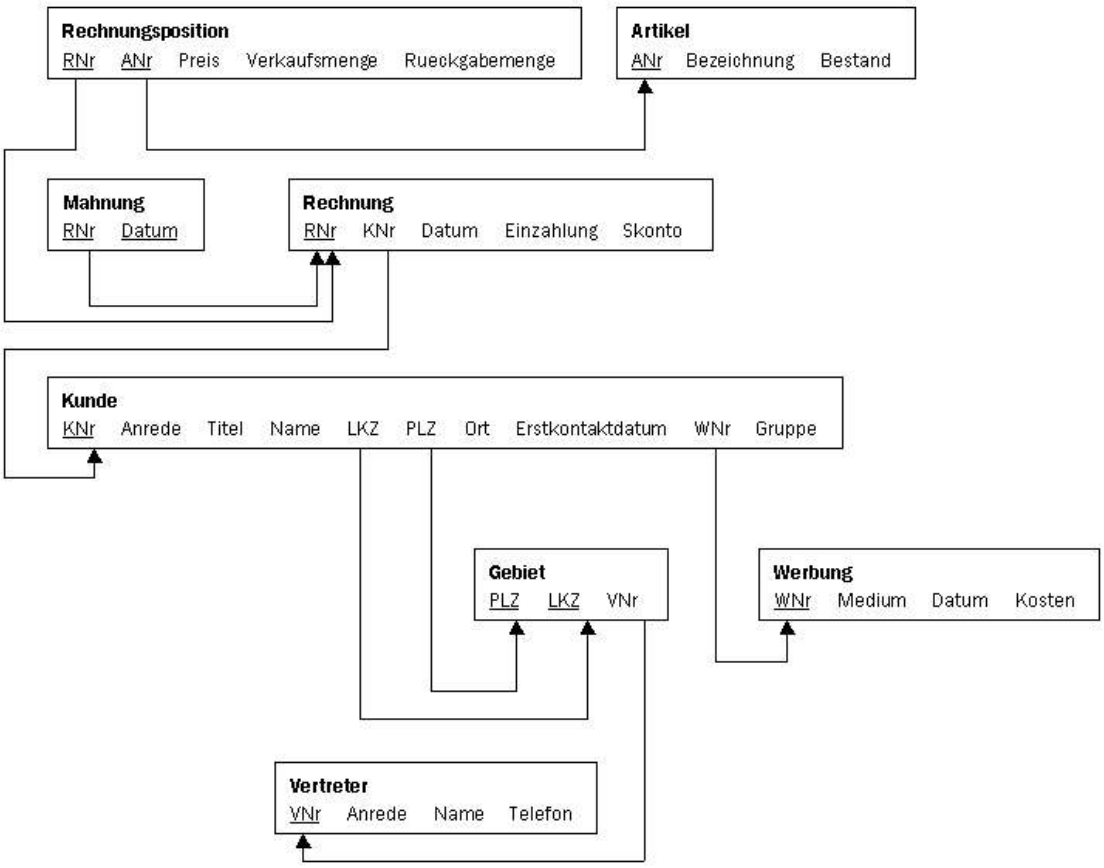
Einführung Datenbanken und SQL

Asvin Goel
goel@uni-koeln.de

21. März 2001



Inhaltsverzeichnis

SQL - Structured Query Language

SQL ist die Datenbanksprache für eine Reihe von relationalen Datenbankmanagementsystemen (DBMS), wie z.B. MySQL, Oracle, Informix.

Literatur: z.B. *Datenbanken und SQL am Beispiel Oracle*, RRZN.

Nach einer kurzen Einführung in die Datenmodellierung werden zunächst die wichtigsten SQL-Befehle zur Abfrage in einer bestehenden Datenbank vorgestellt.

Danach wird auf die Befehle zur Manipulation von Datensätzen und anschliessend auf die Befehle zur Erstellung und Veränderung einer Datenbank eingegangen.

Die gewählte Reihenfolge der vorgestellten SQL-Befehle in diesem Kurs richtet sich nach der typischen Verteilung von Benutzerrechten in Datenbankmanagementsystemen. In diesem Kurs sollen nicht alle Funktionen der vorgestellten Befehle beschrieben werden, sondern vielmehr nur eine Auswahl mit der es möglich sein sollte einfache Projekte selbst zu realisieren.

Als Lernplattform zur Abfrageformulierung wird der **SQL-Trainer** verwendet. Der **SQL-Trainer** ist ein hypermediales Lehrsystem welches am Lehrstuhl für Wirtschaftsinformatik und Operations Research entwickelt wird. Es befindet sich noch in der Entwicklung, so dass darum gebeten wird evtentuell auftretende Fehler, Schwächen und Verbesserungsvorschläge den Entwicklern mitzuteilen.

`http://philostrat.informatik.uni-koeln.de:1080`

1 Datenmodellierung

Beim relationalen Datenbankmodell werden die Datensätze in Tabellen abgelegt, die mit Hilfe von Schlüsselfeldern miteinander verknüpft werden. Andere Datenbankmodelle sind z.B. das hierarchische oder das objektorientierte Datenbankmodell. Auf diese soll hier allerdings nicht eingegangen werden.

Die folgenden Begriffe aus der Theorie und Praxis werden synonym verwendet.

Theorie		Praxis
Relation	$\leftrightarrow$	Tabelle
Tupel	$\leftrightarrow$	Zeile bzw. Datensatz
Tupelelement	$\leftrightarrow$	Spalte bzw. Attribut
Wertebereich	$\leftrightarrow$	Datentyp

1.1 Schlüssel / Primärschlüssel

Um einzelne Datensätze eindeutig zu identifizieren bedient man sich sogenannter Schlüssel.

