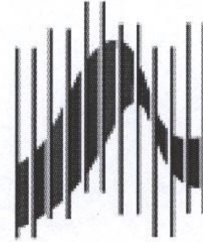
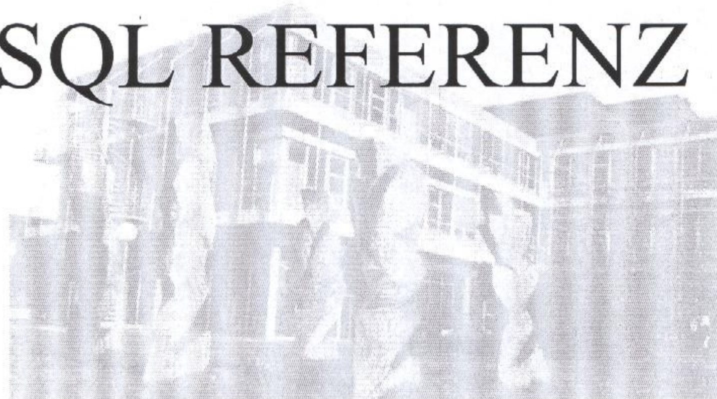


FH Weingarten

Datenbanksysteme

SQL REFERENZ



SQL Referenz

I. Inhaltsverzeichnis

I. Inhaltsverzeichnis ... - 2 -
1. Die wichtigsten SQL-Kommandos ... - 3 -
1.1 CREATE TABLE ... - 3 -
1.2 ALTER TABLE ... - 4 -
1.3 DROP TABLE ... - 4 -
1.4 INSERT INTO ... - 4 -
1.5 UPDATE ... - 5 -
1.6 COMMIT ... - 5 -
1.7 CREATE VIEW ... - 5 -
1.8 DROP VIEW ... - 6 -
2. Einige ausgewählte SQL-Plus-Kommandos ... - 6 -
2.1 CONNECT ... - 6 -
2.2 START ... - 6 -
2.3 QUIT ... - 6 -
3. Das Select-Kommando mit seinen Komponenten ... - 7 -
3.1 Die SELECT-Komponente ... - 7 -
3.2 Die FROM-Komponente ... - 7 -
3.3 Die WHERE-Komponente ... - 8 -
3.3.1 Einfache und zusammengesetzte Vergleichsausdrücke ... - 8 -
3.3.2 Das BETWEEN-Prädikat ... - 9 -
3.3.3 Das IN-Prädikat ... - 10 -
3.3.4 Das EXISTS-Prädikat ... - 10 -
3.3.5 Das LIKE-Prädikat ... - 10 -
3.4 Die GROUP BY-Komponente ... - 11 -
3.5 Die ORDER BY-Komponente ... - 11 -
4. Eingebaute SQL-Funktionen ... - 12 -
4.1 COUNT ... - 12 -
4.2 SUM ... - 12 -
4.3 MIN, MAX ... - 12 -
4.4 AVG ... - 12 -
5. Trigger ... - 13 -
5.1 Aufbau eines Trigger ... - 13 -
5.2 PL-SQL Programmcode ... - 13 -
5.3 Lösung zu Übungsblatt 5: Datenbanktrigger ... **Error! Bookmark not defined.**
5.3.1/2 Lösung Aufgaben 1 und 2 ... **Error! Bookmark not defined.**
5.3.3 Lösung zu Aufgabe 3 ... **Error! Bookmark not defined.**
5.3.4 Lösung zu Aufgabe 4 ... **Error! Bookmark not defined.**
6. Embedded SQL ... **Error! Bookmark not defined.**
6.1 Struktureller Aufbau eines SQL Programmes: ... **Error! Bookmark not defined.**
6.2.1 Beispiel ... **Error! Bookmark not defined.**
6.3 Embedded SQL-Befehle ... **Error! Bookmark not defined.**
7. ODBC ... **Error! Bookmark not defined.**
7.1 Aufbau eines ODBC C-Programms ... **Error! Bookmark not defined.**
7.1.1 Includen der Header Files ... **Error! Bookmark not defined.**
7.1.2 Handle Einrichten (als globale Variablen) ... **Error! Bookmark not defined.**
7.1.3 Verbindung aufbauen / trennen ... **Error! Bookmark not defined.**
7.1.4 Spalten an C-Variablen Binden ... **Error! Bookmark not defined.**
7.1.5 Abfrage ausführen ... **Error! Bookmark not defined.**
7.1.6 Mit Datensätzen arbeiten ... **Error! Bookmark not defined.**

JDBC: http://www.informatik.uni-freiburg.de/~dbis/lehre/db-prakt-sql-ss01/2706.pdf

SQL Referenz

1. Die wichtigsten SQL-Kommandos

1.1 CREATE TABLE

CREATE TABLE <Tabellenname> [(Spaltenname Datentyp [,Spaltenname, Datentyp]...)];

Erzeugen einer neuen Tabelle.

