ...])]
AS $$
  ...    – eigentlicher Source Code
$$ LANGUAGE plpgsql;
```

- Source Code wird als Text übergeben
- Werden diesen “Rahmen” im Folgenden oft auslassen.

Argumente und Rückgabewerte

Argumente:

- Vier Arten von Argumenten: `IN`, `OUT`, `INOUT`, `VARIADIC`
- Angabe eines Namens is optional
  - Zugriff mittels `$i` Notation möglich
- Möglichkeit Default-Wert zu definieren

Argumente und Rückgabewerte

Argumente:

- Vier Arten von Argumenten: `IN`, `OUT`, `INOUT`, `VARIADIC`
- Angabe eines Namens is optional
  - Zugriff mittels `$i` Notation möglich
- Möglichkeit Default-Wert zu definieren

Rückgabewerte

- Typ der Rückgabewerte muss angegeben werden
  - Explizit mittels `RETURNS`
  - Implizit mittels `OUT`/`INOUT` Parametern
- Falls keine Werte zurück geliefert werden: `RETURNS void`
- Kann Tabellen (`RETURNS TABLE`) oder Menge von Werten (`SETOF`) zurückliefern

Argumente und Rückgabewerte: Beispiele

Beispiel

```
CREATE FUNCTION sum (integer, integer)
  RETURNS integer AS $$
    ... $1 + $2  ... $$ LANGUAGE plpgsql;
```

Argumente und Rückgabewerte: Beispiele

Beispiel

```
CREATE FUNCTION sum (integer, integer)
  RETURNS integer AS $$
    ... $1 + $2  ... $$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION nsum (IN a integer,
  IN b integer, OUT c integer) AS $$
   ... c = a + b ... $$ LANGUAGE plpgsql;
```

Argumente und Rückgabewerte: Beispiele

Beispiel

```
CREATE FUNCTION sum (integer, integer)
  RETURNS integer AS $$
    ... $1 + $2  ... $$ LANGUAGE plpgsql;

CREATE FUNCTION nsum (IN a integer,
  IN b integer, OUT c integer) AS $$
   ... c = a + b ... $$ LANGUAGE plpgsql;

SELECT nsum(2,3);
```

Argumente und Rückgabewerte: Beispiele

Beispiel

```
CREATE FUNCTION nsumd (IN a integer = 1,
  IN b integer DEFAULT 1, OUT c integer)
  AS $$
    ... c = a + b ...
  $$ LANGUAGE plpgsql;
```

Argumente und Rückgabewerte: Beispiele

Beispiel

```
CREATE FUNCTION nsumd (IN a integer = 1,
  IN b integer DEFAULT 1, OUT c integer)
  AS $$
    ... c = a + b ...
  $$ LANGUAGE plpgsql;

SELECT nsumd();
```

Inhalt

PL/pgSQL Progamme sind in Blöcken strukturiert

Beispiel (Minimalbeispiel für Summe)

```
CREATE FUNCTION nsum (IN a integer,
  IN b integer, OUT c integer) AS $$
   BEGIN
    c = a + b;
  END; $$ LANGUAGE plpgsql;
```

Struktur eines PL/pgSQL Blocks

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

Struktur eines PL/pgSQL Blocks

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

`label` Weist einem Block einen Namen zu (optional)

Struktur eines PL/pgSQL Blocks

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

`label` Weist einem Block einen Namen zu (optional)

`DECLARE` Enthält Definition der lokalen Variablen (optional)

Struktur eines PL/pgSQL Blocks

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

`label` Weist einem Block einen Namen zu (optional)

`DECLARE` Enthält Definition der lokalen Variablen (optional)

`BEGIN` Enthält die eigentliche Programmlogik

Struktur eines PL/pgSQL Blocks

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
[ EXCEPTION
  excpthandling ]
END [ label ];
```