Eine Attributskombination, die einen Datensatz eindeutig bestimmt, heisst *Schlüsselkandidat*, wenn keine echte Teilmenge von ihr auch den Datensatz eindeutig bestimmt. Oft gibt es mehrere den Datensatz identifizierende Attributskombinationen. In solchen Fällen wird dann einer als *Primärschlüssel* bzw. *primary key* ausgezeichnet.

Attribute die nicht zum Primärschlüssel gehören werden wir *Nichtschlüssel-Attribute* nennen.

Referenziert eine Attributskombination den Primärschlüssel einer anderen Relation, so wird sie *Fremdschlüssel* bzw. *foreign key* genannt.

Faustregel: Der Primärschlüssel sollte nur zur Identifizierung des Datensatzes dienen und keine Informationen enthalten, die für andere Zwecke benötigt werden. Im Notfall sollte man ein zusätzliches Attribut als Primärschlüssel (Index) in die Datenbank aufnehmen.

1.2 Die unnormalisierte Form

Ein trivialer Ansatz eine Kundendatenbank aufzubauen ist, alle den Kunden betreffende Informationen in eine Tabelle abzulegen.

KNr	Name	Ort	RNr	Datum	ANr	Bezeichnung	Verkaufsmenge	Preis
1	Meyer	Koeln	001-2000	1.4.2000	1	Halterung	1	5,00
					2	Schalter	5	9,95
			002-2000	1.5.2000	2	Schalter	3	10,00

Als Primärschlüssel kommt hier z.B. die Kundennummer (KNr) in Betracht.

Nachteile

- Da die Atrribute eine unterschiedliche Anzahl von Elementen haben, ist die Handhabung bei Speicherung und Auswertung unnötig kompliziert.
- **Einfüge-Anomalie:** Wir können nur Artikel mit den entsprechenden Artikelnummern speichern, die schon verkauft worden sind.
- **Änderungs-Anomalie:** Ändert sich eine Artikelbezeichnung, muss die gesamte Datenbank durchsucht und verändert werden.
- **Lösch-Anomalie:** Wir wollen alle Umsätze, die vor dem 1.1.1995 gemacht wurden, aus der Datenbank entfernen. Somit verlieren wir allerdings auch alle Kundeninformationen von Kunden die seit dem 1.1.1995 keinen Umsatz mehr erzielt haben.

Es soll nun in drei Schritten ein Datenmodell erstellt werden, welches diese Nachteile nicht mehr aufweist. Erst der dritte Schritt jedoch sichert, dass die genannten Anomalien nicht mehr auftreten.

1.3 Die 1. Normalform

Eine Relation ist genau dann in der 1. Normalform, wenn alle Attributswerte einelementig oder leer sind.

KNr	Name	Ort	RNr	Datum	ANr	Bezeichnung	Verkaufsmenge	Preis
1	Meyer	Koeln	001-2000	1.4.2000	1	Halterung	1	5,00
1	Meyer	Koeln	001-2000	1.4.2000	2	Schalter	5	9,95
1	Meyer	Koeln	002-2000	1.5.2000	2	Schalter	3	10,00

Achtung: KNr ist nun kein Schlüsselkandidat mehr! Als Primärschlüssel bietet sich nun die Attributskombination Rechnungsnummer (RNr) und Artikelnummer (ANr) an, da jede Rechnung den Kunden eindeutig bestimmt und die Artikelnummel die Rechnungsposition festlegt.

Nachteile

Die oben genannten Anomalien bestehen weiterhin.

1.4 Die 2. Normalform

Eine Relation ist genau dann in der 2. Normalform, wenn sie in der 1. Normalform ist und alle Nichtschlüssel-Atrribute von genau allen Primärschlüsseln abhängig sind.

Vorgehensweise zur Erzeugung der 2. Normalform

- Alle Nichtschlüssel-Attribute auslagern die nicht von genau allen Primärschlüsseln abhängen.
- Erstelle neue Relation für jede logisch zusammenhängende Attributskombination und füge den korrespondierenden Schlüsselteil als Primäschlüssel hinzu. Lösche alle doppelten Zeilen.

RNr	KNr	Name	Ort	Datum
001-2000	1	Meyer	Koeln	1.4.2000
002-2000	1	Meyer	Koeln	1.5.2000

ANr	Bezeichnung
1	Halterung
2	Schalter

RNr	ANr	Verkaufsmenge	Preis
001-2000	1	1	5,00
001-2000	2	5	9,95
002-2000	2	3	10,00

Zur Erzeugung der 2. Normalform muss bekannt sein welche Attribute von welchen abhängen, dies verdeutlicht man sich am besten durch das sogenannte Entity-Relationship Modell (siehe hierzu in der Literatur).

Hier wird der Preis abhängig von der einzelnen Rechnungsposition gesetzt (wg. möglichen Mengenrabatten). Alternativ könnte man die Preise den einzelnen Artikel zuzuordnen, welches jedoch ein anderes Modell zugrunde legen würde.

Hinweis: An dieser Stelle sehen wir, welche Auswirkung die Wahl der Attribute bzw. des Primärschlüssels auf das Datenbankschema hat. Hätten wir in unserer ursprünglichen Modellierung auf das Attribut *ANr* verzichtet, so müssten wir *Bezeichnung* in den Primärschlüssel der 1. Normalform aufnehmen. Bei Übergang zur 2. Normalform würde die Tabelle *ANr, Bezeichnung* wegfallen. In der Tabelle *RNr, Bezeichnung, Verkaufsmenge, Preis* würden alle drei Anomalien auftreten:

- Wir könnten nur Artikel speichern, die schon einmal verkauft wurden (Einfüge-Anomlie).
- Ändert sich eine Artikelbezeichnung, muss die gesamte Tabelle durchsucht und verändert werden (Änderungs-Anomalie):
- Wollen wir alle Informationen über Rechnungen die vor dem 1.1.1995 ausgestellt wurden aus der Datenbank entfernen, würden auch alle Artikel, die seit diesem Zeitpunkt nicht mehr verkauft worden sind, in unserer Datenbank gelöscht (Lösch-Anomalie).

