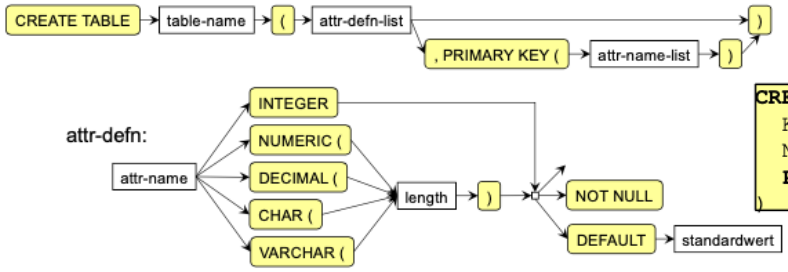
VL06/07_SQL

SQL Teil 1: Schemadefinition (DDL), Datenmanipulation (DML)

Tabellen anlegen

SQL Create Table – Syntax (Ausschnitt)




```
CREATE TABLE Kunde(
  Kdnr INTEGER,
  Name VARCHAR(100), Ort VARCHAR(100),
  PRIMARY KEY (Kdnr)
)
```

Bedeutungen

- `<table-name>` ist der Name der Tabelle
- `<attr-name>` stehen für die Namen der Attribute, d.h. die Spalten der Tabelle
- Hinter den Attributnamen kommt die Bezeichnung der Domäne
- Optionale Parameter haben folgende Bedeutungen:
  - `NOT NULL`: Wert muss sich von allen gültigen Werten unterscheiden!
  - `DEFAULT <standardwert>`: wenn beim Einfügen hier kein Wert angegeben wurde, dann wird dieser Standardwert eingetragen
- Hinter dem letzten Attribut kann dann der Primärschlüssel der Tabelle definiert werden (`PRIMARY KEY(<Attribut>)`)
- Nach Ausführung der `create table` Anweisung existiert eine leere Tabelle mit den gegebenen Attributen.

NULL-Werte

Der Wert eines Attributes kann fehlen

- weil er unbekannt ist
- weil hier kein sinnvoller Wert eingetragen werden kann
- weil er optional ist

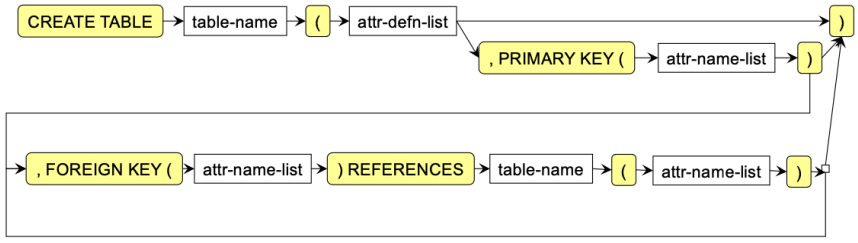
Beispiele:

- Geburtsdatum ist nicht bei allen Kunden bekannt
- Rückgabedatum nur für zurückgegebene Bestellungen sinnvoll
- Eine Bestellung kann optional eine Rechnungsadresse haben

Welchen Wert trägt man in solchen Fällen ein?

- Wert muss sich von allen gültigen Werten unterscheiden
- Dafür gibt es in Datenbanken den besonderen Wert `NULL`
- `NULL` ist nicht 0
- Die Klausel `NOT NULL` verbietet NULL-Werte von Attributen

Tabellen mit Fremdschlüsseln




```
CREATE TABLE Telefon(
  Kdnr INTEGER,
  Telefon VARCHAR(100),
  PRIMARY KEY (Kdnr, Telefon),
  FOREIGN KEY (Kdnr) REFERENCES Kunde(Kdnr)
)
```

`CREATE TABLE` mit `CONSTRAINTS`

Constraints sollten immer mit einem Namen versehen werden.

```
CREATE TABLE Kunde(
  Kdnr INTEGER, Name VARCHAR(100), Ort VARCHAR(100),
  CONSTRAINT pk_kunde PRIMARY KEY (Kdnr)
)
```

Weitere Schlüsselattribute können mit UNIQUE als Attribute mit eindeutigen Werten deklariert werden.

```
CREATE TABLE Kunde(
  Kdnr INTEGER, Name VARCHAR(100), Geboren DATE,
  CONSTRAINT pk_kunde PRIMARY KEY (Kdnr),
  CONSTRAINT u_name UNIQUE (Name, Geboren)
)
```

Datentypen

- Zahlen
  - `numeric(p,s)` oder `decimal(p,s)`: Festkommazahl
    - p: Anzahl Stellen insgesamt, s: Anzahl Nachkommastellen
  - `integer` oder `int`: Ganze Zahlen
  - `float`, `real`, `double precision`: Gleitkommazahl - Unterschiedliche Genauigkeiten

- Zeichenketten

	`char(n)`	`varchar(n)`
Reservierter Speicherplatz	immer n	wie benötigt
Speicherung	Auffüllen mit Leerzeichen	Kein Auffüllen
zu verwenden für	Zeichenkette mit fester Länge	Zeichenkette mit variabler Länge

Problem: Vergleich von Daten unterschiedlicher Typen

- Datumstyp
  - `date`: Tag, Monat, Jahr
  - `time`: Stunde, Minute, Sekunde
  - `datetime`: Beides
  - `timestamp`: Höhere Auflösung (< 1 Sekunde)

Datums/Zeit - Datentypen bestehen aus den Feldern:
- YEAR, MONTH, DAY, HOUR, MINUTE, SECOND
- ggf. mit Bruchteilen von Sekunden

```
DATE'2012-03-14'
TIMESTAMP'2012-03-14 11:23:30.5'
TO_DATE('14.03.2012 12:16:13', 'DD.MM.YYYY
```

`TO_DATE` ist Oracle-spezifisch, der Rest ist ANSI-Standard

- Sonstiges
  - `boolean`
  - `BLOB` (Binary Large Object für Binärdaten wie z.B. Bilder, Audiodaten, usw.)

