SQL

	DDL	DML
Einfügen / erzeugen	CREATE	INSERT
Ändern	ALTER	UPDATE
Löschen	DROP	DELETE

Datentypen

- CHAR(n)/CHARACTER(n): Zeichenkette, fixe Länge
- CHAR VARYING(n)/VARCHAR(n): Zeichenkette, variable Länge
- INT/INTEGER: Ganzzahl
- REAL: Fliesskommazahl
- NUMERIC(p,s)/DECIMAL(p,s): Festkommazahl

DDL – Data Definition Language

- Erzeugen und Löschen von Daten

Schema

```
CREATE SCHEMA <db_name>;
DROP SCHEMA <db_name>;
```

Domain

- Wertebereich
- Einmal zentral definiert -> gilt für das ganze DB-Schema

```
CREATE DOMAIN ssn_type AS CHAR(9);
```

Table

```
1.  CREATE TABLE IF NOT EXISTS Anstellung(
2.    MVorname varchar(100) NOT NULL,
3.    MName    varchar(100) NOT NULL,
4.    FName    varchar(100) NOT NULL,
5.
6.    CONSTRAINT PK_Anstellung PRIMARY KEY (MVorname,MName),
7.    CONSTRAINT FK_Mitarbeiter FOREIGN KEY (MVorname,MName) REFERENCES Mitarbeiter(Vorname,Nachname),
8.    CONSTRAINT FK_Firma FOREIGN KEY (FName) REFERENCES Firma(Name)
9.  );
```

```
1.  ALTER TABLE <table_name> ADD <column_name> <domain> <constraint>;
2.  ALTER TABLE HatGeliefert ADD menge INT NOT NULL DEFAULT 0;
3.  ALTER TABLE CD ALTER titel SET NOT NULL;
4.  ALTER TABLE CD ALTER titel DROP NOT NULL;
```

Triggers

```
1.  CONSTRAINT HatStFKStil FOREIGN KEY (stil) REFERENCES Musikstil
2.  ON UPDATE CASCADE
3.  ON DELETE CASCADE,
4.
```

DML – Data Manipulation Language

Befehl	Beispiel
INSERT	INSERT INTO <table_name> (column_list) VALUES ..
UPDATE	UPDATE <table_name> SET <column_name> = 'X' WHERE <search_condition>
DELETE	DELETE FROM <table_name> WHERE <search_condition>

```
1.  INSERT INTO Salaer(Monat, Lohn, Anzahl, Tag, Jahr) VALUES(12, 35000, 24, 5, 2003);
2.
3.  UPDATE Salaer SET betrag = 36000;
4.  UPDATE Salaer SET betrag = 1.05 * betrag;
5.
6.  DELETE FROM Salaer WHERE betrag > 100000;
7.
8.  ALTER TABLE Anstellung DROP CONSTRAINT fk_firma;
9.  ALTER TABLE Anstellung ADD CONSTRAINT FK_Firma FOREIGN KEY (FName) REFERENCES Firma(Name);
10.
11. DROP TABLE Mitarbeiter CASCADE;
```

DQL – Data Query Language

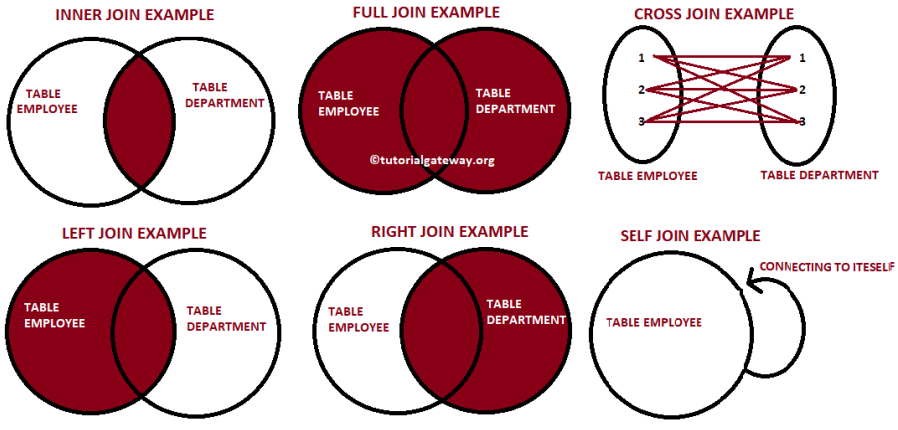
Keyword	Bedeutung
SELECT	Selektion
WHERE	Projektion
EXCEPT	Differenz «ohne»
EXISTS	Existiert ein Element – braucht Suchkondition
GROUP BY	Gruppierung
HAVING	Nach der Gruppierung – braucht Suchkondition
JOIN	Vereinigung
NATURAL JOIN	Vereinigung bei gleichnamigen Attributen
CROSS JOIN	Kreuzprodukt