Man überzeuge sich warum die im Primärschlüssel *RNr* versteckte Information '...-2000' über das Jahr der Rechnungsausstellung nicht zu diesen Anomalien führt.

Nachteile

Obwohl die Gefahr der oben genannten Anomalien gemildert wurde, können sie trotzdem noch eintreten.

1.5 Die 3. Normalform

Eine Relation befindet sich genau dann in der 3. Normalform, wenn sie in der 2. Normalform ist und keine transitiven Abhängigkeiten aufweist. Eine transitive Abhängigkeit liegt dann vor, wenn ein Attribut von einem anderen Attribut abhängig ist, welches wiederum von einem dritten Attribut abhängt.

Vorgehensweise zur Erzeugung der 3. Normalform

- Alle Nichtschlüssel-Attribute auslagern die von Attributen abhängen, die nicht zum Primärschlüssel gehören.
- Erstelle neue Relation für jede logisch zusammenhängende ausgelagerte Attributskombination, und füge das diese bestimmende Attribut als Primäschlüssel hinzu. Lösche alle doppelten Zeilen.

KNr	Name	Ort
1	Meyer	Koeln

RNr	KNr	Datum
001-2000	1	1.4.2000
002-2000	1	1.5.2000

ANr	Bezeichnung
1	Halterung
2	Schalter

RNr	ANr	Verkaufsmenge	Preis
001-2000	1	1	5,00
001-2000	2	5	9,95
002-2000	2	3	10,00

Um mit den Tabellen zu arbeiten muessen sie noch benannt werden. Naheliegenderweise nennen wir sie **Kunde, Rechnung, Artikel** und **Rechnungsposition**.

Hinweis: Das im SQL-Trainer zugrundeliegende Datenschema finden Sie auf der Titelseite. Die in diesem Kursskript vorgestellten Beispiele orientieren sich an dem oben erstellten Ausschnitt daraus.

2 Selektion

2.1 Selektion von Zeilen einer Tabelle - SELECT

Eine einfache SQL-Abfrage hat das Muster

`select` <*spalte1, spalte2, ...*> `from` <*tabelle*> `;`

Alle Spalten einer Tabelle kann man sich mit

`select * from` <*tabelle*> `;`

ausgeben lassen.

Beispiel: Es soll eine Liste aller Kunden mit ihren Wohnorten erstellt werden.

```
select NAME,ORT from KUNDE;
```

Abfragen ohne mehrmaligs Auftreten der gleichen Zeilen erhält man durch

`select distinct` <*spalte1, spalte2, ...*> `from` <*tabelle*> `;`

Beispiel: Es soll eine Liste aller Orte in denen Kunden von uns wohnen erstellt werden.

```
select distinct ORT from KUNDE;
```

Mit dem Zusatz `order by` kann man die Abfrage sortiert ausgeben lassen. Aufsteigende (ASC) Sortierung ist der default.

`select` <*spalte1, spalte2, ...*> `from` <*tabelle*>
`order by` <*spalte1* `[ASC | DESC]` , *spalte2* `[ASC | DESC]`, ...>`;`

Beispiel: Es soll eine nach Wohnort sortierte Liste aller Kunden mit ihren Wohnorten erstellt werden.

```
select NAME,ORT from KUNDE order by ORT;
```

Beispiel: Man kann auch Anhand der Spaltenindizes sortieren. Z.B. werden mit der folgenden Abfrage alle Spalten zunächst nach der 2. Spalte und danach nach der 3. Spalte sortiert ausgegeben.

```
select * from ARTIKEL order by 2 DESC, 3 ASC ;
```

Die Spaltenüberschrift bei der Ausgabe kann man selbst festlegen.

`select` <*spalte*> `as` <*neuer_name*> `from` <*tabelle*> `;`

Beispiel: Es soll eine Liste aller Artikel in unserem Sortiment erstellt werden. Diese soll *ARTIKEL* als Überschrift bekommen.

```
select BEZEICHNUNG as ARTIKEL from ARTIKEL;
```

Es ist auch möglich die Ausgabe zu formatieren. Um verschiedene Spalten miteinander verknüpft auszugeben wird der Verkettungsoperator || benutzt.

Beispiel: Zur Erstellung eines Serienbriefes an alle Kunden soll die Anschrift in der Form „Herr/Frau *NAME* , *ORT*" ausgegeben werden.

```
select  'Herr/Frau/Firma '  || NAME || ', ' || ORT as ANSCHRIFT
        from KUNDE;
```

2.2 Selektion mit Bedingungen - WHERE

Mit den bisherigen Abfragen haben wir stets alle Zeilen einer Attributskombination ausgeben lassen. Mit der `where` Bedingung lassen sich gezielt einzelne Zeilen heraussuchen.

`select` <*spalte*> `from` <*tabelle*> `where` <*bedingung*> `;`

Beispiel: Es soll eine Liste aller Kunden mit Wohnort *Berlin* erstellt werden.

```
select NAME, ORT from KUNDE where ORT = 'Berlin';
```

Hinweis: Bei der Abfrage ist auf Gross-/Kleinschreibung zu achten.