Beispiel:

```
CREATE TABLE Kunde

        (Kunden_Nr CHAR(4),
        Vorname CHAR(15),
        Name  CHAR(15),
        Strasse  CHAR(15),
        PLZ  CHAR(6),
        Ort  CHAR(15));
```

Wichtige Datentypen:

INTEGER	numerisch, Wertebereich - 9.999.999.999 ... 99.999.999.999	
DECIMAL(p, q)	numerisch, p: 1 .. 19, q: 0 .. 18	Beispiel: -0.97856
NUMBER(p, q)	numerisch, p: 1 .. 20, q: 0 .. 18	Beispiel: 0.9e+24
CHAR(n)	Zeichenkette, n: 1 .. 254	Beispiel: 'Testwort'
DATE	Datum	Beispiel: '25.04.96'

SQL Referenz

1.2 ALTER TABLE

ALTER TABLE <Tabellenname>
    ADD | MODIFY (<Spaltenname> <Datentyp> [,<Spaltenname> <Datentyp>] ...);

Hinzufügen von Spalten.

Beipiel:

```
ALTER TABLE Kunde

        ADD (GebDatum DATE);
```

1.3 DROP TABLE

DROP TABLE <Tabellenname>;

Löschen einer Tabelle.

Beispiel:

```
DROP TABLE Kunde;
```

1.4 INSERT INTO

INSERT INTO <Tabellenname> [(<Spaltenliste>)]
    VALUES (<Werteliste>) | SELECT-Befehl;

Fügt eine Reihe von Daten in die Tabelle ein.

Beispiel:

```
INSERT INTO Kunde

        VALUES ('K001', 'Heiner', 'Müller', 'Am Burggraben', '02345',
        'Wallstedt');
```

SQL Referenz

1.5 UPDATE

UPDATE <Tabellenname>
SET <Spaltenname> = <Ausdruck> [,<Spaltenname> = <Ausdruck>] ...
[WHERE <Bedingungsausdruck>];

Verändern bestehender Zeilen.

Beispiel:

```
UPDATE Kunde

    SET  Zuname = 'Meier'
    WHERE Kunden_Nr = 'K001';
```

1.6 COMMIT

COMMIT;

- alle innerhalb einer Transaktion gemachten Änderungen bleibend machen
- eine Transaktion beenden

1.7 CREATE VIEW

CREATE VIEW <Viewname> [(<Spaltenliste>)]
AS <SELECT-Angabe>;

Erzeugen einer Sicht (virtuelle Tabelle).

Beispiel:

```
CREATE VIEW KundView (Rufname, Familienname)

    AS SELECT Vorname, Zuname
    FROM Kunde;
```

SQL Referenz

1.8 DROP VIEW

DROP VIEW <Viewname>;

Löschen einer Sicht (virtuellen Tabelle).

Beispiel:

```
DROP VIEW KundView;
```

2. Einige ausgewählte SQL-Plus-Kommandos

SQL-Plus-Kommandos müssen nicht mit einem Semikolon abgeschlossen werden.

2.1 CONNECT

CONNECT <Datenbankname>

Stellt eine Verbindung zu der angegebenen Datenbank her. Eine bestehende Verbindung wird beendet.

2.2 START

START <Filename>

Führt die in der Textdatei <Filename> gespeicherten SQL-Kommandos aus.

2.3 QUIT

QUIT

Verlassen von SQL-Plus

SQL Referenz

3. Das Select-Kommando mit seinen Komponenten

```
SELECT [DISTINCT] * | <Spaltenliste>
       FROM <Tabellenliste>
       [ WHERE <Ausdrucksliste> ]
       [ GROUP BY <Gruppierungsliste> ]
       [ ORDER BY <Spaltenname> [ASC | DESC][, <Spaltenname> [ASC | DESC]
]... ;
```

3.1 Die SELECT-Komponente

DISTINCT (engl.: unterschiedlich) verhindert, daß im Ergebnis mehrere Zeilen mit denselben Werten vorkommen.