`label`	Weist einem Block einen Namen zu (optional)
`DECLARE`	Enthält Definition der lokalen Variablen (optional)
`BEGIN`	Enthält die eigentliche Programmlogik
`EXCEPTION`	Enthält die Fehlerbehandlung (optional)

Elemente eines PL/pgSQL Blocks

- Blöcke können beliebig geschachtelt werden

Elemente eines PL/pgSQL Blocks

- Blöcke können beliebig geschachtelt werden
- Kleinstmöglicher Block:

```
BEGIN
END;
```

bzw.

```
BEGIN
  NULL;  -- Oracle Kompatibilität
END;
```

Inhalt

Inhalt

Variablen-Definition

- Variablen können beliebigen SQL Typ haben
- Alle Variablen müssen in `DECLARE` Sektion deklariert werden
- Syntax:

```
name [CONSTANT] type [NOT NULL] [{DEFAULT|=} expression];
```

Variablen-Definition

- Variablen können beliebigen SQL Typ haben
- Alle Variablen müssen in DECLARE Sektion deklariert werden
- Syntax:

```
name [CONSTANT] type [NOT NULL] [{DEFAULT|=} expression];
```

Beispiel (Variablendeklarationen)

```
matrknr integer;
input1 ALIAS FOR $1;
nameNeu ALIAS FOR nameAlt;
note integer NOT NULL = 1;
rang CONSTANT varchar(2) = 'C4';
```

Inhalt

Spezielle Variablentypen

- Copying Types
- Row Types
- Record Types

Copying Types

- Ermöglicht es den Typ einer anderen Variable oder einer Tabellenspalte zu übernehmen

Beispiel

```
matrnr Studierende.MatrNr%TYPE;
vorl Vorlesungen.Titel%TYPE;
aktuell vorl%TYPE;
```

Copying Types

- Ermöglicht es den Typ einer anderen Variable oder einer Tabellenspalte zu übernehmen

Beispiel

```
matrnr Studierende.MatrNr%TYPE;
vorl Vorlesungen.Titel%TYPE;
aktuell vorl%TYPE;
```

Vorteile:

- Typ muss nicht bekannt sein
- Wenn sich der Typ ändert muss der Code nicht notwendigerweise geändert werden

Nachteile:

- Unvorhergesehene Seiteneffekte

Row Types

- Aus mehreren Feldern zusammengesetzte Variable (*composite type*)
- Kann gesamte Zeile einer Relation enthalten
- Fixe Struktur
- Einzelne Felder werden mittels Punkt-Notation angesprochen: `rowvar.field`
- Typische Verwendung:
  `name table_name%ROWTYPE`

Beispiel

```
student Studierenden%ROWTYPE;
```

Record Types

Ähnlich wie Row-Type, aber:

- Struktur nicht fest vorgegeben
- Übernimmt Struktur dynamisch, "wiederverwendbar"
- Verwendung: `name RECORD`

Beispiel

```
ergebnis RECORD;
SELECT * INTO ergebnis FROM Studierende ...
SELECT * INTO ergebnis FROM Professorinnen ...
```

Inhalt

Substitution von Variablen in Ausdrücken

- **Problem:** Zeichenkette kann Variable oder Tabelle bezeichnen

Beispiel

```
MatrNr Studierende.Matrnr%TYPE;
SELECT MatrNr INTO MatrNr FROM Studierende
  WHERE MatrNr=MatrNr;
```

Substitution von Variablen in Ausdrücken

- **Problem:** Zeichenkette kann Variable oder Tabelle bezeichnen

Beispiel

```
MatrNr Studierende.Matrnr%TYPE;
SELECT MatrNr INTO MatrNr FROM Studierende
  WHERE MatrNr=MatrNr;
```

- Kein Problem falls auf Grund von Syntax eindeutig
- Falls nicht eindeutig: Fehler wird ausgegeben (default)

