SQL

SQL = Structured Query Language

(SEQUEL)

IBM San Jose Research Laboratory

SYSTEM R

Beispielrelationen

Filiale (Name Leiter Stadt Einlagen)

Konto (KontoNr KundenNr FilialName Saldo)

Kredit (KreditNr Betrag KundenNr FilialName)

Sparbuch (SparbuchNr Guthaben KundenNr FilialName)

Transaktion (vonKtoNr anKtoNr Datum Betrag)

Kunde (KundenNr Name Vorname Straße Stadt)

SQL-2 Datentypen:

- character
- character(*n*)
- character varying(*n*)
- integer oder int
- smallint
- numeric(*m,n*)
- decimal(*n,m*)
- real
- double precision
- float(*m*)
- date
- time
- timestamp

Syntaktische Grundform der SQL-Anfrage

Die Grundform ist eine Selektion gefolgt von einer Projektion:

select	$A_1, A_2, \ldots, A_n$
from	$R_1, R_2, \ldots, R_m$
where	Bedingung;

- **from**-Klausel; muss spezifiziert werden, um die Relationen zu bezeichnen, mit denen gearbeitet werden soll.
- **where**-Klausel; ist optional; damit erfolgt eine Einschränkung der Tupel des Ergebnisses im Sinne einer Selektion.
- **select**-Klausel; ist zwingend notwendig; die hierbei spezifizierten Attribute bewirken eine abschließende Projektion. Falls keine Projektion gewünscht ist kann das Symbol * anstelle der Attributnamen angegeben werden.

Die Ausführung der Anfrage entspricht bis auf die SQL-spezifischen Multisets (Vielfachmengen) dem folgenden Ausdruck der relationalen Algebra:

$$\pi_{A1,A2,...,An} (\sigma_{Bedingung} (R_1 \times R_2 \times \ldots \times R_m))$$

Projektion:

Die Projektion wird durch die Spezifikation der gewünschten Attribute in der **select**-Klausel realisiert.

Beispiel:

```
select Name, Leiter
from Filiale;
```

$\pi_{Name,Leiter}$ (Filiale)

Beispiel:

```
select KundenNr
from Konto;
```

$\pi_{KundenNr}$ (Konto)

SQL verwirklicht das Prinzip der „Vielfachmenge“ (engl. multiset). In den Ergebnismengen können demnach Duplikate auftreten.

Sind keine Duplikate erwünscht, müssen sie explizit durch den Zusatz **distinct** (auch unique) entfernt werden.

```
select distinct KundenNr
from Konto;
```

Selektion:

Die Spezifikation von Selektionen erfolgt in der **where**-Klausel. Bezüglich der Bedingung sind Vergleiche mit den üblichen Operatoren, den logischen Verknüpfungen **and**, **or** und **not** sowie beliebige Klammerungen gestattet.

Beispiel:

```
select *
from Konto
where Saldo > 5.000;
```

$\sigma_{Saldo > 5000}$ (Konto)

Range query

Als Vereinfachung für bestimmte Arten von Abfragen zwischen zwei Grenzwerten steht unter SQL der **between** Operator zur Verfügung.

select KontoNr
from Konto
where Saldo **between** 5.000 **and** 30.000;

$\pi_{KontoNr}$ ($\sigma_{Saldo \geq 5000 \wedge Saldo \leq 30000}$ (Konto))

Kartesisches Produkt:

Werden in der **from**-Klausel mehrere Relationen spezifiziert, so erfolgt die Berechnung des kartesischen Produktes.

Beispiel:

```
select *
from Filiale, Transaktion;
```

Filiale $\times$ Transaktion

Mengenoperationen:

Vereinigungsoperator: Durch den **union**-Operator können einzelne **select**-Anfragen mit attributgleichen und typkompatiblen Ergebnissen miteinander verbunden werden.

Beispiel:

```
select KundenNr, FilialName
from Konto
union
select KundenNr, FilialName
from Kredit;
```

Beachte:

Der **union**-Operator folgt, im Gegensatz zu den meisten anderen SQL-Konstrukten, nicht dem Ansatz der Vielfachmenge!

Will man eventuelle Duplikate nicht beseitigen, so ist **union all** zu verwenden.

Beispiel:

```
select KundenNr, FilialName
from Konto
union all
select KundenNr, FilialName
from Kredit;
```

Durchschnitts und Differenzbildung:

Die diesbezüglichen Operatoren sind **intersect** und **except**.

