Kapitel 5

PL/SQL

[1]PL/SQL (Procedural Language/Structured Query Language) ist eine proprietäre Programmiersprache der Firma Oracle und verbindet die Abfragesprache SQL mit einer prozeduralen Programmiersprache. Die Syntax wurde sehr stark an die Programmiersprache Ada angelehnt.

Unterstützt werden Variablen, Bedingungen, Schleifen und Ausnahmebehandlungen. Ab der Version 8 der Oracle-RDBMS halten auch objektorientierte Merkmale Einzug.

PL/SQL ist für das Arbeiten mit Oracle Datenbanken ausgelegt. Insbesondere kann man im Quelltext SQL-Befehle nach dem Oracle-Standard einfügen. Dabei werden die SQL-Anweisungen nicht als Zeichenketten erzeugt und an eine Datenbankschnittstelle übergeben (wie z. B. bei ODBC, JDBC u. ä.), sondern fügen sich nahtlos in den Programmcode ein. Die Korrektheit der SQL-Statements kann somit schon durch Kompilieren verifiziert werden, zumindest wenn diese statisch, also nicht erst zur Laufzeit erzeugt sind. Dies gilt jedoch nur für DML-Befehle. DDL und DCL müssen mit `dbms_sql` (einem PL/SQL-Package) oder mit der Syntax `execute immediate` <befehl> ausgeführt werden.

Die prozedurale Erweiterung der SQL-Abfragesprache wird inzwischen auch von vielen anderen Datenbankherstellern implementiert. Daher wurde diese prozedurale SQL-Erweiterung inzwischen vom ANSI-Gremium standardisiert.

Bei der Verwendung von PLSQL bieten sich folgende Möglichkeiten:

- Man kann PL/SQL-Code wie SQL-Befehle über ein Datenbank-Frontend absetzen, der dann direkt abgearbeitet wird.
- Man kann einzelne Unterprogramme (Stored Procedures) oder Bibliotheken mehrerer Unterprogramme (Stored Packages) als dauerhafte Datenbankobjekte auf dem Datenbankserver speichern und damit die Funktionalität der Datenbank erweitern; jeder Benutzer der Datenbank kann diese Unterprogramme aufrufen und nutzen. Die Berechtigungen können für jedes einzelne PL/SQL-Paket an einzelne Benutzer oder Benutzergruppen (sogenannte „Rollen“) vergeben werden.
- Programmierung von Datenbanktriggern
- Programmierung in diversen Tools (Oracle-Forms, Oracle-Reports)

5.1 Einführung

PL/SQL Programme werden über so genannte Blocks strukturiert. Jeder Block enthält dabei PL/SQL als auch normale SQL Statements. Ein typischer PL/SQL Block hat dabei folgenden Aufbau:

[1]Quelle: http://www.wikipedia.org

```
[DECLARE
      declaration_statements
]
BEGIN
      executable_statements
[EXCEPTION
      exception_handling_statements
]
END;
```

- die declaration und die exception section sind optional
- declaration_statements deklarieren Variablen die im restlichen Block verwendet werden
- diese Variablen sind nur in diesem Block gültig (Ausnahme bei geschachtelten Blöcken)
- die Deklarationen müssen am Start des Blocks erfolgen
- executable_statements können Wertzuweisungen, Kontrollstrukturen, Schleifen udgl. sein
- exception_handling_statements dienen der Fehlerbehandlung
- alle Anweisungen werden mit einem ';' abgeschlossen
- ein Block wird mit dem Schlüsselwort END abgeschlossen.

Beispiel:

```
DECLARE
        width INTEGER;
        height INTEGER := 2;
        area INTEGER;
BEGIN
        area := 6;
        width := area / height;
        DBMS_OUTPUT.PUT_LINE('width = ' || width);
EXCEPTION
        WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Division by zero');
END;
/
width = 3

PL/SQL procedure successfully completed.

SQL>
```

PL/SQL erlaubt es, Blöcke zu verschachteln. Variablen, die in äußeren Blöcken deklariert werden, sind in allen inneren Blöcken gültig. Variablen, die in inneren Blöcken deklariert werden, sind nicht in äußeren Blöcken gültig.

5.2 Deklarationsteil

In der Declaration Section werden die in dem entsprechenden Block verwendeten PL/SQL-Datentypen, PL/SQL-Variablen und Cursors deklariert.

PL/SQL-Variablen mit werden durch Angabe ihres Datentyps und einer optionalen Angabe eines Default-Wertes deklariert:

```
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
...
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
```

Anstatt `datatype`[2] direkt explizit anzugeben, kann man eine Typdeklaration machen, indem man angibt, mit welcher Variablen der Typ übereinstimmen soll:

```
<variable> <table>.<col>%TYPE [NOT NULL] [DEFAULT <value>];
<variable> <table>%ROWTYPE;
```

Im ersten Fall bekommt `variable` den Datentyp der Spalte `col` in der Tabelle `table`. Häufig werden Records/Strukturen benötigt, deren Typ mit dem Typ einer Zeile einer bestimmten Tabelle übereinstimmt. Solche Deklarationen können wie im zweiten Fall als `ROWTYPE`-Deklaration angegeben werden. Zusätzlich zu TYPE und ROWTYPE kann man in PL/SQL noch folgende Datentypen verwenden: `boolean, binary_integer, natural, positive`, ...

5.3 Anweisungsteil

5.3.1 Kontrollstrukturen

PL/SQL enthält die folgenden Kontrollstrukturen:

- ```
  IF condition THEN statement;
  ... statement;
  [ELSIF condition THEN statement; ... statement;] ...
  [ELSIF condition THEN statement; ... statement;]
  [ELSE statement; ... statement;]
  END IF;
  ```
- ```
  LOOP statement;
  ... statement;
  END LOOP;
  ```
- ```
  WHILE condition
  LOOP statement;
  ... statement;
  END LOOP;
  ```
- ```
  FOR loop-variable IN [REVERSE] lower-bound..upper-bound
  LOOP statement; ... statement;
  END LOOP;
  ```
  Die Variable `loop_index` wird dabei automatisch als `INTEGER` deklariert.
- Mit `EXIT [WHEN <bedingung>];` kann man einen LOOP jederzeit verlassen.
- GOTO-Befehl mit Labels:
  `≪label_i≫ ... GOTO label_j;`

5.3.2 Datenbankzugriff

- **Anfragen** `SELECT attr1 [INTO var1]`
  `FROM relation`
  `WHERE bedingung [FOR UPDATE OF attr1];` Die `FOR UPDATE`-Angabe bewirkt das Sperren der Tabelle. Die `INTO..`-Angabe darf nur bei einem Ergebnistupel verwendet werden. Dieses wird sofort einer Variablen zugeordnet (bei mehreren Ergebnistupeln: Cursor benutzen).