Substitution von Variablen in Ausdrücken

- **Problem:** Zeichenkette kann Variable oder Tabelle bezeichnen

Beispiel

```
MatrNr Studierende.Matrnr%TYPE;
SELECT MatrNr INTO MatrNr FROM Studierende
  WHERE MatrNr=MatrNr;
```

- Kein Problem falls auf Grund von Syntax eindeutig
- Falls nicht eindeutig: Fehler wird ausgegeben (default)
- **Lösung:** Vermeidung
  - Qualifizierte Namen (`Studierende.MatrNr`)
  - Entsprechende Namenskonventionen für Variablen

Inhalt

Inhalt

Wertzuweisungen, Vergleiche, einfache Ausdrücke

- Zuweisungsoperator: `:=` oder `=`
- Operatoren in Ausdrücken: Wie in SQL
  - Arithmetische Operatoren: `+`, `-`, `*`, `/`
  - Vergleichsoperatoren: `=`, `>`, `<`, `>=`, `<=` ungleich: `!=` oder `<>`
  - Logische Operatoren: `AND`, `OR`, `NOT`
  - Stringvergleiche: `LIKE`, `NOT LIKE` (wildcards: `%`, `_`)
  - String Konkatenation: `||`
  - Weitere SQL-Operationen: `IS NULL`, `IS NOT NULL`
    `x BETWEEN a AND b`, `x IN (1,2,3)`
  - ...

Inhalt

SQL Befehle ohne Ergebnis

SQL Befehle die keine Ergebnis (*result set*) zurückliefern, können normal aufgerufen werden

- z.B. `INSERT`, `UPDATE`, ... (ohne `RETURNING`)
- Aber nicht `SELECT`

SQL Befehle ohne Ergebnis

SQL Befehle die keine Ergebnis (*result set*) zurückliefern, können normal aufgerufen werden

- z.B. `INSERT`, `UPDATE`, ... (ohne `RETURNING`)
- Aber nicht `SELECT`

Beispiel

```
... FUNCTION anmelden(integer, integer)...
...
BEGIN
 INSERT INTO hören VALUES ($1, $2);
END; ...
```

PERFORM – Verwerfen von Ergebnissen

`PERFORM` Erlaubt es, Befehle/Abfragen auszuführen und das Ergebnis (result set) sofort zu verwerfen.

```
PERFORM statement
```

PERFORM – Verwerfen von Ergebnissen

`PERFORM` Erlaubt es, Befehle/Abfragen auszuführen und das Ergebnis (result set) sofort zu verwerfen.

```
PERFORM statement
```

- z.B. Ausführen von Abfragen oder Funktionen mit Seiteneffekten
- Bei `SELECT`-Abfragen: Ersetze `SELECT` durch `PERFORM`

Beispiel

```
PERFORM neuesSemester();
PERFORM pg_sleep(2);
```

PERFORM – Verwerfen von Ergebnissen

`PERFORM` Erlaubt es, Befehle/Abfragen auszuführen und das Ergebnis (result set) sofort zu verwerfen.

```
PERFORM statement
```

- z.B. Ausführen von Abfragen oder Funktionen mit Seiteneffekten
- Bei `SELECT`-Abfragen: Ersetze `SELECT` durch `PERFORM`

Beispiel

```
PERFORM neuesSemester();
PERFORM pg_sleep(2);

PERFORM * FROM Studierende;
```

Inhalt

INTO

`INTO` Lädt einzeilige Ergebnisse eines SQL Befehls/einer SQL Abfrage in eine `RECORD` order `ROWTYPE` Variable.

```
SELECT expr INTO [STRICT] target FROM ...;
INSERT...RETURNING expr INTO [STRICT] target;
```

INTO

`INTO` Lädt einzeilige Ergebnisse eines SQL Befehls/einer SQL Abfrage in eine `RECORD` order `ROWTYPE` Variable.

```
SELECT expr INTO [STRICT] target FROM ...;
INSERT...RETURNING expr INTO [STRICT] target;
```