Verbindungsoperationen:

Theta-Verbund:

Werden in der **where**-Klausel Attribute unterschiedlicher Relationen miteinander verglichen, entspricht dies einer Verbund-Operation. Da beliebige Vergleichsoperatoren verwendet werden können, entspricht dies einem allgemeinen Theta-Verbund.

Beispiel:

```
select Name, Vorname, Straße, Stadt
from Filiale FI, Konto KO, Kunde KU
where KU.KundenNr = KO.KundenNr
and FI.Name = „Zeil“;
```

Die in zwei der beteiligten Relationen vorkommenden Attribute KundenNr und Name werden durch Bindung an die Relation eindeutig qualifiziert (renaming).

Die Qualifizierung kann durch den Relationennamen selbst oder wie hier durch einen für die Relation vergebenen *Aliasnamen* erfolgen.

Schachtelung von Abfragen:

Analog zur Vorgehensweise im Tupelkalkül können Anfragen geschachtelt werden.

Beispiel:

```
select Name, Vorname, Straße, Stadt
from Kunde KU
where not exists (
  select *
  from Kredit KR
  where KU.KundenNr = KR.KundenNr);
```

Ergebnis: Alle Kunden, die (noch) keinen Kredit haben.

Beispiel:

Suche alle Kunden, die sowohl ein Konto als auch ein Sparbuch haben.

```
select Name, Vorname
from Kunde KU
where exists (
      select *
      from Konto KO
      where KU.KundenNr = KO.KundenNr
      and exists (
            select *
            from Sparbuch SP
            where
              SP.KundenNr = KO.KundenNr
      ) ) ;
```

Beispiel:

Suche alle Filialen, bei denen mindestens ein Kunde ein Konto hat.

```
select *
from Filiale
where Name in (
      select distinct FilialName
      from Konto);
```

Beispiel:

Suche alle Kunden, die in der „Zeil“-Filiale ein Konto und einen Kredit haben.

```
select distinct KundenNr
from Konto
where FilialName = “Zeil”
and KundenNr in (
      select distinct KundenNr
      from Kredit
      where FilialName = “Zeil”);
```

Mengenvergleiche:

Mehr Möglichkeiten als der **in**-Operator bietet der θ**any** und der θ**all**-Operator, also der Vergleich mit *irgendeinem* oder *jedem* Tupel der Unteranfrage.

Alle Konten außer das, mit dem größten Saldo.

```
select *
from Konto
where Saldo < any ( /* kleiner als irgendeines */
      select Saldo
      from Konto);
```

Das Konto mit dem kleinsten Saldo.

```
select *
from Konto
where Saldo <= all ( /* kleiner gleich jedem */
      select Saldo
      from Konto);
```

Der Vergleich = **any** entspricht dabei dem **in**-Operator und der Vergleich <> **all** entspricht **not in**.

Aggregatfunktionen:

Eine Besonderheit von SQL sind die Operatoren, die Berechnungen über Gruppen von Tupeln anstellen.

Die sog. Aggregatfunktionen können in der **select**-Klausel anstelle von einzelnen Attributen angegeben werden.

- **min**(A) zur Berechnung des Minimalwerts aller Tupel unter dem Attribut A.
- **max**(A) zur Berechnung des Maximalwerts aller Tupel unter dem Attribut A.
- **avg**([**distinct**] A) zur Berechnung des Durchschnittswerts aller Tupel unter dem Attribut A, wobei unter Angabe von distinct mehrfach gleiche Werte nur einmal in die Berechnung eingehen.

Ergebnis einer Aggregatfunktion ist ein Wert, kein Tupel.

- **sum**([**distinct**] A) zur Berechnung der Summe aller Tupel unter dem Attribut A, wobei unter Angabe von distinct mehrfach gleiche Werte nur einmal in die Berechnung eingehen.
- **count**(∗) zum Zählen der Tupel der betrachteten Relation.
- **count**([**distinct**] A) zum Zählen der Tupel der betrachteten Relation, wobei zunächst eine Duplikateneliminierung bezogen auf Werte unter dem Attribut A stattfindet.

Beispiel:

```
select count(∗) as AnzahlKonten
from Konto;
```

Beispiel:

```
select count(distinct KundenName)
from Konto;
```

Beispiel:

```
select sum(Saldo) as Gesammtguthaben
from Konto
where FilialName = “Zeil”;
```

Sortierung:

Eine praktische Eigenschaft von SQL ist die Möglichkeit die Ergebnisrelationen einer Anfrage sortieren zu lassen, auch wenn der Mengencharakter der Relationen dadurch aufgeweicht wird. Es ist sowohl aufsteigendes als auch absteigendes Sortieren möglich (Schlüsselworte **asc** und **desc**).