Einfügen von Daten

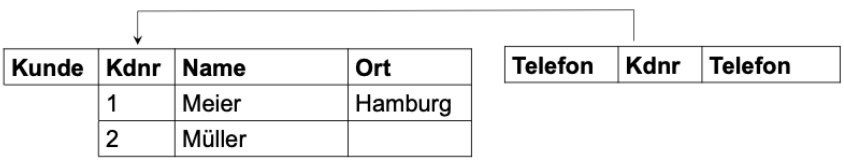
Einfügen von Tupeln in eine Tabelle

- `constantlist` ist eine Liste mit konstanten Werten (Zahlen oder Strings)
- `query` ist eine Anfrage und meint das Ergebnis der Anfrage
- Datenbank überprüft die Korrektheit jedes einzufügenden Datensatzes, d.h. wenn ein Duplikat des Primärschlüssels auftauchte, würde das Einfügen abgebrochen.

```
INSERT INTO kunde VALUES (1, 'Meier', 'Hamburg');
INSERT INTO kunde(Kdnr, Name) VALUES(2, 'Müller');
INSERT INTO kunde(Kdnr, Name) VALUES(2, 'Schulze');
```

Einfügen von Tupeln in Tabellen mit Fremdschlüssel

- Kein Unterschied zum Einfügen in andere Tabellen
- Datenbank prüft jedoch nach, ob der eingegebene Wert im Fremdschlüsselattribut auch tatsächlich als Primärschlüssel vorkommt.



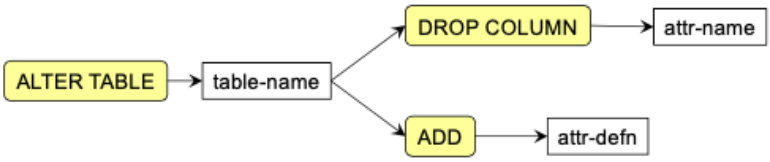
Kunde	Kdnr	Name	Ort
	1	Meier	Hamburg
	2	Müller	

Telefon	Kdnr	Telefon

```
INSERT INTO telefon VALUES (2, '122334455');
INSERT INTO telefon VALUES (3, '122334455');
```

Verändern von Daten und Tabellen

Aktualisieren von Tabellen



Kunde	Kdnr	Name	Ort
	1	Meier	Hamburg
	2	Müller	

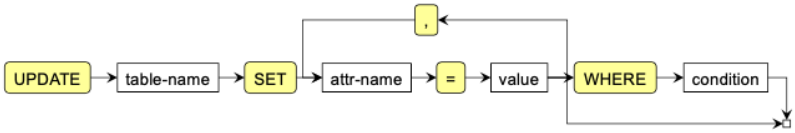
➡

Kunde	Kdnr	Name	GebDat
	1	Meier	
	2	Müller	

Änderungen am Tabellenschema werden durch das Kommando **`ALTER TABLE`** durchgeführt.

```
ALTER TABLE kunde DROP COLUMN Ort;
ALTER TABLE kunde ADD GebDat DATE;
ALTER TABLE kunde ADD CONSTRAINT u_geb
                      UNIQUE(Name, GebDat);
```

Aktualisieren von Tupeln



Kunde	Kdnr	Name	GebDat
	1	Meier	
	2	Müller	

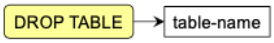
➡

Kunde	Kdnr	Name	GebDat
	1	Meier	1990-07-23
	2	Müller	

Aktualisierungen von Tupeln werden mit dem Kommando **`UPDATE`** durchgeführt.

```
UPDATE kunde SET GebDat = DATE'1990-07-23'
             WHERE Kdnr = 1;
```

Löschen von Daten und Tabellen



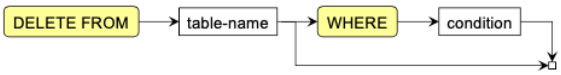
Kunde	Kdnr	Name	GebDat
	1	Meier	1990-07-23
	2	Müller	

➡

Kunde	Kdnr	Name	GebDat
	2	Müller	

Tabellen werden durch das Kommando **`DROP TABLE`** gelöscht.

```
DROP TABLE handy;
```



Tupel werden durch das Kommando **`DELETE FROM`** gelöscht.

```
DELETE FROM kunde WHERE kdnr = 1;
```

`DELETE FROM kunde;`
- Geht nicht da es durch eine andere Tabelle abhängig ist.
- Betrifft nur die Tabellen Zeilen und nicht die Tabelle selbst!!

`DROP TABLE kunde;`
- Fremdschlüssel zeigt auf einen Primarschlüssel der Tabelle daher geht es nicht
- Kann Tabelle nur löschen wenn die vorherige Bedienung nicht erfüllt ist!!

Sequenzen

Sequenz: (wie ein Zähler)

- Datenbankobjekt, das eine Folge von Integer – Werten generiert

```
CREATE SEQUENCE <name>;
```

generiert eine Sequenz 1,2,3, ...
Mit den Pseudospalten `NEXTVAL` und `CURRVAL` werden die Sequenzwerte referenziert:

- `NEXTVAL` gibt den nächsten verfügbaren Sequenzwert zurück.
- `CURRVAL` liefert den aktuellen Sequenzwert (nur gültig nach `NEXTVAL`)

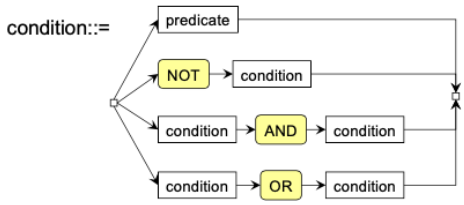
```
CREATE SEQUENCE seq_kdnr;
SELECT seq_kdnr.nextval FROM DUAL;
SELECT seq_kdnr.currval FROM DUAL;
```