In der <Spaltenliste> werden die Namen der gewünschten Spalten der in der <Tabellenliste> angegebenen Tabellen aufgeführt:

[<Qualifikator>.] <Spaltenname> [, [<Qualifikator>.]<Spaltenname>] ...

Vor dem Spaltennamen kann ein s.g. ***Qualifikator*** stehen. Das ist eine Namenserweiterung, damit der Spaltenname eindeutig wird.

Ein Qualifikator kann entweder der vollständige Tabellenname oder ein in der <Tabellenliste> festlegbarer Aliasname der Tabelle sein. Ein ***Aliasname*** ist nur innerhalb eines Select-Befehles gültig !

Beispiel: siehe Beispiel für Vergleichsausdrücke (weiter unten)

Wird anstelle der <Spaltenliste> der Stern (*) angegeben, so werden ***alle Spalten*** der in der Tabellenliste verzeichneten Tabellen zurückgegeben. (keine Auswahl von Spalten)

3.2 Die FROM-Komponente

In der <Tabellenliste> werden die Tabellennamen angegeben, auf die sich der SELECT-Befehl bezieht. Mehr als ein Tabellenname erzeugt aus diesen Tabellen einen s.g. ***Verbund*** (engl.: ***join***)

SQL Referenz

3.3 Die WHERE-Komponente

Der WHERE-Komponente folgt eine <Ausdrucksliste>, die die ***Suchbedingung*** des SELECT-Befehls darstellt.
Die Auswertung der Suchbedingung ergibt entweder den Wert wahr oder falsch. Dadurch wird entschieden, ob eine Zeile in das Ergebnis des SELECT-Befehls aufgenommen wird oder nicht.
Mit Suchbedingungen werden ***Prädikate*** geschrieben. Das sind Aussagen über eine Eigenschaft einer Zeile.
Ein Prädikat kann aus mehreren Teilprädikaten bestehen, die über logische Operatoren (AND / OR / NOT) miteinander verknüpft sind.

Prädikat	Beispiel
einfacher Vergleichsausdruck	Preis > 1000
zusammengesetzter Vergleichsausdruck	Name >= 'A' AND Name < 'N'
BETWEEN-Prädikat	Betrag BETWEEN 100 AND 1000
IN-Prädikat	Ort IN ('Halle', 'Merseburg', 'Querfurt')
EXISTS-Prädikat	EXISTS (Select Kunden_Nr FROM Rechnung)
LIKE-Prädikat	Titel LIKE '%Oracle%'

3.3.1 Einfache und zusammengesetzte Vergleichsausdrücke

Vergleichsoperatoren:

= gleich
< kleiner
> größer
<= kleiner oder gleich
>= größer oder gleich
<> ungleich

Einer der Operanden in einem Vergleichsausdruck ***muß*** einen Spaltennamen enthalten !

In der WHERE-Komponente eines SELECT-Befehls ist es möglich, einen weiteren SELECT-Befehl anzugeben. Dieser geschachtelte SELECT-Befehl wird ***Unterabfrage*** (engl: subquery) genannt.

Beispiel: Kundendaten des Kunden mit dem höchsten Rechnungsbetrag, mindestens jedoch 1000,-DM

```
SELECT Kunden_Nr, Vorname, Name, Ort

    FROM Kunde K
    WHERE 1000 <= (SELECT MAX (Betrag)

                        FROM Rechnung R
                        WHERE R.Kunden_Nr =
                        K.Kunden_Nr);
```

In dieser Unterabfrage wurde zusätzlich eine SQL-Funktion (MAX) verwendet, die im obigen Falle den größten Wert der Spalte mit dem Namen Betrag ermittelt. Außerdem wurde mit ***Alias-Namen*** für die beiden Tabellen Rechnung und Kunde gearbeitet.

3.3.2 Das BETWEEN-Prädikat

Mit dem BETWEEN-Prädiakt kann geprüft werden, ob ein Wert innerhalb eines Intervalls (Bereichsgrenzen eingeschlossen) liegt.
Mit **NOT BETWEEN** kann leicht getestet werden, ob ein Wert außerhalb eines Intervalls liegt.

Beispiel: Kundendaten der Kunden, deren Kundennummer nicht im Bereich K100 bis K400 liegt

```
SELECT Kunden_Nr, Name, Ort

    FROM Kunde
    WHERE Kunden_Nr  NOT BETWEEN 'K100' AND 'K200';
```

SQL Referenz

3.3.3 Das IN-Prädikat

Es wird getestet, ob der vor IN angeführte Ausdruck in einer nach IN angegebenen Ausdrucksliste (Wertemenge) enthalten ist.
Diese Wertemenge kann auch Ergebnis einer Unterabfrage sein.