[2] erlaubte Datentypen: char,long,long raw, varchar, varchar2, date, number, string, character, boolean, float, int, integer, natural, decimal, dec, real, smallint, ...(vollständige Liste unter: Datentypen in PL/SQL)

- **Updates** `INSERT INTO relation VALUES (...);`
  `UPDATE relation SET ... WHERE bedingung;`
  `DELETE FROM relation WHERE bedingung;`

- **Prozedur- und Funktionsaufrufe** `procname(parameter1,...);`
  `var := fktname(parameter1,...);`

- **Cursor-Funktionen**

```
DECLARE
  CURSOR c IS SELECT * FROM weine;
  w weine%ROWTYPE;
BEGIN
  ...
  OPEN c; --jetzt erst wird die Anfrage gestellt
  ...
  FETCH c INTO w --holt naechstes Tupel der Anfrage z.B. in LOOP
  ...
  CLOSE c; --schliesst Cursor, Anfrageergebnis nicht mehr vorhanden
           --Erneutes OPEN nach CLOSE stellt Anfrage neu
  ...
END;
```

Beispiel:

Es existieren 3 Relationen `konto`, `abgelehnt` und `gebucht`:

```
CREATE TABLE konto (
  kontonr NUMBER,
  kontostand NUMBER,
  CONSTRAINT pk_konto PRIMARY KEY (kontonr)
);

CREATE TABLE abgelehnt (
  datum DATE,
  von NUMBER,
  nach NUMBER,
  betrag NUMBER
);

CREATE TABLE gebucht (
  datum DATE,
  von NUMBER,
  nach NUMBER,
  betrag NUMBER
);

INSERT INTO konto VALUES (1,1000);
INSERT INTO konto VALUES (2,2000);
INSERT INTO konto VALUES (3,3000);
```

Bei der Relation `konto` wurde ein neues Attribut `id` eingefügt:

```
ALTER TABLE konto
 ADD id NUMBER;
```

Nun sollen alle Tupeln mittels folgender Stored Procedure `kontoid` eine durchnumerierte `id` erhalten:

```
CREATE OR REPLACE PROCEDURE kontoid IS
  i NUMBER;
  kto konto%rowtype;
  CURSOR c IS
```

```
    SELECT * FROM konto FOR UPDATE; --mit Locking
BEGIN
  i := 1;
  OPEN c;
  LOOP
    FETCH c INTO kto;
    EXIT WHEN c%NOTFOUND;
    UPDATE konto SET id=i WHERE CURRENT OF c;
    -- oder:
    -- UPDATE konto SET id=i WHERE kontonr=kto.kontonr;
    i := i+1;
  END LOOP;
  CLOSE c;
END;
/
```

Für einen Cursor sind folgende Attribute definiert:
`c%FOUND`, `c%NOTFOUND`, `c%ISOPEN` (jeweils `TRUE` oder `FALSE`) und `c%ROWCOUNT` (Anzahl der Tupel)

zusätzlich zu den expliziten Cursorn des vorigen Beispiels besteht auch die Möglichkeit, cursor implizit zu deklarieren, wie das folgende Beispiel zeigt:

```
declare
  v_summe number := 0;
begin
  for v_auftrag_pos in (select * from auftrag_pos) loop
    v_summe := v_summe + v_auftrag_pos.anzahl * v_auftrag_pos.preis;
  end loop;
end;
```

5.4 Stored Procedures

Zusätzlich zu anonymen Blöcken können Sie in PL/SQL auch Prozeduren und Funktionen deklarieren:

```
CREATE OR REPLACE PROCEDURE procedure-name [(argument1 ... [, argumentN) ]
IS
[local-variable-declarations]
BEGIN
        executable-section
[exception-section]
END [procedure-name];
```

Die Variablen sind folgendermaßen definiert:

- procedure-name ist der Name der Prozedur, er unterliegt den Einschränkungen von Oracle-Datenbanken bezüglich der Namensgebung von Objekten.
- argument1 bis argumentN sind optionale Deklarationen von Argumenten, diese sind folgendermaßen aufgebaut:
- argument-name [IN | OUT] datatype [{:= | DEFAULT} value]
- local-variable-declarations sind optionale Deklarationen von Variablen, Konstanten und anderen Prozeduren, die lokal für procedure-name sind.
- executable-section sind die PL/SQL-Anweisungen, aus denen sich die Prozedur zusammensetzt.
- exception-section ist der optionale Abschnitt zur Behandlung von Exceptions in der Prozedur.

Die Unterscheidung zwischen gespeicherten Prozeduren und Prozeduren, die in anonymen Blöcken deklariert und verwendet werden, ist wichtig. Die Prozeduren, die in anonymen Blöcken deklariert und aufgerufen werden, sind temporär; wenn die Ausführung der anonymen Blöcke beendet ist, sind sie für Oracle nicht mehr vorhanden. Eine gespeicherte Prozedur, die mit create procedure erzeugt wurde oder in einem Paket enthalten ist, ist permanent in dem Sinn, dass sie von einem SQL*Plus-Skript, einem PL/SQL-Unterprogramm oder einem Datenbanktrigger aufgerufen werden kann.

Beispiel 1 - Stored Procedure ohne Parameter

Definition der Stored Procedure:[3]

```
CREATE OR REPLACE PROCEDURE HalloWelt
IS
BEGIN
  DBMS_OUTPUT.PUT('Hallo');
  DBMS_OUTPUT.PUT_LINE('Welt');
END;
/
```

Ausführen der Stored Procedure:[4]

```
EXECUTE HalloWelt;
```

Beispiel 2 - Stored Procedure mit einem Parameter

```
CREATE OR REPLACE PROCEDURE HalloWelt2(anzahl INTEGER)
IS
  zaehler INTEGER DEFAULT 1;
BEGIN
  WHILE zaehler<=anzahl
  LOOP
    DBMS_OUTPUT.PUT_LINE('Hallo Welt '||zaehler);
    zaehler:=zaehler+1;
  END LOOP;
END;
/
```

Ausführen der Stored Procedure:

```
EXECUTE HalloWelt2(10);
```

Beispiel 3 - Stored Procedure mit mehreren Parametern

```
CREATE OR REPLACE PROCEDURE HalloWelt3(text VARCHAR, anzahl INTEGER)
IS
BEGIN
  FOR zaehler IN 1 .. anzahl
  LOOP
    DBMS_OUTPUT.PUT_LINE(text||' '||zaehler);
  END LOOP;
END;
/
```

Ausführen der Stored Procedure:

```
EXECUTE HalloWelt3('INSY',10);
```

Beispiel 4

Definition der Stored Procedure:

[3] Fehler bei der Definition können mittles `SQL> SHOW ERROR` angezeigt werden.
[4] Die Ausgabe muss evenutell mit `SET SERVEROUTPUT ON` eingeschaltet werden.

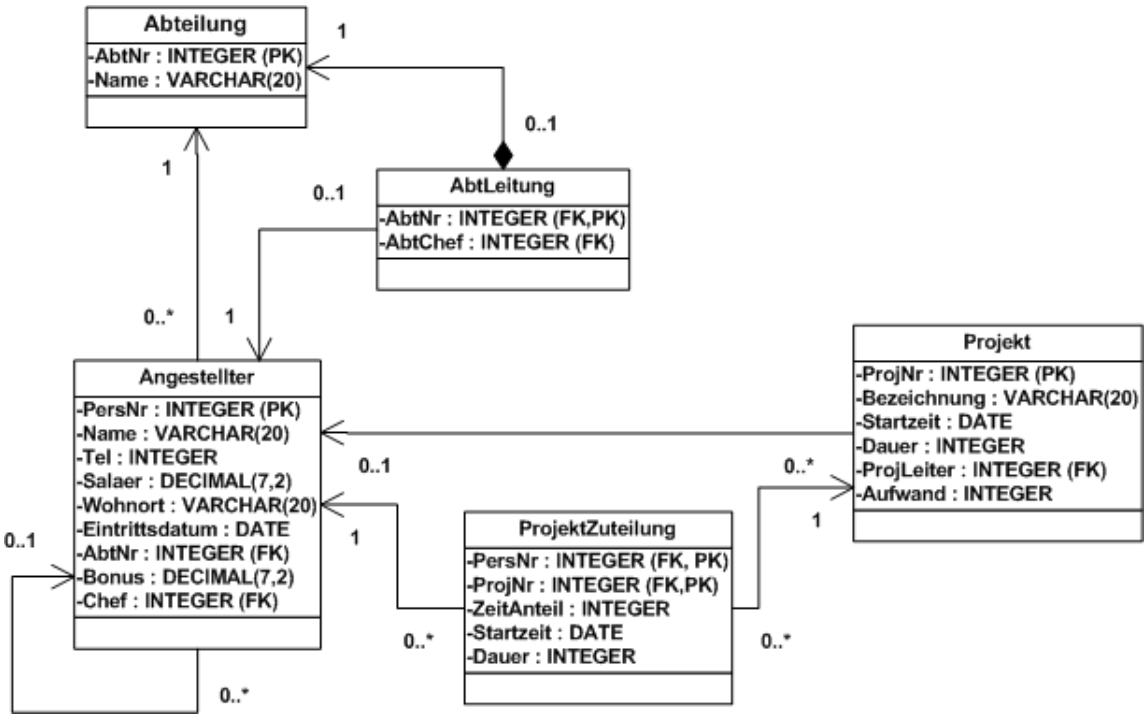
```
CREATE OR REPLACE PROCEDURE ueberweisen(x NUMBER, y NUMBER, betrag NUMBER)
IS
  s konto.kontostand%TYPE;  -- Deklarationsteil
BEGIN
  SELECT kontostand INTO s FROM konto
    WHERE kontonr= x FOR UPDATE OF kontostand;

  IF (s<betrag) THEN
    INSERT INTO abgelehnt VALUES (SYSDATE,x,y,betrag);
  ELSE
     s:=s-betrag;
    UPDATE konto
      SET kontostand=s WHERE kontonr=x;
    UPDATE konto
      SET kontostand=kontostand+betrag WHERE kontonr=y;
    INSERT INTO gebucht VALUES (SYSDATE,x,y,betrag);
  END IF;
  COMMIT;
END;
/
```