SQL Teil 2: SQL Anfragen aus einer Tabelle

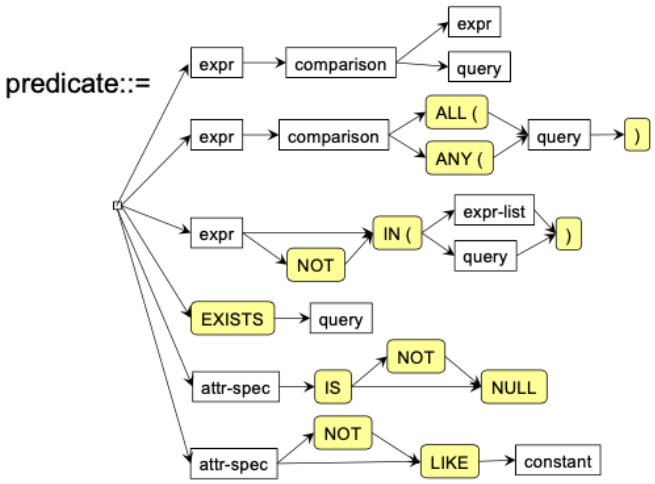
Grundstruktur von Anfragen

Grundstruktur der Anfragen Syntax einer Anfrage (query):

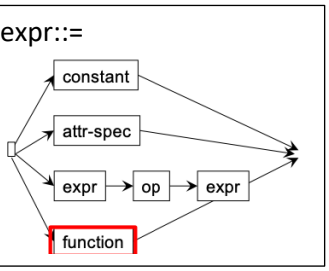
Syntax einer Bedingung (condition):



Prädikate



comparison – Operatoren
= <> < > <= >=



op:+-*/ ||

WHERE-Klausel: Bedingungen definieren

- legt anhand von Suchbedingungen fest, welche Zeilen von den in der `FROM` – Klausel bezeichneten Tabellen ausgewählt werden
- Ergebnismenge enthält nur die Datensätze, für die die Bedingung der `WHERE` – Klausel zu TRUE ausgewertet wird
- Wenn die `WHERE`-Klausel fehlt, wird immer TRUE angenommen!

Bedingungen definieren

Vergleichsoperatoren	> < <> = <= >=
Bereichsprüfung	BETWEEN
Elementprüfung	IN
Mustervergleich	LIKE
Nullwertprüfung	IS NULL, IS NOT NULL
Logische Operatoren	NOT, AND, OR

Bedingungen definieren – Beispiele Vergleichsoperatoren

```
SELECT last_name, salary
FROM hr.employees
WHERE salary*1.1 > 5000;

SELECT last_name, salary
FROM hr.employees
WHERE salary BETWEEN 4000 AND 6000;

SELECT last_name, salary
FROM hr.employees
WHERE manager_id IS NULL;
```

Bedingungen definieren – Beispiele IN

```
SELECT department_name
FROM hr.departments
WHERE manager_id IN (200,201,121);

SELECT department_name
FROM hr.departments
WHERE manager_id = 200
   OR manager_id = 201
   OR manager_id = 121;
```

Bedingungen definieren – Beispiele LIKE

```
SELECT last_name
FROM hr.employees
WHERE upper(last_name) LIKE 'B%';

SELECT last_name
FROM hr.employees
WHERE upper(last_name) LIKE '%SON';
```

NULL – Werte

Dreiwertige Logik: TRUE, FALSE, NULL
Jeder Ausdruck mit NULL ergibt wieder NULL

- außer IS NULL / IS NOT NULL

```
SELECT last_name, commission_pct
FROM hr.employees
WHERE last_name LIKE 'A%'
```

```
LAST_NAME        COMMISSION_PCT
---------------- --------------
Abel                         ,3
Ande                         ,1
Atkinson
Austin
```

```
SELECT last_name, commission_pct
FROM hr.employees
WHERE last_name LIKE 'A%'
  AND (commission_pct = .3
       OR commission_pct <> .3)
```

```
LAST_NAME        COMMISSION_PCT
---------------- --------------
Abel                         ,3
Ande                         ,1
```

Bedingungen definieren – Beispiele IS [NOT] NULL

```
SELECT last_name, commission_pct
FROM hr.employees
WHERE commission_pct IS NOT NULL;

SELECT last_name, commission_pct
FROM hr.employees
WHERE commission_pct IS NULL;

SELECT last_name, commission_pct,
       CASE
           WHEN commission_pct IS NULL THEN 0
           ELSE commission_pct
       END
FROM hr.employees;
```

Bedingungen definieren – Beispiele log. Operatoren

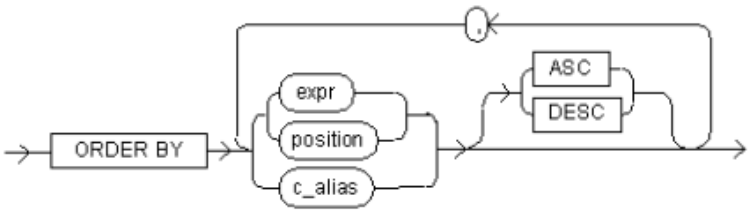
Mehrere Bedingungen können in der WHERE-Klausel verknüpft werden

```
SELECT *
FROM hr.departments
WHERE location_id <> 1700
  AND location_id <> 1800;

SELECT last_name, job_id, salary
FROM hr.employees
WHERE job_id = 'ST_MAN' OR
      job_id = 'ST_CLERK' AND salary < 2500;
```

- Priorität wie gewohnt: Vergleichsoperatoren, NOT, AND, OR
- durch Klammerung steuerbar

Sortierung mit ORDER BY




```
SELECT last_name, employee_id
FROM hr.employees
ORDER BY last_name ASC, employee_id DESC;
```

```
SELECT last_name, employee_id
FROM hr.employees
ORDER BY last_name, employee_id DESC;
```

Oracle – Systemvariablen

Dummy-Tabelle **DUAL**, Abfrage von Werten zu ermöglichen (Tabellen unabhänig)