Beispiel: Ausgabe der Kundendaten der Kunden, die eine Rechnung über 1000 Mark erhalten haben

```
SELECT Kunden_Nr, Name, Vorname, Ort

    FROM Kunde
    WHERE Kunden_Nr IN (SELECT Kunden_Nr

                            FROM Rechnung
                            WHERE Betrag > 1000);
```

3.3.4 Das EXISTS-Prädikat

Es wird bestimmt, ob die dem EXIST-Prädikat folgende Unterabfrage mindestens eine Ergebniszeile liefert.

Beispiel: Auswahl aller Kundendaten, die auch (mindestens) eine Rechnung erhalten haben.

```
SELECT *

    FROM Kunde
    WHERE EXISTS (SELECT Kunden_Nr

                    FROM Rechnung
                    WHERE Rechnung.Kunden_Nr =
                    Kunde.Kunden_Nr;
```

3.3.5 Das LIKE-Prädikat

In dem nach LIKE angegebenen Ausdruck können ***Platzhalter*** verwendet werden. So bedeutet der Unterstrich '_' ***genau ein*** beliebiges Zeichen, das Prozentzeichen '%' steht für eine ***beliebig lange*** Zeichenkette.
Beispiel: Kundendaten der Kunden, die Meierhofer, Mayerhofer oder so ähnlich heißen.

```
SELECT *
    FROM Kunde
    WHERE name LIKE 'M__erhofer';
```

SQL Referenz

3.4 Die GROUP BY-Komponente

Die GROUP BY-Komponente faßt mehrere Zeilen zu einer Zeile zusammen (gruppiert Zeilen)
Der in der GROUP BY-Komponente verwendete Spaltenname muß in der <Spaltenliste> der SELECT-Komponente ebenfalls angegeben worden sein. Zusätzlich zu diesem Spaltennamen dürfen in der Spaltenliste ebenfalls SQL-Funktionen (s.u.) auftreten.

Beispiel: Kundennummer, Zuname und Summe der Rechnungsbeträge je Kunde.

```
SELECT K.Kunden_Nr, Name, SUM(Betrag)
    FROM Kunde K, Rechnung R
    WHERE K.Kunden_Nr = R.Kunden_Nr
    GROUP BY K.Kunden_Nr, Name;
```

3.5 Die ORDER BY-Komponente

Mit der ORDER BY-Komponente wird das Ergebnis einer Abfrage in der gewünschten Reihenfolge bereitgestellt. Die Ergebnisreihen werden nach den angegebenen Spalten sortiert, je nach ASC oder DESC aufsteigend oder absteigend.

Beispiel: Alle Studenten des Matrikel INF94 geordnet nach Alter (der jüngste zuerst)

```
SELECT *
    FROM Studenten
    WHERE Matrikel = 'INF94'
    ORDER BY GebDatum DESC;
```

Werden mehrere Spaltennamen angegeben, so ergibt sich eine hierarchische Sortierung.

Beispiel: Alle Kunden geordnet nach Name und Vorname ausgeben:

```
SELECT *
FROM Kunde
ORDER BY Name, Vorname;
```

4. Eingebaute SQL-Funktionen

4.1 COUNT

Die COUNT-Funktion liefert die Anzahl der ausgewählten Reihen anstelle ihrer Niederschrift.

Beispiel: Anzahl der Kunden in der Kundendatei:

```
SELECT COUNT(*)
        FROM Kunde;
```

4.2 SUM

Die SUM-Funktion liefert die Summe aller Werte in den selektierten Reihen einer ***numerischen*** Spalte.

Beispiel: Summe aller Rechnungsbeträge des Kunden K001.

```
SELECT SUM(Betrag)
        FROM Rechnung
        WHERE Kunden_Nr = 'K001';
```

4.3 MIN, MAX

Die MIN-Funktion liefert den kleinsten Wert aller selektierten Reihen einer Spalte.
Die MAX-Funktion liefert den größten Wert aller selektierten Reihen einer Spalte.
Beispiel: Der kleinste Rechnungsbetrag.

```
SELECT MIN(Betrag)
        FROM Rechnung;
```

4.4 AVG

Die Funktion AVG liefert den Durchschnitt (Gewogenes arithmetisches Mittel) aller selektierten Reihen einer ***numerischen*** Spalte.
Beispiel: Der durchschnittliche Rechnungsbetrag.

```
SELECT AVG(Betrag)
        FROM Rechnung;
```