Ausführen der Stored Procedure:

```
EXECUTE ueberweisen(1,2,500);
```

Aufgabe

Als Grundlage für manche der folgenden Aufgaben wird folgende DB verwendet (UML-Syntax):



`A1` Schreiben Sie eine Stored Procedure ProjektZuteilen in PL/SQL:

- Input
  - PK Angestellter

  - PK Projekt
  - Prozentuale Arbeitszeit
  - Startzeit (default: jetzt)

- Vorbedingungen:
  - Prozentuale Arbeitszeit zwischen 10 und 90
  - Totale prozentuale Arbeitszeit <= 100
  - Angestellter ist dem Projekt noch nicht zugeordnet

- Operation:
  - Eintrag eines Tupels in Tabelle Projektzuteilung

- Output:
  - Resultat Code:
    * -1: Prozentanteil nicht zwischen 10 und 90
    * -2: Total grösser als 100%
    * -3: Angestellter schon in Projekt
    * -4: Angestellter oder Projekt nicht vorhanden

A2: Erstellen Sie einen PL/SQL Block, welcher die fünf bestverdienenden Angestellten in die folgende, neue Tabelle Top5 füllt.

Benutzen Sie für Ihre Lösung einen Cursor.

```
CREATE TABLE Top5 (
Name VARCHAR2(20),
Persnr NUMBER,
Salaer DECIMAL(7,2) );
```

5.5 Ausnahmebehandlung von Fehlersituationen

Die beim Auftreten einer Exception auszuführenden Aktionen werden in der Exception Section definiert. Exceptions können dann an beliebigen Stellen des PL/SQL-Blocks durch RAISE ausgelöst werden.

```
DECLARE
 <exception1> EXCEPTION;
   ...
BEGIN
   ...
   IF ... THEN RAISE <exception1>;
   ...
EXCEPTION
   WHEN <exception1> THEN <PL/SQL-Statement>;
   WHEN <exception2> THEN <PL/SQL-Statement>;
    [WHEN OTHERS THEN PL/SQL-Statement]
END;
```

Wird eine Exception ausgelöst, so wird die in der entsprechenden WHEN-Klausel aufgeführte Aktion ausgeführt und der innerste Block verlassen. Ist für eine Fehlermeldung keine Aktion in der Exception-Section angegeben, stattdessen aber eine Aktion unter `WHEN OTHERS` angegeben wird diese ausgeführt.

Neben eigenen Exceptions gibt es auch vordefinierte Exceptions. Wenn beispielsweise `SELECT * INTO ... FROM ... WHERE ...;` kein oder mehr als ein Tupel liefert, wird die Exception `NO_DATA_FOUND` bzw. `TOO_MANY_ROWS` ausgelöst[5]:

```
...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    ...;
  WHEN TOO_MANY_ROWS THEN
    ...;
  WHEN OTHERS THEN
    ...;
END;
```

Zum Auslösen vordefinierter Fehlermeldungen dient `RAISE_APPLICATION_ERROR`[6].

5.6 PL/SQL-Funktionen

Aufbau einer Funktion:

```
CREATE [OR REPLACE] FUNCTION fktname [(par1,...)] RETURN typ
IS
  ... --lokale Deklarationen
BEGIN
  ...
  RETURN wert;
EXCEPTION
  ...
END;
```

Aufbau der Parameter `par1,...`:
`parametername [IN | OUT | IN OUT ] typ [ :=expr | DEFAULT expr ]`
voreingestellt ist `IN`; `DEFAULT`-Angabe nur bei IN erlaubt

Beispiel:
Die folgende Funktion soll die Anzahl der Kontobewegungen eines bestimmten Kontos berechnen;

```
CREATE OR REPLACE FUNCTION kontobewegungen(kontonummer NUMBER DEFAULT 1)
RETURN NUMBER -- Anzahl der Kontobewegungen soll zurueckgegeben werden
IS
  anz1 NUMBER; --Anzahl der Abbuchungen
  anz2 NUMBER; --Anzahl der Gutschriften
BEGIN
  SELECT COUNT(*) INTO anz1 FROM gebucht WHERE von=kontonummer;
  SELECT COUNT(*) INTO anz2 FROM gebucht WHERE nach=kontonummer;
  RETURN anz1+anz2;
END;
/
```

Der Aufruf einer Funktion kann nur innerhalb einer SQL-Abfrage erfolgen:

```
SELECT kontobewegungen(kontonr)
FROM konto;
```

Soll eine Funktion nur einmal aufgerufen werden, kann dazu die "Dummy"-Relation `dual` benutzt werden, welche genau ein Tupel besitzt.

[5] Weitere in Oracle vordefinerte Exceptions: `LOGIN_DENIED`, `NOT_LOGGED_ON`, `CURSOR_ALREADY_OPEN`, `INVALID_CURSOR`, `INVALID_NUMBER`, `DUP_VAL_ON_INDEX`, `ROWTYPE_MISMATCH`, `STORAGE_ERROR`, `TIMEOUT_ON_RESOURCE`, `VALUE_ERROR`, `PROGRAM_ERROR`, `ZERO_DIVIDE`

[6] siehe Abschnitt 5.7

```
SELECT kontobewegungen(3) FROM dual;
SELECT kontobewegungen FROM dual;
```

5.7 Trigger

Trigger sind eine spezielle Form von PL/SQL-Prozeduren, die beim Eintreten eines bestimmten Ereignisses vom System automatisch ausgeführt werden. In Oracle gibt es verschiedene Arten von Triggern: BEFORE- und AFTER-Trigger[7]

Ein BEFORE- oder AFTER-Trigger ist einer Tabelle (oft auch noch einer bestimmten Spalte) zugeordnet. Seine Bearbeitung wird durch das Eintreten eines Ereignisses (Einfügen, Ändern oder Löschen von Zeilen der Tabelle) ausgelöst. Die Ausführung eines Triggers kann zusätzlich von Bedingungen an den Datenbankzustand abhängig gemacht werden. Weiterhin unterscheidet man, ob ein Trigger vor oder nach der Ausführung der entsprechenden aktivierenden Anweisung in der Datenbank ausgeführt wird. Ein Trigger kann einmal pro auslösender Anweisung (Statement-Trigger) oder einmal für jede betroffene Zeile (Row-Trigger) seiner Tabelle ausgeführt werden.

Da Trigger im Zusammenhang mit Veränderungen an (Zeilen) der Datenbasis verwendet werden, gibt es die Möglichkeit im Rumpf den alten und neuen Wert des gerade behandelten Tupels zu verwenden.