Beispiel:

```
select *
from Kredit
order by FilialName, KreditNr;
```

Es wird hierbei in Reihenfolge der angegebenen Attribute sortiert, also zuerst nach Filialnamen und dann (bei gleichen Filialnamen) nach der Kreditnummer.

Beispielrelationen

Filiale (Name Leiter Stadt Einlagen)

Konto (KtoNr KundenName FilialName Saldo)

Kredit (KreditNr Betrag KundenName FilialName)

Sparbuch (SparbuchNr Guthaben KundenName FilialName)

Transaktion (vonKtoNr anKtoNr Datum Betrag)

Kunde (Name Vorname Straße Stadt Geb-Datum)

Tabellendefinition

Eine Tabelle wird im Minimalfall mit ihrem eindeutigen Namen sowie der Liste der zugehörigen Attribute samt Domänen nach folgendem Schema definiert:

create table *Relations*-N*ame* (
  Attribut-Name Domäne { , *Attribut-Name Domäne*}∗
) ;

Beispiel:

create table Konto (
  KtoNr **integer**,
  KundenName **char**(25),
  FilialName **char**(25),
  Saldo **real**
) ;

create table Kunde (
  Name **char**(25),
  Vorname **char**(25),
  Straße **char**(25),
  Stadt **char**(25),
  Geb-Datum **date**
) ;

Primärschlüssel

Mittels der Klausel **primary key** kann eine unter den Attributfolgen einer Relation – bei der Definition der Tabelle – als Primärschlüssel ausgezeichnet werden. Die Benutzung dieser Klausel ist nur einmal pro Relation gestattet.

Beispiel:

```
create table Konto (
   KtoNr integer primary key,
   KundenName char(25),
   FilialName char(25),
   Saldo real
) ;
```

Wenn mehr als ein Attribut als Primärschlüssel definiert werden sollen, wird die Klausel in der Form **primary key** (*Attributnamen-Liste*) verwendet.

Beispiel:

```
create table Transaktion (
   vonKtoNr integer,
   anKtoNr integer,
   Datum date,
   Betrag real,
   primary key (vonKtoNr, anKtoNr, Datum)
) ;
```

Ändern der Daten

Löschen von Tupeln

Das Löschen von Tupeln ist einfach. Eine Löschanfrage ist einer normalen Anfrage ähnlich. Es können allerdings nur ganze Tupel gelöscht werden.

delete *Relations-Name*
where *Bedingung* ;

Es werden alle Tupel in *Relationsname* gelöscht, für die die *Bedingung* erfüllt ist.

Beispiele:

delete Konto ;

→ löscht alle Tupel der Relation Konto.

delete Konto
where KundeName = "Otto" ;

→ löscht alle Konten des Kunden mit dem Namen "Otto" .

Beispiele:

```
delete Konto
where KtoNr >= 1300 and KtoNr < 1500 ;
```

→ löscht alle Konten mit Nummern zwischen 1300 und 1500.

```
delete Konto
where FilialName in (
  select FilialName
  from Filiale
  where Stadt = "Frankfurt" ) ;
```

→ löscht die Konten von allen Filialen in Frankfurt.

```
delete Konto
where not exists (
        select *
        from Kunde
        where Kunde.Name = Konto.KundenName
) ;
```

→ löscht die Konten für die es keinen entsprechenden Datensatz in der Relation Kunde gibt.

Mögliche Anomalien

Wenn die Löschanfrage in einer Unteranfrage die gleiche Relation referenziert, wie die, in der gelöscht wird, besteht die Gefahr von Anomalien.

Beispiel:

```
delete Konto
where Saldo < (
  select avg(Saldo)
  from Konto ) ;
```

Offensichtlich ändert sich der Durchschnitt der Salden, wenn Tupel gelöscht werden! Wenn das **select** für jedes Tupel neu berechnet wird, ist das Ergebnis von der Reihenfolge der bearbeiteten Tupel abhängig.

Regel zur Vermeidung von Anomalien

Durch die folgende einfache **Regel** vermeidet man solche Anomalien:

Während der Ausführung der Löschanfrage, werden die Tupel nicht wirklich gelöscht, sondern nur markiert.
Erst wenn die Anfrage beendet ist, werden die markierten Tupel gelöscht.

Einfügen von Tupeln

Um Daten einzufügen, spezifiziert man entweder das Tupel, das eingefügt werden soll oder schreibt eine Anfrage, die eine Menge von Tupeln als Ergebnis hat, die eingefügt werden soll.