Einige der Vergleichsoperatoren sind:

Operatoren	
Gleichheit	=
Ungleichheit	!=
Grösser als	>
Kleiner als	<
Grösser oder gleich	>=
Kleiner oder gleich	<=

Wendet man Vergleiche auf Strings an, so wird lexikografisch verglichen, d.h. 'a' < 'ab' < 'b'.

Allerdings muss hier die Reihenfolge der Zeichen beachtet werden: '0'-'9' < 'A'-'Z' < 'a'-'z'.

Mit dem Operator `between` können alle Spaltenwerte die zwischen zwei Werten liegen selektiert werden.

Beispiel: Es soll eine Liste aller Kunden deren Namen mit *A,...,L* beginnen erstellt werden.

```
select NAME from KUNDE where NAME between 'A' and 'Lzzz';
```

Aus einer Liste von gesuchten Werten kann man mit dem Operator `in` auswählen.

Beispiel: Es soll eine Liste aller Kunden mit Wohnort *Berlin* oder *Milano* erstellt werden.

```
select NAME, ORT from KUNDE where ORT in ('Berlin','Milano');
```

Der `like` Operator ermöglicht den Vegleich eines Strings mit einem Suchmuster. Die Suchmuster werden mit den Platzhalter '_' und '%' gebildet. '_' steht für genau ein beliebiges Zeichen, während '%' für eine beliebig lange (evtl. leere) Zeichenkette steht.

Beispiel: Gesucht ist ein Kunde Namens *Geier, Gaier, Geyer, ...*?

```
select NAME from KUNDE where NAME like 'G%er';
```

bzw.

```
select NAME from KUNDE where NAME like 'G__er';
```

Mehrere Bedingung werden mit `and` bzw. `or` verknüpft.

Beispiel: Gesucht ist ein in *Koeln* wohnender Kunde Namens *Geier, Gaier, Geyer, ...*?

```
select NAME, ORT from KUNDE where NAME like 'G__er'
       and ORT like 'Koeln%';
```

Eine Negation einer Bedingung wird mit `not` erreicht.

Beispiel: Es soll eine Liste aller Kunden deren Wohnort nicht *Koeln* oder ist erstellt werden.

```
select NAME, ORT from KUNDE where ORT not like 'Koeln%';
```

Sind nicht alle Information vorhanden bzw. relevant, so sind typischerweise einige Spalten-Werte nicht gesetzt. Hier ermöglicht das Schlüsselwort `null` den Vergleich.

Beispiel: Gesucht sind alle Kunde deren Wohnort nicht bekannt ist.

```
select * from KUNDE where ORT is null;
```

2.3 Selektion mit Berechnungen

Einzelne Spalten lassen sich mittels $+, -, *, /$ auch arithmetisch miteinander verknüpfen. Das Ergebniss wird dann analog zu einer Spalte benutzt.

Beispiel: Aus Rechnung 2017 sollen die Teilbeträge der einzelnen Posten ermittelt werden.

```
select ANR, VERKAUFSMENGE*PREIS as WERT
       from RECHNUNGSPOSITION where RNR=2017;
```

Eine andere Art von Berechnungen sind die eingebauten Funktionen.

Funktion	
`sum(`<*spalte*>`)`	Summe der einzelnen Spaltenwerte
`avg(`<*spalte*>`)`	Durchschnitt der einzelnen Spaltenwerte
`min(`<*spalte*>`)`	Minimum der einzelnen Spaltenwerte
`max(`<*spalte*>`)`	Maximum der einzelnen Spaltenwerte
`count([distinct]`<*spalte*>`)`	Anzahl der einzelnen (verschiedenen) Spaltenwerte

Hinweis: `null` Werte werden bei diesen Funktionen ignoriert.

Beispiel: Von Artikel 10 soll der durchschnittliche Verkaufspreis ermittelt werden.

```
select sum(VERKAUFSMENGE*PREIS)/sum(VERKAUFSMENGE) as DPREIS
       from RECHNUNGSPOSITION where ANR=10;
```

Beispiel: Von Artikel 10 soll der durchschnittliche Verkaufspreis ermittelt werden. Es soll zusätzlich die Artikelnummer ausgegeben werden.

```
select 10 as ANR,
       sum(VERKAUFSMENGE*PREIS)/sum(VERKAUFSMENGE) as DPREIS
       from RECHNUNGSPOSITION where ANR=10;
```

3 Verbund von Tabellen - Equijoin

Eine SQL-Abfrage mit mehreren Tabellen hat das Muster

`select` *<spalte1, spalte2, ...>* `from` *<tabelle1, tabelle2, ...>* `;`

Haben die Tabellen Spalten mit gleichen Namen, so kann (und muss) man durch hinzufügen des Tabellennamens (z.B. *tabelle1.spalte1*) spezifizieren, welche Spalte welcher Tabelle gemeint ist.

Werden in einer Abfrage mehrere Tabellen angegeben, so wird intern jede Zeile aus Tabelle 1 mit jeder Zeile aus Tabelle 2 usw. verknüpft und bei der Abfrage als einzelne Zeile interpretiert. Zur Veranschauung betrachte man die folgende Abfrage.

`select * from` *<tabelle1, tabelle2, ...>* `;`

Um die zusammengehörigen Datensätze einander zuzuordnen müssen sie noch über die Schlüsselfelder verknüpft werden.

`select` *<spalte1, spalte2, ...>* `from` *<tabelle1, tabelle2, ...>*
`where` *tabelle1.schluessel = tabelle2.schluessel* `and ... ,`

Beispiel: Es soll eine Liste der Kunden und den Rechnungsnummern der an sie erteilten Rechnungen erstellt werden.

```
select NAME, RNR from KUNDE, RECHNUNG
where KUNDE.KNR = RECHNUNG.KNR;
```

Bei längeren Verküpfungen ist es sinnvoll die Tabellennamen mit kurzen Aliasen zu versehen.