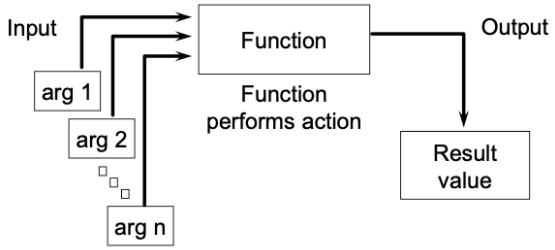
```
SELECT current_date, current_timestamp, user
FROM dual;
```

```
CURRENT_DATE CURRENT_TIMESTAMP                 USER
------------ ------------------------------------
20.04.17     20.04.17 11:41:05,.. EUROPE       FHEINE
                                  /BERLIN
```

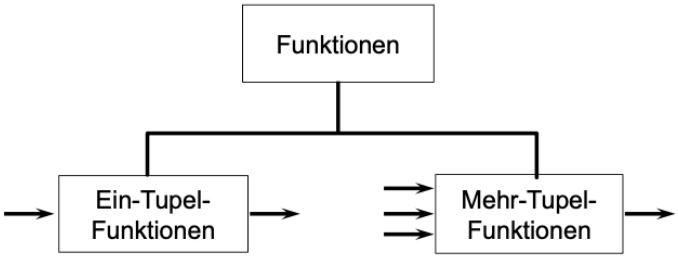
Oracle – Variable	Bedeutung
CURRENT_DATE	Aktuelles Datum
CURRENT_TIMESTAMP	Aktueller Timestamp
USER	Eingeloggte BenutzerIn

Funktionen und Systemvariablen

Funktionen in SQL



Arten von SQL – Funktionen

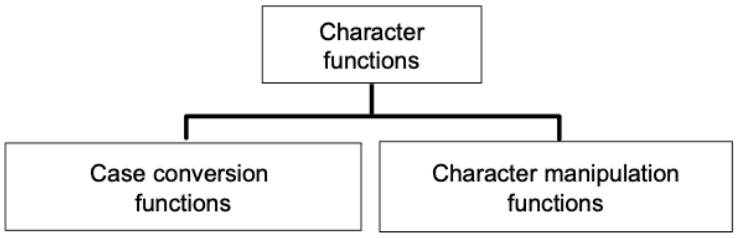


Ein-Tupel-Funktionen

- manipulieren Daten
- erwarten ein oder mehrere Argumente
  - Konstante
  - Variable
  - Spaltenname - Ausdruck
- und geben einen Wert zurück
- geben einen Wert pro Zeile zurück
- können den Datentyp verändern
- können geschachtelt sein

Funktionen und Systemvariablen

Funktionen für Zeichenketten



LOWER
UPPER
INITCAP

CONCAT
SUBSTR
LENGTH
INSTR
LPAD
TRIM

```
SELECT last_name, first_name,
  CONCAT(first_name, last_name),
  LENGTH(last_name),
  INSTR(last_name, 'a'),
  SUBSTR(last_name, 2, 3)
FROM hr.employees;
```

Funktion	Ergebnis
CONCAT('Lang', 'strumpf')	Langstrumpf
'Lang' || 'strumpf'	Langstrumpf
SUBSTR('String', 1, 3)	Str
LENGTH('String')	6
INSTR('String', 'r')	3
TRIM(' Langstrumpf ')	Langstrumpf

```
SELECT 'Die Berufsbezeichnung für ' ||
INITCAP(email)
      || ' ist ' || INITCAP (SUBSTR(job_id, 4))
      AS "AngDetails"
FROM hr.employees
WHERE UPPER(last_name) = 'KING';
```

```
AngDetails
 -------------------------------------
 Die Berufsbezeichnung für Sking ist Pres
 Die Berufsbezeichnung für Jking ist Rep
```

Funktion	Ergebnis
LOWER('SQL Kurs')	sql kurs
UPPER('SQL Kurs')	SQL KURS
INITCAP('SQL Kurs')	Sql Kurs

Funktionen für Numerische Daten

ROUND: rundet Werte auf gegebene Stellen
- ROUND(45.926, 2) → 45.93

TRUNC: schneidet auf gegebene Stellen ab
- TRUNC(45.926, 2) → 45.92

MOD: ergibt Rest bei Division
- MOD(16, 3)→1

```
SELECT ROUND(45.923,2), ROUND(45.923,0),
       ROUND(45.923,-1)
FROM DUAL;
```

```
ROUND(45.923,2) ROUND(45.923,0) ROUND(45.923,-1)
--------------- --------------- ----------------
45.92            46              50
```

```
SELECT TRUNC(45.923,2), TRUNC(45.923), TRUNC(45.923,-1)
FROM DUAL;
```

```
TRUNC(45.923,2) TRUNC(45.923) TRUNC(45.923,-1)
--------------- ------------- ---------------
45.92           45            40
```

Funktionen für Datumswerte

- Addiert oder subtrahiert man eine Zahl zu oder von einem Datum, so ergibt sich ein Datumswert:
  `date +- integer` ergibt `date`
- Subtraktion zweier Daten führt auf die Anzahl der Tage zwischen den Daten:
  `date - date` ergibt `integer`

```
SELECT last_name,
    ROUND((CURRENT_DATE-hire_date)/7, 2) AS weeks
FROM hr.employees
WHERE department_id = 50;
```

```
LAST_NAME                 WEEKS
------------------------- ----------
OConnell                  930,65
Weiss ...                 1083,22 ...
```

Konvertierungsfunktionen

Explizite Datentyp – Konversion

Typumwandlung mit CAST:
`CAST(<WERT> AS <Datentyp>)`

```
SELECT CAST ('16.04.16' AS DATE) FROM DUAL;
SELECT CAST ('1234,56' AS NUMERIC(38,10)) FROM DUAL;
SELECT CAST ('1234' AS INTEGER) FROM DUAL;
SELECT CAST (123.56 AS VARCHAR(30)) FROM DUAL;
```

Achtung: Format für Datum und Zahlen abhängig von Systemeinstellungen.
In Oracle: `TO_CHAR, TO_DATE` und `TO_NUMBER` ermöglichen eigene Formate