Die Werte für die Attribute der Tupel müssen aus der Domäne der Attribute sein.

Beispiele:

```
insert into Kunde
  values ( "Otto", "Hans", "Bäckerweg 12",
           "Frankfurt", "29.2.1970" ) ;
```

→ Der Kunde "Hans Otto" wird eingefügt.

```
insert into Sparbuch
        select KreditNr, 50.00, KundenName, FilialName
        from Kredit
        where FilialName in (
                select FilialName
                from Filiale
                where Stadt = "Frankfurt" );
```

→ Alle Kunden, die bei Filialen in Frankfurt einen Kredit haben, bekommen ein Sparbuch mit einem Startkapital von 50,- €.

Modifizieren von Tupeln

Es gibt Situationen, in denen nur ein bestimmter Wert eines Tupels geändert werden soll. Für diesen Fall gibt es die **update**-Anweisung. Wie bei **insert** und **delete** können die betroffenen Tupel per **select** ausgewählt werden.

Beispiele:

```
update Konto
set Saldo = Saldo * 1.05 ;
```

→ Dieses Update verändert jedes Tupel in der Relation Konto.

Angenommen, man wollte für alle Konten mit einem Saldo über 10.000 den Saldo um 6% erhöhen und für die anderen um 5%:

```
update Konto
set Saldo = Saldo * 1.06
where Saldo > 10000 ;

update Konto
set Saldo = Saldo * 1.05
where Saldo <= 10000 ;
```

In diesem Fall ist die Reihenfolge der Updates wichtig für das Ergebnis. Vertauscht man die Reihenfolge, werden einige Konten um 11,3% erhöht!!

Sichten (views)

Ein wichtiges Konzept, um eine Datenbank an die Bedürfnisse der Benutzer anpassen zu können, sind **Views**.

Eine View ist eine Relation, die nicht Teil des **konzeptuellen Schemas** der Datenbank ist, sondern dem Benutzer als **virtuelle Relation** zur Verfügung gestellt wird. Views können nicht gespeichert werden sondern müssen für jede Anfrage, die sie referenziert **neu berechnet** werden.

In SQL wird eine View folgendermaßen definiert:

create view *View-Name* **as**
<Anfrage-Ausdruck>

Beispiel:

```
create view alleKunden as
  ( select KundenName, FilialName
  from Konto )
  union
  ( select KundenName, FilialName
  from Kredit ) ;
```

Ab jetzt kann die View in Anfragen benutzt werden:

```
select KundenName
from alleKunden
where FilialName = "Westend" ;
```

Probleme mit Views

Obwohl Views sehr nützlich sein können, bringen sie Probleme mit, wenn sie in **update**, **insert** oder **delete**-Anfragen verwendet werden.

Das Problem ist, die Veränderungen in den virtuellen Relationen auf die wirklichen Relationen, die im konzeptuellen Schema vorhanden sind, zu übertragen.

Beispiel:

```
create view Kredit_Info as
select FilialName, KreditNr, KundenName
from Kredit ;
```

Da SQL es erlaubt, eine View in jedem Ausdruck zu verwenden, könnte man versucht sein, etwa Folgendes auszuführen:

```
insert into Kredit_Info
values ("Westend", 143, "Meiser");
```

Diese Einfüge-Operation muss zu einem **insert** in der Relation Kredit führen, dafür fehlt allerdings der Wert für das Attribut Betrag.

Es gibt zwei unterschiedliche Ansätze, um mit diesem Problem umzugehen:

1. Zurückweisen der Operation, und eine Fehlermeldung an den Benutzer.
2. Einfügen des Tupels (143, **null**, "Meiser", "Westend") in die Relation Kredit.

Der **null**-Wert repräsentiert hierbei einen nicht existierenden Wert.

Null-Werte

Der Umgang mit **Null**-Werten in Datenbanken ist im Allgemeinen nicht trivial, Insbesondere bei den **Aggregatfunktionen** und **Vergleichen** sind Null-Werte problematisch.

Beispiel:

```
select sum (Betrag)
from Kredit ;
```

Die Summe der Beträge ist schwierig zu berechnen, wenn für das Attribut „Betrag“ Null-Werte gespeichert sind.

→ Alle Aggregatfunktionen (außer **count**) ignorieren Tupel mit Null-Werten.

→ Alle Vergleiche mit Null-Werten sind per Definition **falsch**. Um auf Null-Werte testen zu können gibt es deshalb das spezielle Schlüsselwort **null**.