`select` *<spalte1, spalte2, ...>* `from` *<tabelle1 alias1, tabelle2 alias2, ...>* `;`

Beispiel: Es soll eine Liste der Kunden und den von ihnen gekauften Artikeln erstellt werden.

```
select K.NAME as KUNDE, A.BEZEICHNUNG as ARTIKEL
from KUNDE K, RECHNUNG R, RECHNUNGSPOSITION P, ARTIKEL A
where K.KNR = R.KNR and R.RNR = P.RNR and P.ANR = A.ANR;
```

4 Vereinigung und Schnitt von Tabellen

4.1 Vereinigung von Tabellen - UNION

Abfrageergebnisse mit gleichen Spalten können mit `union` zusammengefügt werden.

```
select <spalte1, spalte2, ...> from <tabelle1> where <bedingung1>
union
select <spalte1, spalte2, ...> from <tabelle2> where <bedingung2>;
```

Beispiel: Es soll einer Liste aller Kunden und aller Vertreter (aus Tabelle *VERTRETER*) erstellt werden.

```
select NAME from KUNDE
union
select NAME from VERTRETER;
```

Beispiel: Von Artikel 10 und 14 soll der durchschnittliche Verkaufspreis ermittelt werden.

```
select 10 as ANR,
       sum(VERKAUFSMENGE*PREIS)/sum(VERKAUFSMENGE) as DPREIS
       from RECHNUNGSPOSITION where ANR=10
union
select 14 as ANR,
       sum(VERKAUFSMENGE*PREIS)/sum(VERKAUFSMENGE) as DPREIS
       from RECHNUNGSPOSITION where ANR=14;
```

4.2 Durchschnitt von Tabellen - INTERSECT

Die gemeinsamen Zeilen zweier Abfrageergebnisse mit identischen Spalten können mit `intersect` gefunden werden.

```
select <spalte1, spalte2, ...> from <tabelle1> where <bedingung1>
intersect
select <spalte1, spalte2, ...> from <tabelle2> where <bedingung2>;
```

5 Gruppenbildung

5.1 Selektion mit Gruppenbildung - GROUP BY

Bisher wurden die Funktionen immer auf alle Zeilen einer Spalte angewandt. Es ist aber auch möglich eine Tabelle in Gruppen aufzuteilen, auf die die Funktionen dann angewandt werden.

```
select <spalte>, <funktion> as <name> from <tabelle>
       group by <spalte> ;
```

Hinweis: Die in der `group by` angegebene Spalte muss auch in dem ersten Teil der `select` Anweisung angegeben werden.

Beispiel: Es soll eine Liste aller Kunden mit Anzahl ihrer Rechnungen erstellt werden.

```
select KNR, count(RNR) as ANZAHL from RECHNUNG
       group by KNR;
```

Beispiel: Von allen Artikeln soll der durchschnittliche Verkaufspreis ermittelt werden.

```
select ANR,
       sum(VERKAUFSMENGE*PREIS)/sum(VERKAUFSMENGE) as DPREIS
       from RECHNUNGSPOSITION group by ANR;
```

Um nur bestimmte Gruppen in die Betrachtung einzubeziehen kann mit dem Zusatz

`having` eine Bedingung gesetzt werden.

```
select <spalte>, <funktion> as <name> from <tabelle>
       group by <spalte> having <bedingung>;
```

Beispiel: Es soll eine Liste aller Kunden mit Anzahl ihrer Rechnungen erstellt werden. Ausgegeben werden sollen nur die Kunden, die schon mehrere Rechnungen bekommen haben.

```
select KNR, count(RNR) as ANZAHL from RECHNUNG
       group by KNR having count(RNR) >= 2;
```

Beispiel: Von Artikel 10 und 14 soll der durchschnittliche Verkaufspreis ermittelt werden.

```
select ANR,
       sum(VERKAUFSMENGE*PREIS)/sum(VERKAUFSMENGE) as DPREIS
       from RECHNUNGSPOSITION group by ANR;
       having ANR=10 or ANR=14;
```

5.2 Selektion der k grössten bzw. kleinsten Werte

Zur Selektion der k grössten bzw. kleinsten Werte einer Tabelle muss der Rang der Zeilen explizit erzeugt werden.

```
select A.spalte, count(B.spalte) as Rang
       from tabelle A, tabelle B
       where A.vergleichsspalte <= B.vergleichsspalte
       group by A.spalte having count(B.spalte) <= 5;
```

Zunächst wird die Tabelle mit sich selbst verknüpft und nur die Zeilen gewählt, in denen der zweite Wert grösser oder gleich dem ersten ist. Nun zählt man die Zeilen gruppiert nach dem ersten Attribut und wählt nur die Gruppen, in denen der Rang nicht grösser als 5 ist.

Hinweis: Unter Umständen (bei gleichen Werten) werden hier weniger als fünf Zeilen ausgegeben! Im letzten Teil der Abfrage darf (zumindest bei Oracle) nicht `having Rang <= 5;` stehen, der die Spaltenüberschrift bei der Auswertung der Abfrage noch nicht bekannt ist.

Beispiel: Es sollen die fünf teuersten Werbemassnahmen ermittelt werden.

```
select A.WNR, count(B.WNR) as Rang
       from WERBUNG A, WERBUNG B
       where A.KOSTEN <= B.KOSTEN
       group by A.WNR having count(B.WNR) <= 5;
```

6 Unterabfragen

6.1 Verschachtelte Abfragen

Die Ausgabe einer Abfrage ist eine Tabelle die wiederum abgefragt werden kann.

```
select <spalte1, spalte2, ...> from ( select ... );
```

Beispiel: Es soll eine Liste aller Kunden und ihrem Umsatz ausgegeben werden.