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
BEFORE | AFTER
   {INSERT | DELETE | UPDATE} [OF <column-list>]
   [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
   ...
   [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
ON <table>
[REFERENCING OLD AS <name> NEW AS <name>]
[FOR EACH ROW]
[WHEN (<trigger-condition>)]
<pl/sql-block>;
```

Erklärung:

- Mit `BEFORE` und `AFTER` wird angegeben, ob der Trigger vor oder nach Ausführung der auslösenden Operation (einschließlich aller referentiellen Aktionen) ausgeführt wird.
- Die Angabe von `OF <column>` ist nur für `UPDATE` erlaubt. Wird `OF <column>` nicht angegeben, so wird der Trigger aktiviert wenn irgendeine Spalte eines Tupels verändert wird.
- Mittels der in `REFERENCING OLD AS ... NEW AS ...` angegebenen Transitions-Variablen kann auf die Zeileninhalte vor und nach der Ausführung der aktivierenden Aktion zugegriffen werden. Als Default erreicht man diese Werte unter `:OLD` bzw. `:NEW`.
- Schreibzugriff auf `:NEW`-Werte ist nur mit `BEFORE`-Triggern erlaubt.
- `FOR EACH ROW` definiert einen Trigger als Row-Trigger. Fehlt diese Zeile, wird der Trigger als Statement-Trigger definiert.
- Mit `WHEN` kann die Ausführung eines Triggers weiter eingeschränkt werden. Insbesondere können die Transitionsvariablen `OLD` und `NEW` in der `WHEN`-Bedingung verwendet werden.
- Der `<pl/sql-block>` eines Triggers darf keine Befehle zur Transaktionskontrolle (`commit/rollback`) enthalten.

[7] Weiters gibt es seit Version 8 auch noch INSTEAD OF-Trigger, welche für die Behandlung von Views verwendet werden können.

- Ist ein Trigger für verschiedene Ereignisse definiert, kann das auslösende Ereignis im Rumpf durch `IF INSERTING THEN`, `IF UPDATING OF <column-list> THEN` usw. abgefragt werden.

Beispiel 1:

Ein Versandhaus verwaltet den Lagerbestand in einer Relation bestand. Sollen bestimmte Artikel von einer Zulieferfirma nachbestellt werden, so wird ein Eintrag in der Relation bestellung fällig. Dazu existieren folgende 2 Tabellen:

```
CREATE TABLE bestand (
  artNr NUMBER(10),
  anzahl NUMBER(10),
  mindAnzahl NUMBER(10),
  CONSTRAINT pk_bestand PRIMARY KEY (artNr)
);

CREATE TABLE bestellung (
  datum DATE,
  artNr NUMBER(10)
);
```

Es soll nun ein Trigger `nachbestellung` erstellt werden, der eine automatische Bestellung auslöst, falls der aktuelle Lagerbestand eines Artikels unter seinem Eintrag `mindAnzahl` liegt.

```
CREATE TRIGGER nachbestellung
AFTER UPDATE OF anzahl ON bestand
FOR EACH ROW
BEGIN
   IF :new.anzahl < :new.mindAnzahl THEN
      INSERT INTO bestellung VALUES (SYSDATE, :new.artNr);
   END IF;
END;
/
```

Beispiel 2:

Dieses Beispiel soll demonstrieren, wie mittels Trigger Exceptions ausgelöst werden können. Betrachten wir nochmals die Relation `konto` aus einem vorangegangen Beispiel. Wenn es nicht erlaubt ist, dass Bankkunden ihr Konto überziehen, so muss stets deren Kontostand kontrolliert werden:

```
CREATE OR REPLACE TRIGGER kontoueberziehung
BEFORE UPDATE OF kontostand ON konto
FOR EACH ROW
BEGIN
   IF :new.kontostand < 0 THEN
      raise_application_error(-20001,'Keine Kontoueberziehung erlaubt');
   END IF;
END;
/
```

oder alternativ:

```
CREATE OR REPLACE TRIGGER kontoueberziehung2
BEFORE UPDATE OF kontostand ON konto
FOR EACH ROW
WHEN (new.kontostand<0)
BEGIN
   raise_application_error(-20001,'Keine Kontoueberziehung erlaubt');
END;
/
```

Natürlich kann so eine Exception von einem Trigger während einer Transaktion abgefangen werden. Wenn dieser Trigger schon am Beginn dieses Kapitels existiert hätte, so hätte die Stored Procedure `ueberweisen` auch auf folgende Art definiert werden können:

```
CREATE OR REPLACE PROCEDURE ueberweisen(x NUMBER, y NUMBER, betrag NUMBER)
IS
BEGIN
  UPDATE konto SET kontostand=kontostand+betrag WHERE kontonr=y;
  UPDATE konto SET kontostand=kontostand-betrag WHERE kontonr=x;
  INSERT INTO gebucht VALUES (SYSDATE,x,y,betrag);
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    INSERT INTO abgelehnt VALUES (SYSDATE,x,y,betrag);
    COMMIT;
END;
/
```

5.8 Aufgaben

5.8.1 Entfernungsberechnungen

Beispiel 1

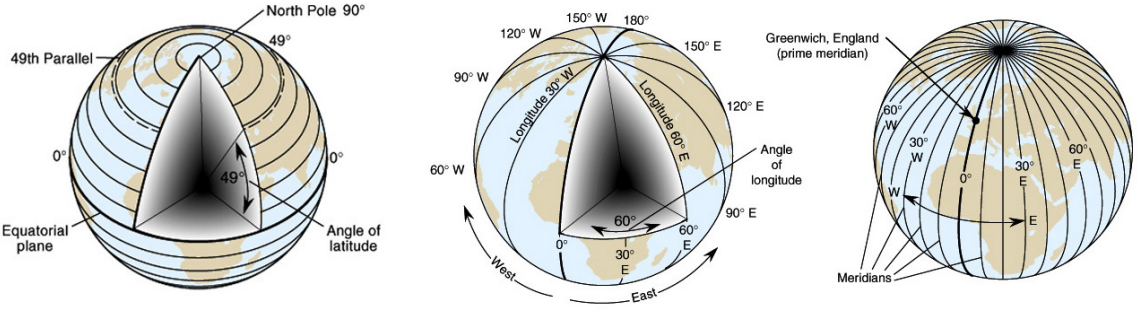
Testen Sie sämtliche Stored Procedures, Funktionen und Trigger aus diesem Kapitel.

Beispiel 2

Erstellen Sie eine Relation `city` wie folgt und fügen Sie einige Beispiel-Tupel ein.

```
CREATE TABLE city (
  name VARCHAR2(35),
  country VARCHAR2(4),
  province VARCHAR2(32),
  population NUMBER CONSTRAINT check_CityPop
    CHECK (population >= 0),
  longitude NUMBER CONSTRAINT check_CityLon
    CHECK ((longitude >= -180) AND (longitude <= 180)),
  latitude NUMBER CONSTRAINT check_CityLat
    CHECK ((latitude >= -90) AND (latitude <= 90)),
  CONSTRAINT pk_City PRIMARY KEY (name, country, province)
);

INSERT INTO city VALUES ('Athens', 'GR', 'Greece', 885737, 23.7167, 37.9667);
INSERT INTO city VALUES ('Dublin', 'IRL', 'Ireland', 502337, -6.35, 53.3667);
INSERT INTO city VALUES ('Linz', 'A', 'Upper Austria', 203000, 14.18, 48.18);
INSERT INTO city VALUES ('St. Polten', 'A', 'Lower Austria', 51102, 15.38, 48.13);
...
```

Erstellen Sie eine Funktion `distance(Latitude1, Longitude1, Latitude2, Longitude2)`, welche den Abstand von 2 Städten berechnet und als Rückgabewert liefert.



Hinweis zur Berechnung:

$$D = acos(sin(a) * sin(b) + cos(a) * cos(b) * cos(P))$$

[3mm]

D ... angular distance between points A and B
a ... latitude of point A
b ... latitude of point B
P ... longitudinal difference between points A and B

[2mm]

Der Abstand in km kann durch Multiplikation des Winkels D (in Grad) mit 111km berechnet werden.[8]

[8] Dies ist nur eine Näherung, da in diesem Fall die Erde als Kugel betrachtet wird.

Testen Sie anschließend diese Funktion folgendermaßen:

- Abstand 'Athens' - 'Vienna':

- Abstand aller österreichischen Orte voneinander:

```
SELECT distance(c1.latitude,c1.longitude,c2.latitude,c2.longitude) as distance
FROM city c1, city c2
WHERE c1.country='A' AND c2.country='A' AND c1.name<c2.name
ORDER BY by distance;
```

Beispiel 3

Fügen Sie zur Relation `city` ein neues Attribut `cityid` hinzu. Es soll dazu dienen, alle Städte zu numerieren. Schreiben Sie eine PL/SQL-Prozedur um diese Numerierung durchzuführen.

Beispiel 4

Erzeugen Sie eine Relation `distances` mit den Attributen `cityid1`, `cityid2` und `distance`, welche die Entfernung aller Städte von einander aufnehmen soll. Schreiben Sie anschließend eine PL/SQL-Prozedur, um alle Tupel einzufügen[9]. Sind die Koordinaten einer Stadt nicht bekannt (NULL-Werte), so braucht diese Stadt nicht berücksichtigt werden.

Beispiel 5

Im Beispiel 3 wurde ein Attribut `cityid` eingeführt. Dieses Attribut sollte eindeutig sein. Üblicherweise werden eindeutige Attribute mit dem Schlüsselwort `UNIQUE` versehen, um diese Integritätsbedingung beim Einfügen neuer Datensätze zu erfüllen.
In diesem Beispiel soll das Schlüsselwort `UNIQUE` absichtlich nicht verwendet werden. Programmieren Sie einen Trigger, der diese Integritätsbedingung überwacht: Prüfen Sie vor dem Einfügen eines Datensatzes, ob die neue `cityid` bereits in der Relation vorhanden ist. Wenn sie noch nicht vorkommt, dann soll der neue Datensatz eingefügt werden, ansonsten eben nicht.
Testen Sie die Wirkungsweise des Triggers, mit einem gültigen und einem ungültigen INSERT-Statement.

5.8.2 Reiseveranstalter

Ein kleiner Veranstalter, der Tagesausflüge zu beliebten Zielen in Österreich anbietet, möchte seine Verwaltung effizienter gestalten. Dafür soll unter anderem eine Datenbank angelegt werden. Diese enthält folgende Informationen:

- Von allen Mitarbeitern werden Vorname ("vname"), Nachname ("nname") und das Monatsgehalt ("gehalt") gespeichert. Zur eindeutigen Identifizierung wird die Sozialversicherungsnummer ("svnr") vermerkt. Reiseleiter und Fahrer sind spezielle Mitarbeiter. Bei den Reiseleitern wird vermerkt welche Sprache(n) sie beherrschen und wie gut sie darin sind ("niveau"). Das Niveau kann dabei die Werte "A1", "A2", "B1", "B2", "C1" und "C2" (entsprechend den standardisierten Sprachlevels von "Anfänger" (A1) bis "annähernd Muttersprache" (C2)) annehmen. Jede Sprache hat eine eindeutige Bezeichnung ("bezeichnung"). Bei den Fahrern wird zusätzlich vermerkt, welcher Fahrer für welche Kollegen im Krankheitsfall zur Vertretung eingeteilt ist.

[9] Alternativ würde sich für diese Aufgabe ein INSERT INTO ... SELECT ... - Statement anbieten.

- Der Veranstalter bietet ein Sortiment an Ausflügen an. Jeder Ausflug wird durch eine Identifikationsnummer ("aid") eindeutig gekennzeichnet, hat eine Bezeichnung ("bezeichnung"), eine bestimmte Dauer in (ganzzahligen) Stunden ("dauer"), einen Preis ("preis"), eine textuelle Beschreibung ("beschreibung") und eine Streckenlänge ("km"). Bei jedem Ausflug werden ein oder mehrere Orte besucht. Zu jedem Ort wird der Name gespeichert. Da Ortsnamen jedoch nicht unbedingt eindeutig sind, wird zur Identifizierung ein künstlicher Schlüssel ("oid") angelegt. An jedem Ort gibt es mehrere Sehenswürdigkeiten zu bestaunen. Diese haben eine eindeutige Bezeichnung ("bezeichnung") sowie eine Beschreibung ("beschreibung"). Jede Sehenswürdigkeit gehört zu genau einem Ort. Jeder Ort wiederum hat genau eine Sehenswürdigkeit, welche das Wahrzeichen des Ortes darstellt. (zum Beispiel Goldenes Dachl in Innsbruck oder Lindwurm in Klagenfurt).

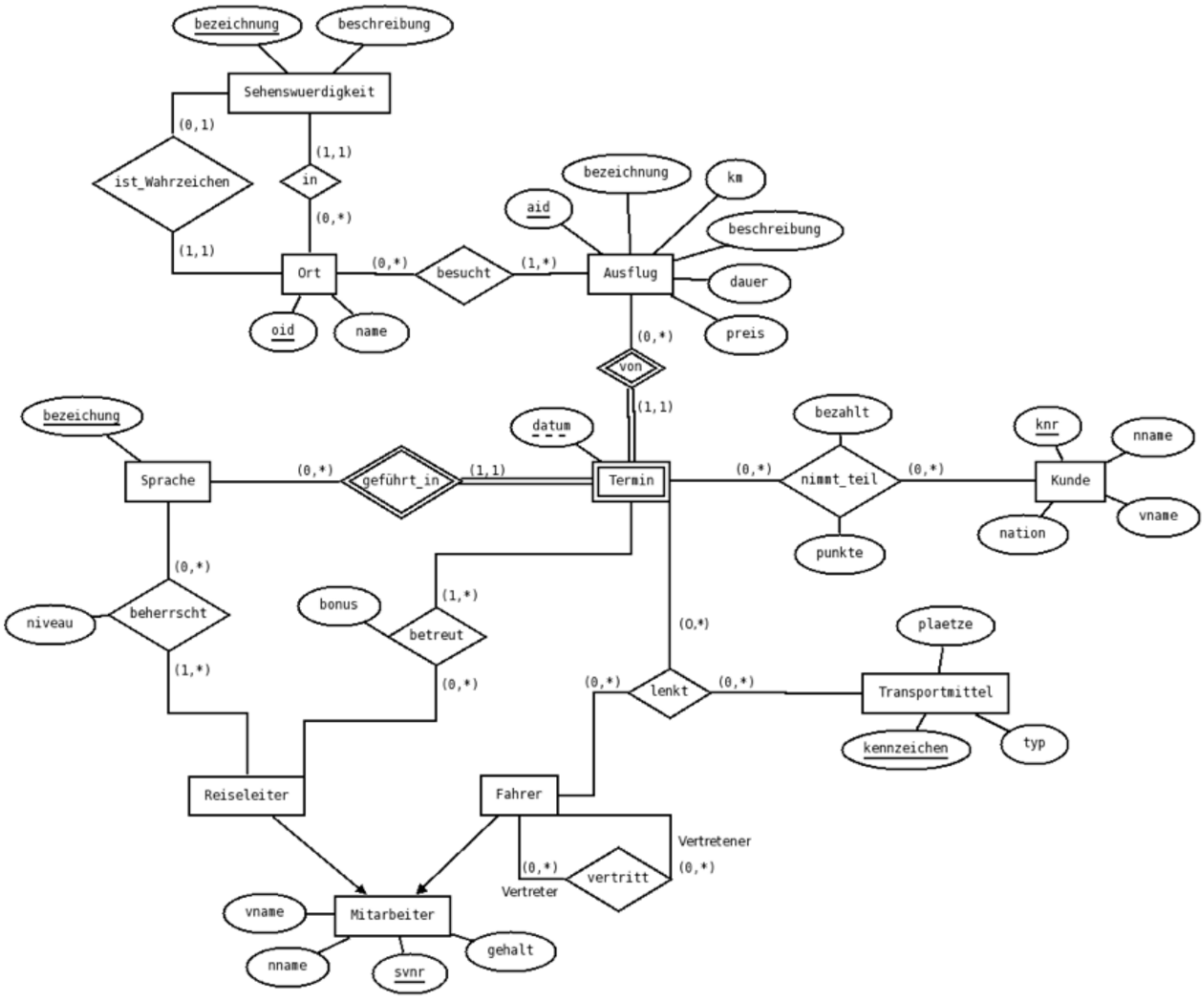
- Jeder Ausflug wird zu mehreren Terminen angeboten. Zu jedem Termin wird das entsprechende Datum ("datum") gespeichert. Weiters wird vermerkt in welcher Sprache (bzw. welchen Sprachen) ein Termin geführt wird. Ein Termin wird eindeutig identifiziert über den Ausflug der an diesem Termin gemacht wird, das Datum an dem er stattfindet, sowie die Sprache in der die Führung abgehalten wird. Diese Ausflugstermine werden jeweils von einem oder mehreren Reiseleitern betreut, die dafür einen Bonus ("bonus") erhalten.

- Kunden buchen Ausflugstermine an denen sie teilnehmen wollen. Natürlich kann jeder Kunde an beliebig vielen Ausflügen teilnehmen. Ein Kunde wird durch eine Kundennummer ("knr") identifiziert. Weiters werden sein Vor- ("vname") und Nachname ("nname"), sowie seine Nationalität ("nation") vermerkt. Zu jedem gebuchten Ausflug kann als gespeichert werden, ob der Kunde ihn bereits bezahlt hat ("bezahlt"), wobei "bezahlt" nur die Werte "JA" und "NEIN" annehmen darf. Weiters wird eine Bewertung des Kunden ("punkte") vermerkt. Jeder Kunde kann 1 (sehr schlecht) bis maximal 6 (ausgezeichnet) Punkte vergeben, mit denen er beurteilt wie gut ihm ein Ausflug gefallen hat. Bei fehlenden Bewertungen wird der Wert 0 eingetragen.