CASE WHEN

Ermöglicht die Formulierung von `IF-THEN-ELSE` - Statements

CASE WHEN prüft nacheinander die Bedingungen
- bei Übereinstimmung mit Bedingung i wird Ausdruck i zurückgeliefert
- bei keiner Übereinstimmung wird **Ausdruck n** zurückgeliefert

```
CASE   WHEN <Bedingung 1> THEN <Ausdruck 1>
       WHEN <Bedingung 2> THEN <Ausdruck 2>
       ...
       ELSE <Ausdruck n>
END
```

```
SELECT last_name, job_id, salary,
       CASE   WHEN job_id = 'FI_ACCOUNT' THEN salary*1.1
              WHEN job_id = 'ST_CLERK' THEN salary*1.15
              WHEN job_id = 'ST_MAN' THEN salary*1.20
             ELSE salary
       END AS new_salary
FROM hr.employees;
```

Aggregatsfunktionen und Gruppierung: GROUP BY, HAVING

SUM	Summe der Werte einer Spalte
AVG	Durchschnitt der Werte einer Spalte
COUNT	Zahl von Werten in einer Spalte
MAX	größter Wert in einer Spalte
MIN	kleinster Wert in einer Spalte

LAST_NAME	DEPARTMENT_ID	SALARY
Whalen	10	4400
Hartstein	20	13000
Fay	20	6000
Raphaely	30	11000
Khoo	30	3100
Baida	30	2900
Colmenares	40	2500
Mavris	40	6500
Weiss	50	8000
Fripp	50	8200

```
SELECT SUM(salary) FROM hr.employees;
SELECT AVG(salary) FROM hr.employees;
SELECT COUNT(*) FROM hr.employees;
SELECT COUNT(manager_id) FROM hr.employees;
SELECT COUNT(DISTINCT manager_id)
FROM hr.employees;
```

Gib den Namen und das Gehalt des bestverdienenden Angestellten aus:

```
SELECT last_name, MAX(salary) FROM hr.employees;
-----------------------------------------------
FEHLER in Zeile 1:
ORA-00937: keine Gruppenfunktion für Einzelgruppe
-----------------------------------------------
SELECT last_name, salary FROM hr.employees
WHERE salary =
         (SELECT MAX(salary) FROM hr.employees);
```

Gruppierung mit der GROUP - BY - Klausel

- Gruppierung ist die Zerlegung einer Relation bezüglich eines oder mehrerer Attribute in disjunkte Teilmengen.
- GROUP BY gruppiert Zeilen auf der Basis gleicher Attributwerte
- Beispiel: Gib für jede Abteilung deren Nummer und das Durchschnittsgehalt der Angestellten dieser Abteilung an.

```
SELECT department_id, ROUND(AVG(salary))
FROM hr.employees
GROUP BY department_id;
DEPARTMENT_ID     ROUND(AVG(SALARY))
------------- ----------------------
100                             8600
30                              4150
20                              9500
```

HAVING - Klausel

- HAVING selektiert nur diejenigen Gruppen im GROUP BY - Teil, die eine Bedingung erfüllen
- Beispiel: Gib für alle Abteilungen mit mehr als 10 Angestellten die Abteilungsnummer und das Durchschnittsgehalt der Angestellten dieser Abteilung an.

```
SELECT department_id, ROUND(AVG(salary))
FROM hr.employees
GROUP BY department_id
HAVING COUNT (*) > 10;
```

- Die HAVING-Klausel erlaubt es, die Gruppen als Ganzes zu untersuchen, indem Bedingungen mit aggregierten Werten gestellt werden können.

SQL Teil 3: SQL Anfragen aus mehreren Tabellen

Verbundoperationen (Join)

JOIN - Operatoren in der **FROM** - Klausel:

```
SELECT <select – liste>
FROM <tabelle1> <jointyp> JOIN <tabelle2>
ON <Suchbedingung> | USING <Spaltenname>

<jointyp>::= INNER | <outer-join-typ> [OUTER]

<outer-join-typ>::= LEFT | RIGHT | FULL
```

Natürlicher Verbund:

```
SELECT <select – liste>
FROM <tabelle1> NATURAL JOIN <tabelle2>
```

Kreuzprodukt:

```
SELECT <select – liste>
FROM <tabelle1> CROSS JOIN <tabelle2>
```

JOIN - Beispiele

Das Kreuzprodukt mit Komma! Macht kein Unterschied!

```
SELECT * FROM A,B;
```

```
SELECT *
FROM A CROSS JOIN B;
```

```
SELECT d.department_id, e.last_name
FROM hr.departments d CROSS JOIN hr.employees e;
```

```
SELECT d.department_id, e.last_name
FROM hr.departments d, hr.employees e;
```

```
SELECT *
FROM A, B
WHERE <condition>
```

```
SELECT *
FROM A INNER JOIN B
     ON <condition>
```

```
SELECT d.department_name, e.last_name
FROM hr.departments d JOIN hr.employees e
     ON d.department_id = e.department_id;
```

Outer Join

```
SELECT d.department_name, e.last_name
FROM hr.departments d LEFT JOIN hr.employees e
     ON d.department_id = e.department_id;
```

Es werden auch die Namen derjenigen Abteilungen ausgewählt, die keine Mitarbeiter haben. In die Datensätze mit fehlenden Übereinstimmungen werden NULL-Werte eingetragen.

Finde die Namen aller Abteilungen, die keine Mitarbeiter haben:

```
SELECT d.department_name, e.last_name
FROM hr.departments d LEFT JOIN hr.employees e
     ON d.department_id = e.department_id
WHERE e.department_id IS NULL;
```

Mengenoperationen

- **EXCEPT**: Mengendifferenz
- **INTERSECT**: Durchschnitt
- **UNION**: Vereinigungsmenge ohne Duplikate
- **UNION ALL**: Duplikate in Vereinigungsmenge zulassen

ORACLE:
Statt EXCEPT MINUS verwenden.