Zunächst muss hierzu für alle Rechnungen, die Rechnungsssumme bestimmt werden.

```
select RNR, sum(VERKAUFSMENGE*PREIS) as SUMME
       from RECHNUNGSPOSITION group by RNR;
```

Danach muss diese Liste den einzelnen Kunden zugeordnet werden.

```
select R.KNR, S.SUMME from RECHNUNG R, (
  select RNR, sum(VERKAUFSMENGE*PREIS) as SUMME
  from RECHNUNGSPOSITION group by RNR) S
where R.RNR = S.RNR;
```

Nun können die Rechnungssummen nach Kundennummern gruppiert aufaddiert werden.

```
select KNR, sum(SUMME) as UMSATZ from (
  select R.KNR, S.SUMME from RECHNUNG R, (
    select RNR, sum(VERKAUFSMENGE*PREIS) as SUMME
    from RECHNUNGSPOSITION group by RNR) S
  where R.RNR = S.RNR )
group by KNR;
```

6.2 Der IN Operator

Mit dem `in` Operator ist es möglich nur Zeilen zu selektieren, bei denen ein Attribut im Ergebniss einer zweiten Abfrage enthalten ist.

```
select <spalte1, spalte2, ...> from <tabelle>
       where <spalte> in ( select ... );
```

Hinweis: Die zweite Abfrage muss genau eine Spalte liefern, deren Datentyp mit dem Datentyp von <*spalte*> vergleichbar ist. Ein Vergleich zwischen numerischen Feldern und Textfeldern ist nicht möglich. Jede Unterabfrage mit dem `in` Operator ist auch als equijoin formulierbar. Dies gilt nicht bei der Negation (`not in`).

Beispiel: Gesucht werden alle Kunden die noch keine Rechnung bekommen haben.

```
select NAME from KUNDE  where KNR not in
( select KNR from RECHNUNG );
```

6.3 Unterabfragen mit Vergleich - ANY, ALL

Mit den Operatoren `any` bzw. `any` ist es möglich nur Zeilen zu selektieren, deren Attribut z.B. grösser oder gleich einem beliebigen bzw. allen Werten einer Unterabfrage ist.

```
select <spalte1, spalte2, ...> from <tabelle>
       where <spalte> >= any ( select ... );
select <spalte1, spalte2, ...> from <tabelle>
       where <spalte> >= all ( select ... );
```

Hinweis: Die zweite Abfrage muss genau eine Spalte liefern, deren Datentyp mit dem Datentyp von *<spalte>* vergleichbar ist. Ein Vergleich zwischen numerischen Feldern und Textfeldern ist nicht möglich.

Beispiel: Gesucht werden alle Artikel deren Bestand nicht der kleinste ist (diese Abfrage lässt sich natürlich auch mit der `min` Funktion formulieren).

```
select BEZEICHNUNG from ARTIKEL  where BESTAND > any
( select BESTAND from ARTIKEL );
```

6.4 Der EXISTS Operator

Eine korrelierte Unterabfrage ist eine Unterabfrage deren Ergebnis von der Oberabfrage abhängig ist. Für jedes Tupel der Oberabfrage wird die Unterabfrage neu durchlaufen. Mit dem `exists` Operator ist es möglich die Zeilen zu selektieren, bei denen die korrelierte Unterabfrage mindestens eine Zeile als Ergebnis liefert.

```
select <spalte1, spalte2, ...> from <tabelle1>
       where exists
       ( select <spalte1, spalte2, ...> from <tabelle2>
                where <tabelle1.spalte> = <tabelle2.spalte> );
```

Beispiel: Gesucht werden alle Kunden die ihren Erstkontakt an einem Tag hatten, an dem eine Werbemassnahme stattfand.

```
select NAME from KUNDE  where exists
( select * from WERBUNG where DATUM=ERSTKONTAKDATUM );
```

6.5 Outer Join

Beim equijoin werden nur die Zeilen selektiert, bei denen die Spalten beider Tabellen den gleichen Wert aufweisen. Mit dem outer join ist es möglich auch die Zeilen zu selektieren bei denen in der Verbundspalte ein Nullwert ist, bzw. bei denen kein Verbund hergestellt werden kann.

```
select <tabelle1.spalte1, tabelle2.spalte2>
```

```
from <tabelle1, tabelle2>
       where <tabelle1.spalte> = <tabelle2.spalte>
union
select <tabelle1.spalte1> , '-' from <tabelle1>
       where not exists
       ( select * from <tabelle2>
         where <tabelle1.spalte> = <tabelle2.spalte> );
```

oder alternativ

```
select <tabelle1.spalte1, tabelle2.spalte2>
from <tabelle1, tabelle2>
       where <tabelle1.spalte> = <tabelle2.spalte>
union
select <tabelle1.spalte1> , '-' from <tabelle1>
       where <spalte> not in
       ( select <spalte> from <tabelle2> );
```

Beispiel: Gesucht werden alle Kunden die ihren Erstkontakt an einem Tag hatten, an dem eine Werbemassnahme stattfand. Es soll ggfls. die entsprechende WNR mit ausgegeben werden.

```
select NAME, WERBUNG.WNR from KUNDE, WERBUNG
       where DATUM = ERSTKONTAKDATUM
union
select NAME, '-' from KUNDE  where not exists
( select * from WERBUNG where DATUM = ERSTKONTAKDATUM );
```

7 Erstellen von Sichten - Views

Arbeitet man oft mit bestimmten Datenausschnitten, so ist es nicht sinnvoll den gleichen `select` Befehl immer wieder neu aufzurufen. Hier bietet sich die Möglichkeit einen Ausschnitt fest als Benutzersicht (View) zu definieren.

```
create view <name> as <abfrage> ;
```

Mit <*abfrage*> ist hier eine beliebige `select` Abfrage gemeint.