- Der Ausflugsveranstalter verwaltet auch eine Reihe von Transportmitteln wie PKWs und Busse. Von jedem Transportmittel ist sein eindeutiges Kennzeichen ("kennzeichen"), der Typ ("typ"), sowie eine Sitzplatzanzahl ("plaetze") bekannt. Es wird gespeichert welcher Fahrer welches Transportmittel bei welchem Ausflugstermin lenkt. Bei jedem Ausflugstermin können beliebig viele Transportmittel verwendet und Fahrer eingeteilt werden.

Sie finden nun im folgenden Bild ein ER-Diagramm, das die grobe Struktur der Datenbank abbildet, wobei Sie davon ausgehen können, dass die Abbildung sowohl den Sachverhalt als auch alle Kardinalitäten bereits korrekt abdeckt.

ER - Diagramm:



Relationenschema:

Mitarbeiter (svnr, vname, nname, gehalt)
Reiseleiter (svnr:*Mitarbeiter.svnr*)
Fahrer (svnr:*Mitarbeiter.svnr*)
vertritt (vertreter:*Mitarbeiter.svnr*, vertretener:*Mitarbeiter.svnr*)

Sprache (bezeichnung)
beherrscht (svnr:*Reiseleiter.svnr*, sprache:*Sprache.bezeichnung*, niveau)

Ausflug (aid, bezeichnung, beschreibung, preis, dauer, km)
Ort (oid, name, wahrzeichen:*Sehenswuerdigkeit.bezeichnung*)
Sehenswuerdigkeit (bezeichnung, beschreibung, oid:*Ort.oid*)
besucht (aid:*Ausflug.aid*, oid:*Ort.oid*)

Termin (datum, aid:*Ausflug.aid*, sprache:*Sprache.bezeichnung*)
betreut (svnr:*Reiseleiter.svnr*, datum:*Termin.datum*, aid:*Termin.aid*, sprache:*Termin.sprache*, bonus)

Kunde (knr, vname, nname, nation)
nimmt_teil (knr:*Kunde.knr*, datum:*Termin.datum*, aid:*Termin.aid*, sprache:*Termin.aid*, bezahlt, punkte)

Transportmittel (kennzeichen, typ, plaetze)
lenkt (svnr:*Fahrer.svnr*, kennzeichen:*Transportmittel.kennzeichen*, datum:*Termin.datum*, aid:*Termin.aid*, sprache:*Termin.sprache*)

Die notwendigen SQL-Dateien zum Anlegen der Tabellen sowie für das Einfügen von Testdaten finden Sie in Moodle.

Aufgabe 1. Trigger

Schreiben Sie einen BEFORE INSERT ON - Trigger ("t_before_betreut"), welcher vor einem Insert in die Tabelle "betreut" die sprachliche Qualifikation des Reiseleiters überprüft. Ein Datenbankeintrag soll nur dann stattfinden, wenn der betreffende Reiseleiter die Sprache, in der der Termin geführt wird, auch tatsächlich beherrscht. Falls der Reiseleiter die Sprache gar nicht beherrscht (es existiert also kein Datenbankeintrag), so soll die von Ihnen definierte Exception "qualifikation_fehlt" geworfen werden. Sollte zwar ein Eintrag existieren, der Reiseleiter die Sprache aber nur auf dem Niveau "A1" oder "A2" beherrschen, so werfen sie bitte eine Exception "qualifikation_zu_gering". Verzichten Sie auf eine Fehlerbehandlung innerhalb des Triggers (da ansonsten ein fehlerhaftes INSERT-Kommando als erfolgreich angesehen würde).

Aufgabe 2. Funktion f_bewertung

Schreiben Sie eine Funktion ("f_bewertung"), welche für einen Ausflugstermin die durchschnittliche Bewertung der Kunden berechnet: Die Funktion erhält als Eingabeparameter "a_datum" ("datum" des Termins), "a_aid" ("aid" des Ausflugs) sowie "a_sprache" ("bezeichnung" der Sprache). Die Berechnung der durchschnittlichen Bewertung soll folgendermaßen erfolgen:

- Zunächst wird die Summe über alle Punkte, die Kunden für diesen Termin vergeben haben, gebildet und der Wert anschließend durch die Anzahl jener Kunden, die eine Bewertung abgegeben haben, dividiert. Beachten Sie, dass der Wert "0" in der Spalte "punkte" bedeutet, dass von dem entsprechenden Kunden keine Bewertung abgegeben wurde. Dieser Wert soll daher nicht in die Bewertung einfließen!
- Falls ein Ausflugstermin von keinem einzigen Teilnehmer bewertet wurde, so soll als durchschnittliche Bewertung "3" angenommen werden. Achtung: Ausflugstermine, an denen niemand teilnimmt, sollen mit "0" bewertet werden!
- Weiters soll die Dauer des Ausflugs in die Bewertung einfließen: Die zuvor ermittelten durschnittlichen Punkte werden mit der Dauer des Ausflugs multipliziert.
- Der daraus berechnete Wert soll von der Funktion zurückgeliefert werden.

Aufgabe 3. Funktion f_bonus

Schreiben Sie eine weitere Funktion ("f_bonus"). Diese berechnet den Bonus, welcher einem Reiseleiter für einen von ihm betreuten Ausflugstermin zusteht. (Bei Terminen, die von mehreren Reiseleitern betreut werden, erhält jeder den selben Bonus.)

- Als Eingabe erhält die Funktion die Bewertung eines Ausflugtermins ("bewertung").
- Die Bonushöhe ist wie folgt abgestuft:
  - bewertung < 10: 0 Euro
  - 10 <= bewertung < 20: 10 Euro
  - 20 <= bewertung < 30: 30 Euro
  - 30 <= bewertung < 40: 70 Euro
  - 40 <= bewertung < 50: 110 Euro
  - bewertung >= 50: 150 Euro werden vergeben.
- Der ermittelte Bonus wird von der Funktion zurückgeliefert.

Werfen Sie eine Exception "ungueltiger_wert", sollte der Wert von "bewertung" kleiner als "0" sein und geben Sie auf dem Bildschirm eine sinnvolle Fehlermeldung aus.