- Mit `STRICT`: Abfrage muss exakt eine Zeile zurückliefern
- Ohne `STRICT`: Auch Ergebnisse mit mehr/weniger als einer Zeile erlaubt
  - Keine Zeile: (Werte in) `target` werden auf `NULL` gesetzt
  - Mehr als eine Zeile: “erste” Zeile wird zurückgegeben

Dynamische SQL Befehle ausführen

`EXECUTE` Führt den (dynamisch erstellten) SQL Befehl aus. Einzeilige Ergebnisse können mittels `INTO` in eine Variable geschrieben werden.

```
EXECUTE cmd [INTO [STRICT] var] [USING expr];
```

Dynamische SQL Befehle ausführen

`EXECUTE` Führt den (dynamisch erstellten) SQL Befehl aus. Einzeilige Ergebnisse können mittels `INTO` in eine Variable geschrieben werden.

```
EXECUTE cmd [INTO [STRICT] var] [USING expr];
```

- Kein Caching des Query-Plans
- Parameter können nur für Datenwerte verwendet werden

Dynamische SQL Befehle ausführen

`EXECUTE` Führt den (dynamisch erstellten) SQL Befehl aus. Einzeilige Ergebnisse können mittels `INTO` in eine Variable geschrieben werden.

```
EXECUTE cmd [INTO [STRICT] var] [USING expr];
```

- Kein Caching des Query-Plans
- Parameter können nur für Datenwerte verwendet werden

Beispiel

```
EXECUTE format('SELECT count(*) FROM %I '
 'WHERE Sem > $1', tabname)
 INTO c USING v_sem;
```

Inhalt

Inhalt

Rückgabewerte einer Funktion

- `OUT` und `INOUT` Variablen:
  - Geben bei Funktionsende enthaltene Werte zurück

Rückgabewerte einer Funktion

- `OUT` und `INOUT` Variablen:
  - Geben bei Funktionsende enthaltene Werte zurück
- `RETURN` *expr* ;
  - Beendet Funktion, gibt Wert von *expr* zurück
  - Nur `RETURN;` bei Typ `void` oder `OUT`/`INOUT` Variablen

Rückgabewerte einer Funktion

- `OUT` und `INOUT` Variablen:
  - Geben bei Funktionsende enthaltene Werte zurück
- `RETURN` *expr* ;
  - Beendet Funktion, gibt Wert von *expr* zurück
  - Nur `RETURN;` bei Typ `void` oder `OUT`/`INOUT` Variablen
- `RETURN NEXT` *expr* ;
  `RETURN QUERY` *query* ;
  - Erweitert Ausgabe um entsprechenden Wert
  - Beendet nicht die Funktion
  - Variante mit `EXECUTE` existiert für `RETURN QUERY`

Inhalt

IF-THEN-ELSE

```
IF boolean-expr THEN
    statements
[ ELSIF boolean-expr THEN
    statements
[ ELSIF boolean-expr THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

CASE

"Simple CASE"

```
CASE search-expr
 WHEN expr [,expr...] THEN
    statements
 [WHEN expr [,expr...] THEN
    statements
    ... ]
 [ELSE
    statements ]
END CASE;
```

"Searched CASE"

```
CASE
 WHEN boolean-expr THEN
   statements
 [WHEN boolean-expr THEN
   statements
    ... ]
 [ELSE
   statements ]
END CASE;
```

CASE

"Simple CASE"

```
CASE search-expr
 WHEN expr [,expr...] THEN
    statements
 [WHEN expr [,expr...] THEN
    statements
    ... ]
 [ELSE
    statements ]
END CASE;
```

"Searched CASE"