Beispiel: Um das Kaufverhalten zu untersuchen soll eine Liste der Kunden, der von ihnen gekauften Artikeln, die verkaufte Menge und dem Kaufdatum analysiert werden. Hierzu soll eine Ansicht Kaufverhalten erstellt werden.

```
create view KAUFVERHALTEN as
select K.NAME as KUNDE, A.BEZEICHNUNG as ARTIKEL,
R.DATUM as KAUFDATUM, P.VERKAUFSMENGE as MENGE
from KUNDE K, RECHNUNG R, RECHNUNGSPOSITION P, ARTIKEL A
where K.KNR = R.KNR and R.RNR = P.RNR and P.ANR = A.ANR;
```

Die erstellte Ansicht kann jetzt genau wie eine Tabelle nach ihren Inhalten abgefragt werden.

```
select * from KAUFVERHALTEN;
```

Beispiel: Es soll eine Liste der Artikel erstellt werden, die die Anzahl der verschiedenen Kunden und die von ihnen gekaufte Menge ausgibt. Die Liste soll nach der Anzahl der verkauften Artikel sortiert werden.

```
select ARTIKEL, count(distinct KUNDE) as KUNDEN,
       sum(MENGE) as ANZAHL from KAUFVERHALTEN
       group by ARTIKEL order by ANZAHL DESC;
```

Neben der Vereinfachung von Abfragen können durch Ansichten Daten auch bewusst verborgen werden. Die Abteilung Marketing erhält so z.B. keine Informationen aus der Buchführung, wie z.B. die Rechnungsnummern oder gewährte Rabatte.

Da die Ansicht die Daten nicht selbst speichert, sondern aus anderen Tabellen bzw. Ansichten bezieht können die Daten nicht in der Ansicht selbst verändert werden.

Mit `drop` lässt sich eine solche Ansicht wieder entfernen.

```
drop view <name> ;
```

8 Datenmanipulation

8.1 Eingeben von Datensätzen - INSERT

Mit dem `insert` Befehl werden Datensätze in eine Tabelle eingefügt. Man kann entweder alle Spalten-Werte in der vorgegeben Reihenfolge eingeben,

`insert into` <*tabelle*> `values (`<*wert1, wert2, ...*>`);`

oder die Spalten in die man Eintragungen machen möchte explizit aufführen.

`insert into` <*tabelle*> `(`<*spalte1, spalte2, ...*>`) values (`<*wert1, wert2, ...*>`);`

Beispiel: Frau Schmidt und Herr Mueller (weitere Angaben nicht bekannt) sollen in die Kundenliste aufgenommen werden.

```
insert into KUNDE (KNR,NAME)
       values (6234,'Schmidt');
```

Eleganter ist es jedoch die Kundennummer automatisch generieren zu lassen.

```
insert into KUNDE (KNR,NAME) values
       ( (select max(KNR)+1 from KUNDE) ,'Mueller');
```

8.2 Einfügen von Datensätzen aus anderen Tabellen

Es ist auch möglich ganze Datensätze die aus Abfragen anderer Tabellen gewonnen wurden in eine Tabelle einzufügen.

`insert into` <*tabelle*> (<*spalte1, spalte2, ...*>`)`
`select` <*spalte1, spalte2, ...*> `from` <*tabelle*>`;`

Hinweis: Es ist darauf zu achten, dass durch ein solches Vorghehen nicht unnötige Redundanz auftritt, d.h. dass gleiche Daten nicht unnötig doppelt gespeichert werden. In solchen Fällen sollte man lieber eine entsprechende Ansicht (View) erstellen.

Die hier gezeigte Form des Einfügens von Datensätzen ist insbesondere dann wichtig, wenn das gewählte Datenbankschema nachträglich geändert werden muss.

8.3 Verändern von Datensätzen - UPDATE

Einzelne Spalten-Werte können mit `update` verändert werden.

`update` <*tabelle*> `set` <*spalte1=wert1, ...*> `where` <*bedingung*>`;`

Hinweis: Ohne zusätzliche Bedingung werden die Werte der ausgewählten Spalten in allen Zeilen verändert, daher sollte unbedingt die `where` Bedingung gesetzt werden.

Beispiel: Frau Schmidt wohnt in Berlin.

```
update KUNDE set ORT='Berlin' where KNR=6234;
```

Hinweis: Es sollte unbedingt der Primärschlüssel (also die Kundennummer) zur Identifikation des zu ändernden Datensatzes benutzt werden, da sonst evtl. auch andere Datensätze betroffen sein könnten!

8.4 Löschen von Datensätzen - DELETE

Mit `delete` lassen sich Datensätze aus einer Tabelle löschen.

`delete from` <*tabelle*> `where` <*bedingung*> `;`

Hinweis: Ohne die zusätzliche `where` Bedingung werden alle Zeilen gelöscht.

Beispiel: Frau Schmidt und Herr Mueller sollen aus der Kundeliste entfernt werden.

```
delete from KUNDE where KNR=6234 or KNR=6235;
```

Hinweis: Es sollte unbedingt der Primärschlüssel (also die Kundennummer) zur Identifikation des zu löschenden Datensatzes benutzt werden, da sonst evtl. auch andere Datensätze gelöscht werden könnten!

9 Datendefinition

9.1 Erstellen einer Tabelle - CREATE

Eine Tabelle wird mittels `create` erstellt.

```
create table <tabelle>
             ( <spalte1> <typ1> [default <wert1>] ,
               <spalte2> <typ2> [default <wert2>] ,
               ...
               <constraints> ,
               ... );
```

Wobei <*constraints*> folgende Ausdrücke sein können.