Beispiel: Liste aller Namen von Angestellten und Kunden:

```
SELECT manager_id FROM hr.employees
UNION
SELECT manager_id FROM hr.departments;
```

Unterabfragen

- Unterabfrage ist eine **SELECT** – Anweisung in einer **SELECT** – Anweisung
- kann in **WHERE**, **HAVING** oder **FROM** – Klauseln stehen
- wird vor allem mit Suchbedingungen mit **ANY, ALL, IN** und **EXIST** verwendet

Unterabfrage in der WHERE-Klausel

- Die in ihrer WHERE-Klausel eine weitere Abfrage enthaltende **SELECT**- Anweisung wird als äußere Abfrage bezeichnet, die in der WHERE-Klausel stehende entsprechend als Unterabfrage, innere Abfrage, SUBQUERY oder SUBSELECT.
- Die in einer äußeren Abfrage betrachteten Relationen und Attribute sind in **jeder inneren sichtbar**.

Welche Angestellten verdienen überdurchschnittlich?

```
SELECT last_name
FROM hr.employees
WHERE salary > (SELECT AVG(salary) FROM hr.employees);
```

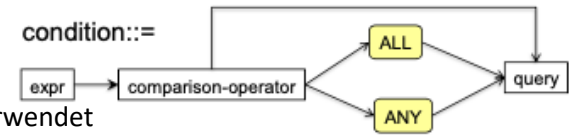
Welche Angestellten haben vor Herrn King in der Firma zu arbeiten begonnen?

```
SELECT last_name
FROM hr.employees
WHERE hire_date <
       (SELECT hire_date FROM hr.employees
         WHERE last_name = 'King'
           AND first_name = 'Steven');
```

ANY | ALL – Bedingung

Unterabfrage

- ist eine **SELECT**–Anweisung in einer **SELECT**–Anweisung
- kann in **WHERE**, **HAVING** oder **FROM**–Klauseln stehen
- wird vor allem mit Suchbedingungen mit **ANY**, **ALL**, **IN** und **EXISTS** verwendet



Fehlen **ALL** und **ANY**, so darf die Unterabfrage nur einen einzelnen Wert liefern!

ALL: die Bedingung muss für alle Ergebnisse der Unterabfrage wahr sein
ANY: die Bedingung muss für mindestens ein Ergebnis der Unterabfrage wahr sein
Ohne **ALL** und **ANY** darf die Unterabfrage nur einen einzigen Wert liefern

ANY | ALL – Bedingung - Beispiele

Welche Angestellten haben vor Herrn Taylor in der Firma zu arbeiten begonnen?

```
SELECT last_name
FROM hr.employees
WHERE hire_date <
       (SELECT hire_date FROM hr.employees
        WHERE last_name = 'Taylor');
```

→ Fehler!! (weil es zwei Personen mit diesem Namen gibt....)

Wer verdient mehr als alle Angestellten von Frau Greenberg?

```
SELECT *
FROM hr.employees
WHERE salary >= ALL (SELECT e1.salary
                     FROM hr.employees e1 JOIN hr.employees e2
                      ON e1.manager_id = e2.employee_id
                       WHERE UPPER(e2.last_name) = 'GREENBERG');
```

Wer verdient mehr als irgendein Angestellter von Frau Greenberg?

```
SELECT *
FROM hr.employees
WHERE salary >= ANY (SELECT e1.salary
                     FROM hr.employees e1 JOIN hr.employees e2
                      ON e1.manager_id = e2.employee_id
                       WHERE UPPER(e2.last_name) = 'GREENBERG');
```

IN – Bedingung

<condition> ::= <expr> [NOT] IN <query>

Wenn `expr` NOT NULL:

- Ergebnis (**IN**) ist TRUE, wenn
  - `expr` in der Ergebnismenge der Unterabfrage `query` gefunden wird
- Ergebnis (**IN**) ist FALSE, wenn
  - `expr` in der Ergebnismenge der Unterabfrage `query` nicht gefunden wird
  - Ergebnismenge der Unterabfrage `query` leer ist (**IN**)

Wenn `expr` NULL:

- Ergebnis (**IN** oder **NOT IN**) UNKNOWN (Datensatz kommt nicht in die Ergebnismenge der Abfrage)

Welche Angestellten arbeiten in Verkaufsabteilungen?

```
SELECT *
FROM hr.employees
WHERE department_id IN (
       SELECT department_id
       FROM hr.departments
       WHERE department_name LIKE '%Sales%');
```

Welche Angestellten heißen nicht so wie ein Manager?

```
SELECT *
FROM hr.employees
WHERE last_name NOT IN (SELECT last_name
                        FROM hr.employees
                        WHERE job_id LIKE '%MAN' );
```

EXISTS- Bedingung

- ist wahr, wenn die Unterabfrage mindestens einen Datensatz selektiert
- kann in **WHERE**, **HAVING** oder **FROM** – Klauseln stehen
- `EXISTS` entspricht dem Existenzquantor
  Ergebnis ist TRUE, wenn die Unterabfrage `query` mindestens ein Tupel liefert

Bsp: Alle Angestellten, die eine Abteilung leiten:

```
SELECT *
FROM hr.employees e
WHERE EXISTS (
       SELECT *
       FROM hr.departments d
       WHERE d.manager_id = e.employee_id
);
```

In der äußeren Abfrage werden alle Angestellten betrachtet. In der inneren Abfrage werden alle Abteilungen selektiert, die von dem aktuellen Angestellten geleitet werden (evtl. 0). Ein Angestellter wird nur dann zurückgeliefert, wenn die innere Abfrage mind. einen Datensatz geliefert hat.

Wie lauten die Namen der Länder, in denen keine Niederlassung existiert?

```
SELECT DISTINCT country_name FROM hr.countries c
WHERE NOT EXISTS (
                  SELECT *
                  FROM hr.locations l
                  WHERE l.country_id = c.country_id);
```

Korrelierte Unterabfragen

• Eine Unterabfrage heißt korreliert, wenn es Attribute der äußeren SELECT– Anfrage gibt, die mit Attributen der inneren SELECT–Anweisung in Beziehung gesetzt sind.

```
SELECT e1.last_name, e1.salary, e1.department_id FROM hr.employees e1
WHERE salary >= ALL (SELECT salary
                     FROM hr.employees e2
                     WHERE e1.department_id = e2.department_id);
Ergebnis: Alle Angestellten, die jeweils das Maximum in ihrer Abteilung verdienen
```

Unterabfragen in der FROM–Klausel - Beispiel

• In der FROM –Klausel kann anstelle von Tabellennamen eine Unterabfrage stehen. Für die Ergebnismenge der Unterabfrage muss ein alias–Name angegeben werden.