LIKE

```
1.  SELECT name FROM Mitarbeiter NATURAL JOIN Person WHERE name LIKE  M__er ;
```

_: einzelnes Zeichen
%: beliebige Anzahl Zeichen

Joins



Ordnung von ORDER BY

Wenn verschiedene Tupel gemäss Sortierkriterium gleichwert sind, kann es weiterhin zu Umsortierungen kommen! Man kann dies ausschliessen, indem man nach einem Schlüssel sortiert.

Spalten werden immer in der gleichen Reihenfolge ausgegeben (so wie sie definiert wurden)!

Reihenfolge der Abfrage

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

=> Gruppiert wird erst nach WHERE!

Beispiele

Gesucht ist eine Liste von Restaurants mit ihrem Namen, ihrem Suppenpreis, sowie dem durchschnittlichen Suppenpreis aller Restaurants derselben Strasse

```
1.  SELECT x.Name, x.Suppenpreis, y.Durchschnittspreis FROM Restaurant x
2.  JOIN (
3.    SELECT Strasse, AVG(Suppenpreis) AS Durchschnittspreis FROM Restaurant
4.    GROUP BY Strasse) AS y
5.  ON x.Strasse = y.Strasse;
```

```
-- Zutaten, die in keinem Pizzatyp enthalten sind
-- IDEE: Lister aller Zutaten - liste der verwendeten Zutaten.

SELECT zname
FROM zutat
EXCEPT ALL
SELECT zname
FROM enthält;

-- Eine Liste aller Bestellungen (Kundenname und Bestellnummer) jeweils mit der Summe a

SELECT kname,
       bestnr,
       SUM(menge) AS sumPizzas
FROM umfasst
GROUP BY bestnr,
         kname;

-- Bestellungen, die alle bekannten Pizzatypen umfassen.
-- IDEE: Alle bestellungen bei denene es keinen pizzatypen gibt der nicht existiert.
-- ACHTUNG: Falls in einer Bestellung 0x eine Type einer Pizza bestellt wird, zählt die

SELECT kname, bestnr
FROM umfasst
GROUP BY umfasst.bestnr, umfasst.kname
HAVING NOT EXISTS
    (SELECT pizzatyp.pname
     FROM pizzatyp
     EXCEPT ALL SELECT pname
     FROM umfasst AS umfasst2
     WHERE umfasst2.bestnr = umfasst.bestnr);
```

COALESCE

- Gibt den ersten Wert zurück, der nicht NULL ist

```
1.  SELECT Name,Vorname, SUM(COALESCE(Frequenz,0)) AS AnzahlBesuche FROM Besucher AS b
2.  LEFT OUTER JOIN Gast AS g ON b.Name = g.Bname AND b.Vorname = g.Bvorname
3.  GROUP BY b.Name,b.Vorname
4.  ORDER BY AnzahlBesuche DESC;
```

CASE

Ermöglicht Fallunterscheidungen

```
"CASE"
   "WHEN" <Bedingung1> "THEN" <Wert1>
   {"WHEN" <Bedingung2> "THEN" <Wert2>}
   ["ELSE" <Wert3>]
"END"
```

```
1.  SELECT SID,
2.         Vorname,
3.         Nachname,
4.         (
5.             CASE
6.                 WHEN SUM(Abloese) >= 100000 THEN 'teuer'
7.                 WHEN SUM(Abloese) < 100000 AND SUM(Abloese) >= 50000 THEN 'normal'
8.                 WHEN SUM(Abloese) < 50000 THEN 'billig'
9.                 END
10.            ) as Kategorie
11. FROM Isler.Sportler
12.          NATURAL JOIN Isler.Spieler
13.          NATURAL JOIN Isler.Transfers T
14. WHERE Transferdatum BETWEEN '2010-01-01' AND '2018-12-31'
15. GROUP BY SID
16. HAVING COUNT(TID) > 1;
17.
```