```
CASE
 WHEN boolean-expr THEN
   statements
 [WHEN boolean-expr THEN
   statements
    ... ]
 [ELSE
   statements ]
END CASE;
```

- Erstes passendes `WHEN` wird ausgeführt
- Weitere `WHEN` werden übersprungen
- Erreichen eines fehlenden `ELSE` führt zu Fehler

Beispiel: Simple vs. Searched CASE

Beispiel (Simple CASE – Searched CASE)

```
CASE note
 WHEN 1 THEN
  txt = 'Tutor?';
 WHEN 2,3,4 THEN
  txt = 'Positiv';
 ELSE
  txt = 'Negativ';
END CASE;
```

```
CASE
 WHEN note = 1 THEN
  txt = 'Tutor?';
 WHEN note BETWEEN
        2 AND 4 THEN
  txt = 'Positiv';
 ELSE
  txt = 'Negativ';
END CASE;
```

Inhalt

Endlosschleife LOOP ...END LOOP;

```
[ <<label>> ]
LOOP
  statements
END LOOP;
```

- Endlosschleife; explizites Verlassen mittels RETURN oder EXIT

Weitere Schleifen

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

Weitere Schleifen

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

```
[ <<label>> ]
FOR name IN [REVERSE] expr ... expr [BY expr] LOOP
  statements
END LOOP [ label ];
```

Kontrollbefehle für Schleifen: EXIT und CONTINUE

`EXIT` Bricht Bearbeitung der Schleife ab; Kontrollfluss setzt nach entsprechendem `END LOOP;` fort.

`CONTINUE` Bricht aktuellen Schleifendurchlauf ab; nächster Durchlauf der entsprechenden Schleife beginnt.

Syntax:

```
EXIT/CONTINUE [label] [WHEN boolean-expr];
```

Abfrageergebnisse durchlaufen

```
[ <<label>> ]
FOR target IN query LOOP
  statements
END LOOP [ label ];
```

- Intern als Cursor realisiert (bald mehr dazu)

Abfrageergebnisse durchlaufen

```
[ <<label>> ]
FOR target IN query LOOP
  statements
END LOOP [ label ];
```

- Intern als Cursor realisiert (bald mehr dazu)

Beispiel

```
FOR s IN SELECT * FROM Studierende LOOP
 INSERT INTO hoeren VALUES (s.MatrNr,184686);
END LOOP;
```

Beispiel

Beispiel

```
CREATE OR REPLACE FUNCTION
  suche(matrnr numeric(10)) RETURNS void AS $$
DECLARE
name     varchar(30);
semester numeric(2);
BEGIN
  SELECT s.name, s.semester INTO name, semester
    FROM students s WHERE s.matrnr = matrnr;
  IF (name IS NULL) THEN
    RAISE NOTICE 'Leider nichts gefunden';
  ELSE
    RAISE NOTICE 'Name: %, Semester: % ',
      name, semester;
  END IF;
END; $$ LANGUAGE plpgsql;
```

Inhalt

Fehlerbehandlung mittels EXCEPTIONS

```
BEGIN
  statements
EXCEPTION
  WHEN cond [OR Cond ...] THEN
    handler_statements
  [ WHEN cond [OR Cond ...] THEN
    handler_statements
    ... ]
END;
```

- Abschnitt eines Blockes; ähnlich try . . . catch in Java)
- Nicht abgefangener Fehler: Weitergabe “nach außen”, schließlich Abbruch; ROLLBACK von Datenbankänderungen
- Abgefangener Fehler: Werte lokaler Variablen bleiben erhalten, ROLLBACK aller Datenbankänderungen innerhalb des Blocks.

Fehlerbehandlung – Beispiel

Beispiel (Ändern/Anlegen einer Note)