• Wer verdient am wenigsten und wie viel ist das?

```
SELECT *
FROM (
      SELECT MIN(salary) m
      FROM hr.employees ) e
JOIN hr.employees
ON e.m = salary;
```

Unterabfragen in der INSERT oder UPDATE -Anweisung

- In einer INSERT oder UPDATE-Anweisung kann eine Unterabfrage stehen, mit der Werte aus einer anderen Tabelle übertragen werden können.
- Das Geburtsdatum aller Kunden ohne gebdat soll auf das minimale Datum aller bisherigen Kunden gesetzt werden.

```
UPDATE kunde
SET gebdat = (SELECT MIN(gebdat) FROM kunde)
WHERE gebdat IS NULL;
```

- In die Kundentabelle sollen alle Angestellten, deren Berufsbezeichung auf `MAN` endet, eingefügt werden. Die Kundennummer dieser neuen Kunden soll als `employee_id+1000` berechnet werden.

```
INSERT INTO kunde (kdnr, name, gebdat)
SELECT employee_id+1000, last_name, NULL
FROM hr.employees
WHERE job_id LIKE '%MAN';
```

Auswertungsreihenfolge von SQL-Anweisungen

5	**SELECT**:	Ausgabespalten definieren
1	**FROM**:	Tabellen lesen, Joins verarbeiten
2	**WHERE**:	Entscheiden, welche Zeilen weiterverarbeitet werden
3	**GROUP BY**:	Die verbliebenen Zeilen gruppieren
4	**HAVING**:	Entscheiden, welche Gruppen weiterverarbeitet werden
6	**ORDER BY**:	Ergebnis sortieren

Dies bedeutet: Alias-Namen aus `SELECT` sind nur im `ORDER BY` bekannt!

Reguläre Ausdrücke

Regulärer Ausdruck: Repräsentation eines Musters (pattern)

Anhand eines Musters werden in der zu durchsuchenden Tabelle passende Zeichenketten gefunden („pattern matching“).

- Zeichen, die beim Suchen direkt übereinstimmen müssen, werden auch als solche in einem regulären Ausdruck notiert.
- Zusätzlich können Metazeichen notiert werden - sie stehen für ganze Gruppen von Zeichen.

Metazeichen

Zeichen	Bedeutung	Beispiel
.	steht für ein beliebiges Zeichen	hob.it passt zu hobait, hobbit,...
[ABC]	Ein Zeichen der angegeben (A,B oder C)	[ab]bc passt zu abc und bbc
[A-Z]	Ein Zeichen aus dem angegebenen Bereich	[a-c]bc passt zu abc,bbc,cbc
[[:<ZK>:]]	Ein Zeichen aus der Zeichenklasse wie alpha (alphab. Zeichen), digit (Ziffern), alphanum, upper (A-Z),...	[[:lower:]] = [a-z]
+	kennzeichnet Mengenangabe, steht für **ein oder mehrere** Vorkommen des Zeichens oder Metazeichens	ba+rk passt auf bark, baark,baaark, ...
*****	kennzeichnet Mengenangabe, steht für **kein oder mehrere** Vorkommen des Zeichens oder Metazeichens	ba*rk passt auf brk, bark, baark, ...
?	kennzeichnet Mengenangabe, steht für **kein oder ein-maliges** Vorkommen des Zeichens oder Metazeichens	ba?rk passt auf brk, bark
{n} oder {n,m}	kennzeichnet Mengenangabe, steht n-maliges oder n bis m-maliges Vorkommen des Zeichens oder Metazeichens	hob{2}it passt auf hobbit
^	steht für die Startposition der Zeichenfolge	^A stimmt mit A als erstem Zeichen überein
$	steht für die Endposition der Zeichenfolge	b$ stimmt mit b als letztem Zeichen überein

REGEXP_LIKE

stellt fest, ob ein Muster in der Zeichenkette existiert

```
ALTER TABLE kunden
ADD CONSTRAINT ch_email_gueltig CHECK(REGEXP_LIKE(email,
'^[A-Za-z0-9._-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$'));
```

REGEXP_REPLACE

ersetzt die zum Ausdruck passende Teilzeichenkette durch eine andere.

```
SELECT
REGEXP_REPLACE('Willi Wiberg',' {2,}',' ') normalisiert
FROM dual;
```

REGEXP_INSTR

gibt die Zeichenposition zurück, an der die zum Ausdruck passende Teilzeichenkette beginnt

```
SELECT REGEXP_INSTR(
  'Maenner und Frauen passen einfach nicht zusammen',
  'F[[:alpha:]]{5}') result
FROM dual;
```

REGEXP_SUBSTR

extrahiert die zum regulären Ausdruck passende (Teil-)Zeichenkette

```
SELECT REGEXP_SUBSTR(
  'Maenner und Frauen passen einfach nicht zusammen',
  '[A-Z][[:alpha:]]{5}', 1, 2) result
FROM dual
```

SQL Teil 4: Sichten

Sichten (Views) dienen dazu, einen Ausschnitt der Datenbank in einer speziellen Darstellung zu zeigen.