Aufgabe 4. Prozedur p_bonus_eintragen

Schreiben sie eine Prozedur ("p_bonus_eintragen"), welche in der Tabelle "betreut" die Boni der Reiseleiter für von ihnen betreute Ausflugstermine folgendermaßen einträgt:

- Der Prozedur wird Beginn und Ende jenes Zeitraums übergeben, für welchen die Berechnung durchgeführt werden soll. ("beginn" und "ende" sind jeweils vom Typ DATE)
- Die Prozedur berechnet alle den Reiseleitern zustehenden Boni für den angegebenen Zeitraum, was durch den Aufruf der Funktionen "f_bewertung" und "f_bonus" erreicht wird. Nur falls der bereits in der Tabelle vermerkte Bonus kleiner als der neu berechnete Wert ist, soll dieser überschrieben werden. Zu große Boni sollen unverändert bleiben. (Damit bleibt die Möglichkeit erhalten, Reiseleitern auch mehr zu zahlen, als ihnen durch die Bonusberechnung zustehen würde.)
- Stellen Sie im Exception-Teil einen WHEN OTHERS Zweig bereit, der bei einem unvorhergesehenen Fehler die Oracle-Fehlernummer und Oracle-Fehlermeldung ausgibt. Außerdem müssen Sie durch entsprechende Transaction Control Kommandos sicherstellen, dass entweder ALLE gewünschten Änderungen in der Datenbank festgeschrieben werden, oder diese unverändert bleibt.

Aufgabe 5. Prozedur p_entferne_nichtzahler

Schreiben Sie eine weitere Prozedur ("p_entferne_nichtzahler"), welche dazu dient, die Buchungen der Teilnehmer aus der Datenbank zu löschen, die für einen Ausflugstermin angemeldet sind, diesen aber nicht bezahlt haben.

- Die Prozedur erhält aus Input-Parameter ein Datum("input_datum").
- Nun sollen zu allen Ausflugsterminen, die vor "input_datum" liegen, die Buchungen jener Kunden gelöscht werden, die den Ausflug noch nicht bezahlt haben.
- Wird bei "input_datum" kein Wert (d.h. NULL) übergeben, so soll das aktuelle Datum (SYSDATE) stattdessen verwendet werden.
- Stellen Sie (wie schon bei "p_bonus_eintragen") im Exception-Teil einen WHEN OTHERS Zweig bereit, der bei einem unvorhergesehenen Fehler die Oracle-Fehlernummer und Oracle-Fehlermeldung ausgibt. Außerdem müssen Sie durch entsprechende Transaction Control Kommandos sicherstellen, dass entweder ALLE gewünschten Änderungen in der Datenbank festgeschrieben werden, oder diese unverändert bleibt.

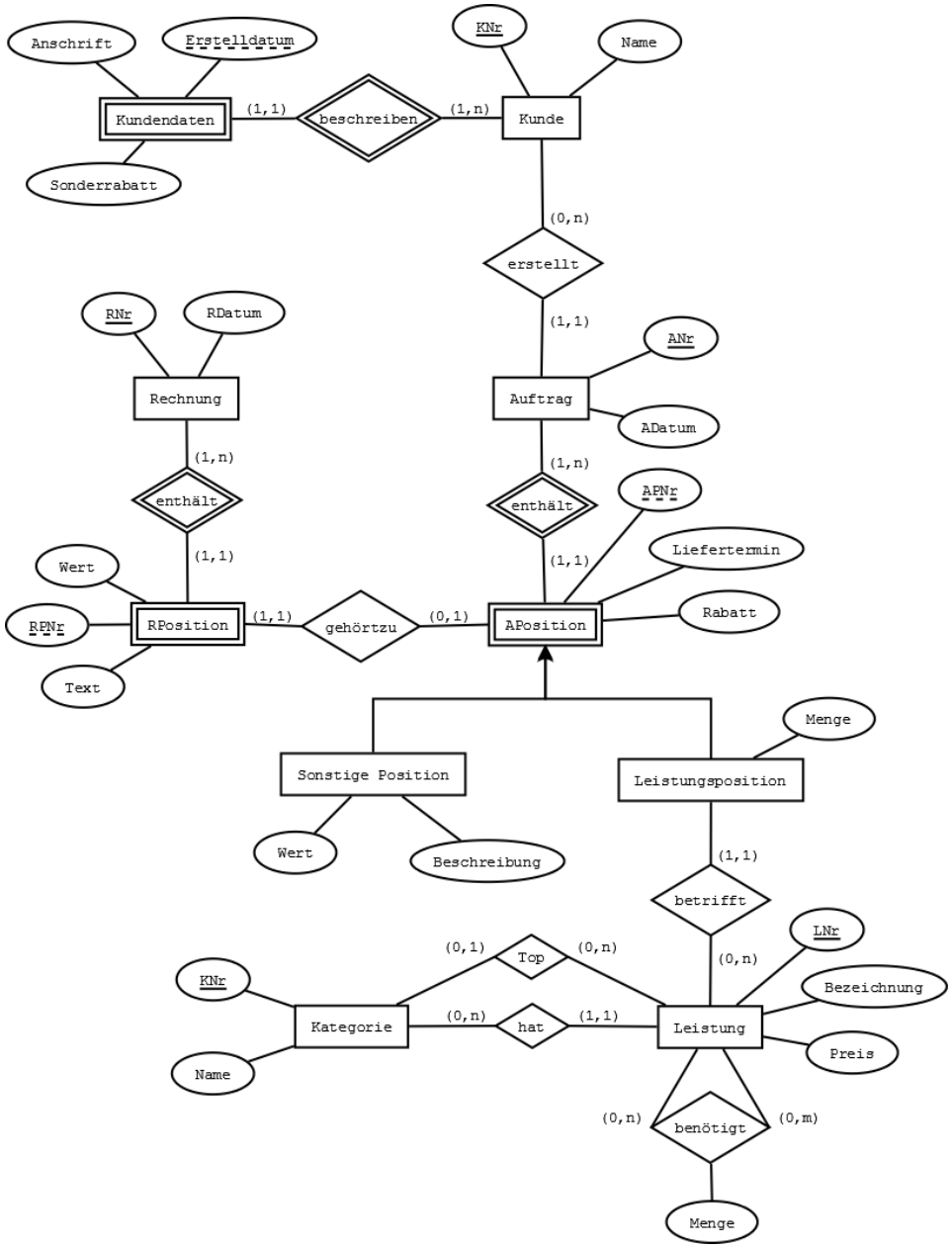
5.8.3 Eventtechnikfirma

Eine Eventtechnikfirma möchte seine Auftragsverwaltung in einer Datenbank erfassen. Dazu sollen folgende Informationen gespeichert werden:

- Jedem/r Kunden/in wird eine Nummer (KNr) zugeordnet über die diese/r eindeutig identifiziert werden kann. Der Name jeder Person wird ebenso gespeichert. Die restlichen Daten werden historisch in der Tabelle Kundendaten gespeichert, dh. es ist zu jedem Zeitpunkt möglich, die aktuelle Anschrift und den aktuellen Sonderrabatt zu ermitteln.
- Ein Auftrag wird genau von einer Kundin oder einem Kunden erstellt. Jeder Auftrag wird eindeutig durch eine Auftragsnummer (ANr) identifiziert, weiters wird das Auftragsdatum gespeichert. Aufträge enthalten mindestens eine Auftragsposition. Diese wird durch den Auftrag und einer Nr identifiziert. Zusätzlich werden ein Liefertermin und ein Rabatt gespeichert. Eine Auftragsposition ist entweder eine Leistungsposition, dh. es wurde eine bestimmte Leistung angefordert, oder eine sonstige Position. Für die sonstige Position wird ein Wert und eine Beschreibung gespeichert. Für die Leistungsposition die Menge und die zu erbringende Leistung.
- Eine Leistung wird durch eine eindeutige Nummer (LNr) identifiziert und hat eine Bezeichnung und einen Preis. Leistungen können eine bestimmte Menge anderer Leistungen benötigen. Jede Leistung ist einer Kategorie zugeordnet.

- Jede Kategorie besitzt neben einer eindeutigen Nummer (KNr), einen Namen, und optional eine „Topleistung“. Diese Leistung soll diese Kategorie repräsentieren, z.B. auf der Kategorieseite in einem zukünftigen Online-Shop.
- Nachdem ein Auftrag zur Gänze oder teilweise erfüllt worden ist, wird für die erbrachte Leistung eine Rechnung mit einer eindeutigen Nummer (RNr) und einem Rechnungsdatum angelegt. Jede Rechnung hat wiederum mindestens eine oder mehrere Rechnungspositionen. Diese werden durch eine Nummer (RPNr) identifiziert, haben einen Wert, einen Text, und gehören zu genau einer Auftragsposition.

Sie finden nun im folgenden Bild ein ER-Diagramm, das die grobe Struktur der Datenbank abbildet, wobei Sie davon ausgehen können, dass die Abbildung sowohl den Sachverhalt als auch alle Kardinalitäten bereits korrekt abdeckt.



Die notwendigen SQL-Dateien zum Anlegen dieser Datenbank sowie für das Einfügen von Testdaten werden ihnen am Beginn der Übung zur Verfügung gestellt.

Aufgabe 6. SQL