```
LOOP
 UPDATE pruefen SET Note=1 WHERE MatrNr=mn
    AND VorlNr=vn AND PersNr=pn;
 RETURN WHEN FOUND;
 BEGIN
   INSERT INTO pruefen VALUES (mn,vn,pn,1);
   RETURN;
 EXCEPTION WHEN unique_violation THEN
   -- nichts, einfach update nochmal probieren
 END;
END LOOP;
```

Inhalt

Inhalt

Zeilenweises Lesen eines Ergebnisses – Cursor

Cursor ...

- erlauben das schrittweise durchlaufen von Ergebnissen (verhindert dass gesamtes Ergebnis auf einmal in Hauptspeicher geladen werden muss)
- können wie andere Variablen genutzt werden (auch als Parameter/Rückgabewerte von Funktionen)

Deklaration:

```
name refcursor;
name [[NO] SCROLL] CURSOR [(arguments)]
                          FOR query;
```

Cursor – Beispiel

Beispiel (Cursor – Deklarationen)

```
curs1 refcursor;
curs2 CURSOR FOR SELECT * FROM Studierende;
curs3 CURSOR (mn integer) FOR
    SELECT *
    FROM Studierende
    WHERE MatrNr = mn;
```

Inhalt

Cursor: Verwendung und Zugriff

`OPEN` Vor der Verwendung muss ein Cursor geöffnet werden. Syntax abhängig davon ob Cursor gebunden ist oder nicht.

`FETCH` Liest die nächste Zeile des Ergebnis. Gibt es keinen weiteren Eintrag wird `NULL` gelesen.

`FOUND` Variable die angibt ob `FETCH` einen Wert gelesen hat.

`MOVE` Erlaubt es den Cursor zu bewegen ohne zu lesen.

`CLOSE` Schließt den Cursor und gibt die gehaltenen Ressourcen wieder frei.

Verwendung eines Cursors – OPEN

Beispiel (Öffnen eines Cursors)

```
OPEN curs1 FOR SELECT * FROM Professorinnen;
OPEN curs2;
OPEN curs3(42);
```

- [NO] SCROLL bei nicht gebundenem Cursor
- FOR *query* nur bei nicht gebundenem Cursor

Verwendung eines Cursors – FETCH

```
FETCH [direction {FROM|IN}] cursor INTO target;
```

Beispiel

```
FETCH curs1 INTO rowvar;
FETCH LAST FROM curs2 INTO stud;
FETCH PRIOR FROM curs2 INTO stud;
FETCH RELATIVE 2 FROM curs2 INTO stud;
FETCH curs3 INTO mn, name, sem;
```

Schreibzugriff

- Cursor erlauben es auch den aktuellen Datensatz zu modifizieren (`UPDATE`/`DELETE`)
- Abfrage muss entsprechend einfach sein (z.B. keine Aggregation)
- Empfohlen: Cursor als `FOR UPDATE` deklarieren
  - Sperrt den Datensatz
  - Prüft ob Abfrage bearbeiten erlaubt

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

Cursor – Beispiel

Beispiel

```
Vorlesungen(VorlNr, SWS):
INSERT INTO Vorlesungen VALUES (26120, 3), (27550,4);

CREATE FUNCTION reassign() RETURNS void AS $$
DECLARE
  c CURSOR FOR SELECT * FROM Vorlesungen FOR UPDATE;
  c2 refcursor;
  count integer = 0;
  row RECORD;
BEGIN
  -- nächste Folie
END; $$ LANGUAGE plpgsql;
```

Cursor – Beispiel (contd.)

Beispiel

```
FOR r IN c LOOP
  RAISE NOTICE '(%, %)', r.VorlNr, r.SWS;
  UPDATE Vorlesungen SET VorlNr=count WHERE CURRENT OF c;
  count = count + 1;
END LOOP;

OPEN c2 FOR SELECT * FROM Vorlesungen;
LOOP
  FETCH c2 INTO row;
  RAISE NOTICE '(%, %)', row.VorlNr, row.SWS;
  EXIT WHEN NOT FOUND;
END LOOP;
CLOSE c2;
```

Ausgabe: (26120, 3), (27550, 4), (0, 3), (1, 4)