- Vereinfachen dadurch den Zugriffs auf bestimmte Daten

Beispiel / Demo: Abfrage von Department-Informationen:

```
SELECT department_id, COUNT(*) cnt_emp,
ROUND(AVG(salary),0) avg_sal, MIN(salary) min_sal,
MAX(salary) max_sal
FROM hr.employees
WHERE department_id IS NOT NULL GROUP BY department_id;
```

Erzeugen und Aktualisieren von Sichten

- Eine Sicht wird als Ergebnis einer Abfrage formuliert.
- Sichten stellen abgeleitete Relationen dar, deren Inhalt bezüglich des Schemas redundant ist.
  - **virtuelle Sichten**: Sichtdefinition im Schema gespeichert, Daten nicht physisch gespeichert
  - **materialisierte Sichten (snapshot):** Daten physisch gespeichert
  - "Sicht" normalerweise äquivalent zu "virtuelle Sicht"
- Auf Sichten können „im Prinzip“ dieselben Operationen angewendet werden wie auf die„Basisrelationen“
  - bis auf (deutliche) **Einschränkungen beim Ändern**.
- **Änderungen von Basisrelationen** werden sofort in allen darauf definierten virtuellen Sichten sichtbar.
- Nachteil kann im **Unterschätzen des Aufwands** der zugehörigen Abfragen liegen (virtuelle Sichten)

Anwendungen von Sichten

Bereitstellung einer **Schnittstelle** für eine Anwendung

- Beispiel: Daten über die Abteilungen für einen Bericht bereitstellen

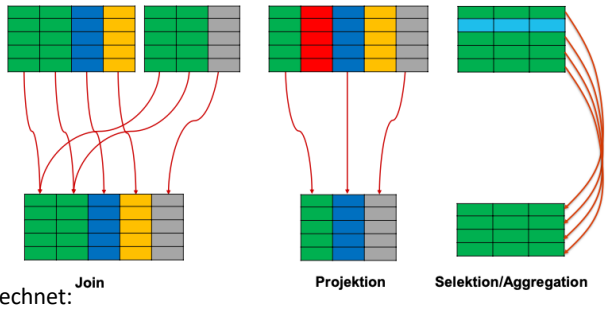
Einschränkungen des Zugriffs auf Daten

- Beispiel: Sicht auf Angestellten-Daten ohne Gehälter

Integration von Daten aus verschiedenen Quellen

Beispiel:

- 2 Anwendungen haben jeweils eine Produkte-Tabelle
- Eine Sicht ermöglicht den einheitlichen Zugriff auf alle Produkte



Sichten – Beispiele

Beispiel: Es wird eine Sicht definiert, die aus dem Geburtsdatum das Alter der Kunden berechnet:

```
CREATE VIEW kunde_alter AS SELECT name,
TRUNC(MONTHS_BETWEEN(current_date, gebdat) / 12) k_alter,
gebdat FROM kunde;
```

```
SELECT * FROM kunde_alter;
```

Änderungen über Sichten

- Änderungen von Werten in den Basistabellen über Sichten sind nur dann möglich, wenn eindeutig klar ist, welche Änderungen aus welchen Basisrelationen gemeint sind.
- Im allgemeinen sind Änderungen auf Sichten nur dann möglich, wenn die Sicht auf **einer** Basisrelation aufbaut.

Geschachtelte Sichten- Definition

Sichten dürfen auch auf Sichten aufgebaut werden.

- Beispiel: Aufbauend auf der Sicht kunde_alter könnte man eine Sicht definieren, in der die/der jüngste/jüngster Kunde_in angegeben wird.

```
CREATE OR REPLACE VIEW nesthaekchen AS SELECT *
FROM kunde_alter
WHERE k_alter =
    (SELECT MIN(k_alter) FROM kunde_alter);
```

Beispiel-Zugriffe auf das Data Dictionary

Alle Tabellen des aktuellen Benutzers:

```
SELECT * FROM user_tables;
```

Alle Tabellen des Benutzers HR:

```
SELECT * FROM all_tables WHERE owner = 'HR';
```

Alle Sichten des aktuellen Benutzers:

```
SELECT * FROM user_views;
```

Zusammenfassung Teil 1

- Tabellen werden mit
- CREATE TABLE angelegt,
- mit ALTER TABLE verändert
- und mit DROP TABLE gelöscht (DDL).
- Unbekannte oder nicht passende Werte können als NULL gespeichert werden.
- Datentypen für Spalten: Numerisch, Zeichenketten, Datumstypen.
- Daten können mit INSERT, UPDATE und DELETE angelegt, verändert und gelöscht werden (DML).
- Fortlaufende IDs können mit Sequenzen generiert werden.

Zusammenfassung Teil 2

Selects können im Where-Teil komplizierte Bedingungen enthalten
Selects, Updates und Inserts können Funktionen enthalten wie

- Ein-Tupel – Funktionen
  - Zeichenketten-Konversion: `LOWER, UPPER, INITCAP`
  - Zeichenketten-Manipulation: `CONCAT, SUBSTR, LENGTH, INSTR, LPAD`
  - numerische Funktionen: `ROUND, TRUNC, MOD`
  - Datumsfunktionen: `MONTHS_BETWEEN, ADD_MONTHS, NEXT_DAY, ....`
  - Konversionsfunktionen: `TO_NUMBER, TO_CHAR, TO_DATE`
- Mehr-Tupel-Funktionen
  - Aggregierungsfunktionen
  - Gruppierung

Zusammenfassung Teil 3

Daten aus mehreren Tabellen abfragen:

- Join: Cross, Natural, Outer
- Mengenoperationen
- Unterabfragen:
  - Mit einem Ergebnis
  - Mit mehreren Ergebnissen: Any, All
  - IN und EXISTS
  - Korrelierte Unterabfragen
  - In der FROM-Klausel
  - In Verbindung mit INSERT und UPDATE
- Reguläre Ausdrücke

Zusammenfassung Teil 4

- Sichten (Views) ermöglichen die Anpassung an die Bedürfnisse verschiedener Benutzergruppen
- Sichten (Views) dienen dazu, einen Ausschnitt der Datenbank in einer speziellen Darstellung zu zeigen.
- Eine Sicht wird als Ergebnis einer Abfrage formuliert.
- Sichten können auf Sichten aufbauen.
- Sichten können virtuell oder materialisiert sein.
- Änderungen an Sichten sind nur eingeschränkt möglich.
- Auf das Data Dictionary (Katalog) wird über Views zugegriffen.