Beispiel:

```
select KundenName
from Kredit
where Betrag is null ;
```

Beispiel:

```
create view Filiale-Stadt as
select FilialName, Stadt
from Kredit, Kunde
where Kredit.KundenName = Kunde.Name ;
```

Betrachtet man nun folgende Einfüge-Operation auf dieser View:

```
insert into Filiale-Stadt
values ( "Zeil", "Frankfurt" ) ;
```

Die einzige Möglichkeit, Tupel in die darunter liegenden Relationen Kredit und Kunde einzufügen, ist es, das Tupel (null,null,null,"Zeil") in Kredit und (null,null,null,"Frankfurt",null) in Kunde einzufügen. Als Ergebnis erhielte man die folgenden Relationen:

KreditNr	Betrag	KundenNa-me	FilialName
124	2000,00	Otto	Westend
125	47,38	Kunze	Mitte
…	…	…	…
null	null	null	Zeil

Name	Vorname	Straße	Stadt	Geb-Datum
Otto	Hans	Waldweg	Frankfurt	8.1.1960
Kunze	Elke	Marktstr.	Offenbach	7.7.1970
…	…	…	…	…
null	null	null	Frankfurt	null

Das Tupel ("Zeil", "Frankfurt") würde nun in der Anfrage

```
select *
from Filiale-Stadt ;
```

nicht mehr vorkommen, da der Vergleich
Kredit.KundenName = Kunde.Name
aufgrund von null-Werten in den entsprechenden Tupeln nicht wahr werden kann.

Aufgrund dieser Probleme gilt für viele Datenbanken folgende Einschränkung:

- Eine Modifikation durch eine View ist nur dann erlaubt, wenn die betroffene View auf einer einzigen Relation der aktuellen Datenbank definiert ist.

Zusammenfassung:
Views sind nützliche Mechanismen, um Anfragen an Datenbanken zu vereinfachen, Modifikationen auf der Datenbank durch Views auszuführen kann aber zu Problemen führen. Modifikationen sollten deshalb nur auf echten Relationen der Datenbank ausgeführt werden.

Persistenz

Jede mit mittels **create table** erstellte Tabelle ist ohne weiteres Zutun persistent. Alle Einfügungen, Änderungen oder Löschungen aus solchen Tabellen haben ebenfalls dauerhafte, die Lebensdauer des diese Befehle ausführenden Programms überdauernde Wirkung.

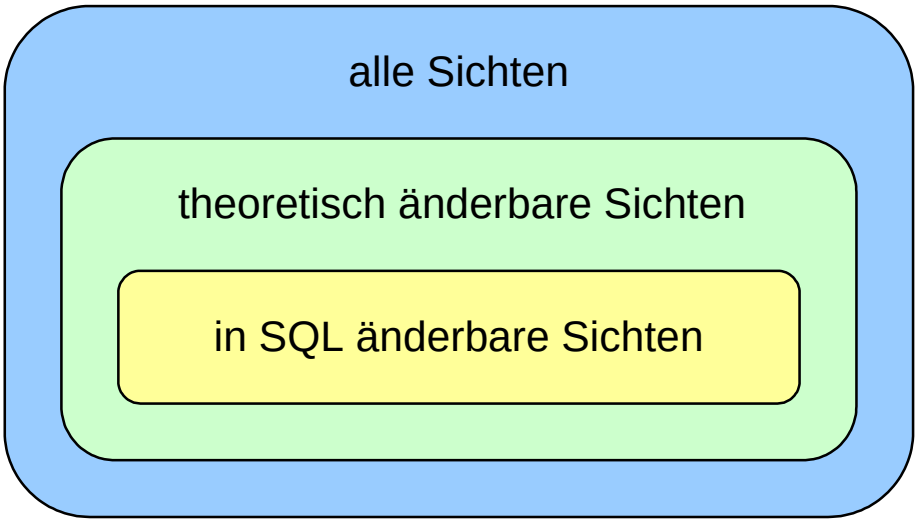
Es kann aber sein, dass Zwischenergebnisse zu berechnen und zu verarbeiten sind, die ebenfalls den Charakter einer Relation haben. Hier würde sich die Einführung von Relationen mit eingeschränkter Lebensdauer anbieten.

In SQL-2 ist dies durch die Angabe des Schlüsselwortes **temporary** bei Ausführung von **create table** möglich:

create table *Relations*-N*ame* (

Attribut-Name Domäne { , *Attribut-Name Domäne*}∗

) **temporary** ;


alle Sichten
theoretisch änderbare Sichten
in SQL änderbare Sichten