- Mit `not null` wird die Einfügung eines Wertes in eine Spalte erzwungen.
  `constraint` <*name*> `not null (`<*spalte*>`)`
- Mit `unique` wird die Eindeutigkeit der Werte einer Spalte erzwungen.
  `constraint` <*name*> `unique (`<*spalte*>`)`
- Mit `primary key` wird der Primärschlüssel für die Tabelle festgelegt.
  `constraint` <*name*> `primary key (`<*spalte1, spalte2, ...*>`)`
- Mit `foreign key` wird erzwungen das die Einträge in diese Spalte einem Primärschlüssel einer anderen Tabelle entsprechen.
  `constraint` <*name*> `foreign key (`<*spalte1, spalte2, ...*>`)`
  `references` <*fremde_tabelle*> `(`<*fremde_spalte1, fremde_spalte2, ...*>`)`
  `[ on delete { restrict | cascade | set null | set default } ]`
  `[ on update { restrict | cascade | set null | set default } ]`
  Mit den optionalen Zusätzen `on delete` bzw. `on update` ist es möglich beim Verändern der referenzierten Tabelle die angegebenen Aktionen durchzuführen um die Fremd-, Primärschlüssel Integrität zu wahren. Der Versuch einen Primärschlüsseleintrag zu entfernen bzw. ändern wird normalerweise verweigert (`restrict`), falls dieser referenziert wird. Mit `cascade` kann man in solchen Fällen die referenzierende Zeile automatisch löschen bzw. entsprechend ändern lassen. Mit `set null` bzw. `set default` werden die Fremdschlüsseleinträge entsprechend auf `null` bzw. den `default` Wert gesetzt.
- Mit `check` kann beim Einfügen oder Ändern einer Spalte die Einhaltung der angegebenen Bedingung erzwungen werden.
  `constraint` <*name*> `check (`<*bedingung*>`)`

Durch `default` kann man einen Wert spezifizieren, der bei leerer Eingabe in die Spalte eingetragen wird.

Eine kleine Auswahl der verfügbaren Datentypen:

Typ	Beschreibung
`int` bzw. `integer`	Eine ganze Zahl
`decimal`$(m[, n])$	Eine Zahl mit m Vor- und n Nachkommastellen. Wird n weggelassen, so handelt es sich um eine Gleitkommazahl mit m stellen.
`varchar`(n)	Eine Zeichenkette der Länge n
`date`	Ein Datum

Es ist auch möglich eine Abfrage direkt in eine automatisch erstellte Tabelle zu kopieren.

```
create table <tabelle> as select ... ;
```

Es wird so eine Tabelle mit den der Abfrage entsprechenden Spalten angelegt und mit den erfragten Datensätzen aufgefüllt. Die *constraints* werden jedoch nicht übernommen.

Hinweis: Es sei nochmal auf die evtl. unnötig erzeugte Redundanz hingewiesen. Erfüllt eine Ansicht (View) den selben Zweck?

9.2 Verändern einer Tabelle - ALTER

Mit dem `alter` Befehl können Tabellen nachträglich geändert werden. Eine Spalte kann man wie folgt hinzufügen oder ändern.

```
alter table <tabelle> {add | modify}
            ( <spalte> <typ> [default <wert>] );
```

Um eine der Beschränkungen zu verwerfen oder hinzuzufügen benutzt man folgendes Statement.

```
alter table <tabelle> drop constraint <name>;
alter table <tabelle> add constraint <name> ... ;
```

Um eine der Beschränkungen vorübergehend aufzuheben oder wieder zu aktivieren benutzt man folgendes Statement.

```
alter table <tabelle> {disable | enable} constraint <name>;
```

9.3 Löschen Tabelle - DROP

Mit `drop` wird eine Tabelle mit all ihren Einträgen gelöscht.

```
drop table <tabelle> ;
```

10 Zugriff auf fremde Tabellen und Ansichten

Man kann den Zugriff auf eigene Tabellen oder Sichten anderen Benutzern mittel `grant` ermöglichen.

```
grant { all | <befehl> } on <tabelle>
      to { public | <benutzername> }
      [ with grant option ];
```

Als <*befehl*> wird der Befehl, den der angegebene Benutzer auf die Tabelle anwenden darf, angegeben, z.B. `select` oder `insert`. Mit `all` werden alle erteilbaren Rechte dem angegebenen Benutzer erteilt. Wird `public` anstatt <*benutzername*> angegeben, so erhalten alle Benutzer die spezifizierten Rechte. Mit dem Zusatz `with grant option` können die angegeben Benutzer wiederum Rechte auf die Tabelle an andere Benutzer weitergeben.

Beispiel: Der Benutzer *MARKETING* soll Zugriff auf die Ansicht Kaufverhalten erhalten.

```
grant select on KAUFVERHALTEN to MARKETING;
```

Der angegebene Benutzer hat dann die entsprechenden Zugriffsrechte und darf z.B. eine Abfrage auf die Tabelle durchführen.

`select` <*spalte1, spalte2, ...*> `from` <*owner*>.<*tabelle*> `;`

Beispiel: Der Benutzer MARKETING soll nun auf die Ansicht Kaufverhalten zugreifen. Eigentümer dieser Ansicht ist der Benutzer *VERKAUF*.

```
select * from VERKAUF.KAUFVERHALTEN;
```

Die Rechte werden mit `revoke` wieder entzogen.

```
revoke { all | <befehl> } on <tabelle>
       from { public | <benutzername> };
```

Mit `all` werden sinngemäss alle erteilten Rechte auf die angegebene Tabelle dem Benutzer entzogen.