- Verändern Sie die Datei `insert.sql` dahingehend, als dass sie statt der fix vorgegebenen Nummern in den entsprechenden Tabellen die bereits angelegten Sequencer verwenden.
- Erstellen Sie eine Query, die alle Auftragsnummern ausgibt, welche noch unerledigte Auftragspositionen haben (Unerledigte Auftragspositionen haben keine zugehörige Rechnungsposition).
- Erstellen Sie eine View, die den Preis jeder Leistung inklusive der benötigten Leistungen ausgibt. Beachten Sie das Attribut Menge in der „benötigt“ Relation.
- Erstellen Sie eine View, die alle Kunden mit ihren aktuellen (neuestes Erstelldatum) Kundendaten ausgibt.

Aufgabe 7. PL/SQL

- Schreiben Sie eine Funktion `create_rechung()`. Die Funktion soll folgende Tätigkeiten durchführen:
  - Mit Hilfe einer Sequenz eine neue Rechnungsnummer generieren.
  - Eine neue Rechnung mit aktuellem Datum in der Tabelle rechnung anlegen.
  - Die generierte Rechnungsnummer mittels RETURN zurückgeben (ausgeben).
- Schreiben Sie eine Funktion `f_calc_apos`, welche als Parameter die Auftragsnummer und die APNr einer Auftragsposition erhält. Die Funktion `f_calc_apos` soll nun für die übergebene Auftragsposition den Wert berechnen. Wobei folgendes zu beachten gilt:
  - Für eine Sonstige Position ist der zurückgegebene Wert der Wert der Sonstigen Position minus dem Positionsrabatt minus dem aktuellen Sonderrabatt des Kunden.
  - Für eine Leistungsposition muss der Wert der Leistung inklusiver aller benötigten Leistungen ermittelt und mit der bestellten Menge multipliziert werden (sie können die angelegte View verwenden). Von diesem Betrag werden der Positionsrabatt minus der Sonderrabatt des Kunden abgezogen.
- Schreiben Sie eine Prozedur `create_rpos(rnr,anr,apnr)`. Die Prozedur soll für die Rechnung rnr eine Rechnungsposition anlegen die zur Auftragsposition mit dem Key (Anr,APnr) gehört. Dazu führen Sie folgende Tätigkeiten durch:
  - Überprüfen Sie, ob es zur Auftragsposition (ANr,APNr) entweder eine Leistungsposition oder eine Sonstige Position gibt.
  - Falls es noch keine Rechnungsposition für diese Rechnung gibt, setzen Sie RPNr mit 1 fest. Ansonsten lesen Sie die höchste RPNr für diese Rechnung aus und erhöhen Sie sie um 1.
  - Der Text der Rechnungsposition entspricht der Beschreibung der Sonstigen Position oder der Bezeichnung der in der Leistungsposition referenzierten Leistung. Überlegen Sie sich eine einzelne Query (benötigt UNION), die Ihnen diese Information in einer Variable speichert.
  - Den Wert der Rechnungsposition können Sie mit Hilfe der in der ersten Übung erstellen Funktion `f_calc_apos()` berechnen lassen.
  - Fügen Sie nun die erzeugten Daten in die Tabelle Rechnungsposition ein.
- Schreiben Sie einen Trigger, der überprüft, dass Rechnungspositionen einer Rechnung immer zum selben Auftrag gehören.
- Schreiben Sie einen Trigger, der überprüft, dass eine Auftragsposition entweder eine Sonstige Position oder eine Leistungsposition sein kann (nicht beides).

5.9 SQL/PSM (SQL/Persistent Stored Modules) in Mysql bzw MariaDB

5.9.1 Unterlagen

siehe Moodle

5.9.2 myspotify

ER-Modell und Relationen siehe Moodle

Aufgabe 8. Trigger 1

Erstellen Sie einen Trigger, der beim Anlegen einer hinzugefuegt Beziehung sicherstellt, dass das datum in hinzugefuegt nicht vor dem datum liegt, an dem die Playlist erstellt wurde. Falls das datum des neuen Tupels von hinzugefuegt diese Bedingung verletzt, soll stattdessen das Datum an dem die Playlist erstellt wurde gesetzt werden. (Beispiel, es gibt die Playlist P seit dem 23.10.2018. Es wird ein Eintrag erzeugt der sagt, dass ein Lied am 1.4.2017 zu P hinzugefuegt wurde. Das Datum soll beim Einfügen mit dem Erstellungsdatum von P, dem 23.10.2018, ersetzt werden.)

Aufgabe 9. Trigger 2

Erstellen Sie einen Trigger, der folgendes Verhalten bei einer Änderung in der Relation hoert implementiert:

- Wenn anzahl für ein Tupel erhöht wurde, dann soll zuletzt auf NOW() gesetzt werden. Weiters soll in diesem Fall die Zeit, die gesetzt wurde, als Teil einer Nachricht ausgegeben werden.
- Wenn anzahl per UPDATE auf den selben Wert gesetzt wurde der bereits gespeichert war, dann soll eine Warnung ausgegeben werden.

Aufgabe 10. Trigger 3

Erstellen Sie einen Trigger, der beim hinzufügen eines Lieds L in eine Playlist P folgendes Verhalten implementiert:

- Wenn P Kind anderer Playlists ist, dann soll L auch in alle Eltern-Playlists hinzugefügt werden.
- Wenn L bereits in eine Eltern Playlist hinzugefügt wurde (unabhängig von welchem Account), dann soll das Lied nicht nochmal hinzugefügt werden.
- Die Felder datum und email für die neue Tupel werden vom ursprünglichen Eintrag übernommen.

Bedenken Sie, dass Sie sich nicht um die rekursiven Abhängigkeiten zwischen den Playlists kümmern müssen, sondern dass Trigger sehr wohl wieder Trigger ausführen, usw.