Double not-exists

Finde die Teilenummern aller Teile, welche an alle Projekte in Winterthur geliefert werden

= Es existiert KEIN Projekt in Winterthur, für welches gilt:
Es existiert KEIN Eintrag in LTP mit dieser Teilenummer und dieser Projektnummer

```
SELECT Tnr
FROM T
WHERE NOT EXISTS ( SELECT 1
    FROM P
    WHERE P.Stadt = 'Winterthur'
    AND NOT EXISTS ( SELECT 1
        FROM LTP
        WHERE LTP.PNr = P.PNr
        AND LTP.TNr = T.TNr ));
```

SELECT Tnr FROM T
WHERE NOT EXISTS (
SELECT 1 FROM P WHERE P.STADT = «Winterthur»
AND NOT EXISTS (SELECT 1
FROM LTP WHERE LTP.PNR = P.PNr
AND LTP.TNr = T.TNr));

Abgeleitete Tabellen

```
SELECT
  x.Name, x.Suppenpreis, y.Durchschnittspreis
FROM Restaurant x
JOIN
(
  SELECT Strasse, AVG(Suppenpreis) AS Durchschnittspreis
  FROM Restaurant
  GROUP BY Strasse
) AS y
ON x.Strasse = y.Strasse;
```

Views

snapshot, der aktualisiert wird

- Updaten von views geht nicht allgemein
- Löschen sowieso nicht

Vorteile

- Man kann komplexe Abfragen schrittweise aufbauen
- Man kann Tabellen zusätzliche Attribute hinzufügen und die vorhergehende Version noch als Sicht zur Verfügung stellen.
- Man kann Benutzern Leserechte auf Sichten erteilen, ohne dass sie Zugriffsrechte auf die zugrunde liegenden Tabellen haben müssen. Man kann damit z.B. auch Zugriff nur auf einzelne Attribute bereitstellen.

Nachteile

- Performance: Die Sicht wird bei jeder Verwendung neu berechnet.
- Sichten können nicht sortiert werden (d.h. es ist keine ORDER BY-Klausel in Sichten erlaubt). Es gibt aber Umgehungsmöglichkeiten.
- In der Praxis entsteht oft ein «Wildwuchs» an Sichten, d.h. es braucht eine disziplinierte Organisation und Verwaltung für Sichten

Materialized view

snapshot, der nicht aktualisiert wird

CREATE VIEW <name> AS <Abfrage>
REFRESH materialized view [name]
DROP VIEW <name>

Common table expressions CTE's

- ein **temporäres Resultat einer Abfrage** auf das in einer anderen Abfrage Bezug genommen werden kann.

```
WITH query_name1 AS (
  SELECT ...
),
query_name2 AS (
  SELECT ...
  FROM query_name1
  ...
)
SELECT ...
```

Beispiel für eine CTE

Erstellen Sie eine Liste mit **allen** Spielern (Vor- und Nachname) und wenn möglich dem zugehörigen aktuellen Mannschaftsnamen und dem zugehörigen Trainer (Vor- und Nachname)

```
1.  WITH
2.  Spielername AS(
3.    SELECT SID, Vorname, Nachname
4.    FROM Spieler s JOIN Sportler USING(SID)),
5.  Trainername AS(
6.    SELECT SID, Vorname, Nachname
7.    FROM Trainer JOIN Sportler USING(SID))
8.  SELECT spielername.vorname, spielername.nachname, name, trainername.vorname, trainername.nachname
9.  FROM Spielername
10. LEFT OUTER JOIN gehoertzu USING(SID)
11. LEFT OUTER JOIN Mannschaft USING(MID)
12. LEFT OUTER JOIN trainiert USING(MID)
13. LEFT OUTER JOIN Trainername ON(trainiert.SID = Trainername.SID);
```