5. Trigger

5.1 Aufbau eines Trigger

create or replace trigger <trigger-name>	Trigger-Name
before | after	Auslösezeitpunkt
insert or update of <Spalte1>, <Spalte2>, … or delete on <Tabellenname>	Trigger-Ereignis
for each row	Datensatz- oder Befehls-Trigger
when <Bedingung>	Trigger-Bedingung

`DECLARE`
`Variablenname Typ`
`z.B.: dummy INTEGER;`
	Variablen-Deklaration

PL-SQL Programmcode	Trigger-Rumpf mit eigentlichem Trigger-Programm

5.2 PL-SQL Programmcode

Zugriff auf aktuellen Datensatz mit **:old** oder **:new**, z.B **:old.Gehalt**

:old bezeichnet den Datensatz
- vor der Änderung bei update
- vor dem Löschen bei delete

:new bezeichnet den Datensatz
- nach der Änderung bei update
- nach dem Einfügen bei insert

Spezielle Bedingungen:
- if inserting then
- if deleting then
- if updating then
- if updating (‘<Spalte>’) then

Ausgabe:
Raise <Text oder Variable>

Abbruch:
Explizites Setzen von Fehlern durch
raise_application_error(<Fehler-Nr>, <Fehler-Text>);

Trigger in Oracle

Trigger sind spezielle PL/SQL Konstrukte ähnlich den Prozeduren. Während eine Prozedur allerdings explizit durch einen anderen Block über einen Prozeduraufruf gestartet wird, wird ein Trigger automatisch ausgeführt, wenn ein "triggerndes" Ereignis eintritt.

Diese Ereignisse können DML-Statements sein, aber auch DDL-Anweisungen und eine Anzahl von Systemereignissen. Der Trigger kann vor oder nach dem auslösenden Ereignis ausgeführt werden. Ein Trigger kann ein Row-Level oder Statement-Trigger sein. Ersterer feuert (wird ausgeführt) für jede einzelne Zeile, die durch eine DML-Anweisung verändert wird, letzterer feuert nur einmal für das triggernde Kommando.

Hier sehen Sie die Syntax zum Erzeugen eines Triggers (eine vereinfachte Form):

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
    {BEFORE|AFTER} {INSTEAD OF} {INSERT|DELETE|UPDATE {OF col}} ON <table_name>
    [FOR EACH ROW [WHEN (<trigger_condition>)]]
    <trigger_body>
```

Wichtig ist, daß

- BEFORE und AFTER Trigger nur für Tabellen angelegt werden können. INSTEAD OF Trigger funktionieren nur auf Views. Typischerweise werden sie genutzt, um View Updates zu realisieren, die sonst ja nur unter bestimmten Bedingungen möglich sind.

- Ereignisse, die einen Trigger auslösen, mit OR kombiniert werden können. Mehr noch, bei UPDATE Triggern können Trigger auch nur bei der Veränderung bestimmter Spalten ausgelöst werden (das Schlüsselwort UPDATE wird dann gefolgt von OF und der Liste der Spaltenattribute).

Hier sind einige Beispiele:

```
... INSERT ON tab ...
... INSERT OR DELETE OR UPDATE ON tab ...
... UPDATE OF col1, col2 OR INSERT ON tab ...
```

Ist das Schlüsselwort FOR EACH ROW spezifiert, ist der Trigger ein ROW-LEVEL Trigger, sonst ein Statement-Level Trigger.

Bei ROW-LEVEL Triggern kann eine WHEN Bedingung formuliert werden, die in runden Klammern eingeschlossen sein muß. Die Bedingung muß erfüllt sein, damit der Trigger feuert. Diese Bedingung muß eine PL/SQL Bedingung sein, sie darf kein Subselect enthalten. Vorteil der WHEN-Klausel ist, dass der SQL-Interpreter die Bedingung prüft und, – falls diese nicht erfüllt ist -, die Kontrolle gar nicht erst an den PL/SQL-Interpreter übergibt. Folge ist ein Performancegewinn.

Der *trigger_body* ist ein PL/SQL-Block, keine Aneinanderreihung von SQL Statements!
ORACLE kennt einige Beschränkungen, was man in Triggern tun darf. Damit sollen Situationen vermieden werden, in denen ein Trigger Aktionen vornimmt, die wiederum einen weiteren Trigger auslösen und so weiter, so dass bspw. in der Regel keine Endlosschleifen entstehen. Diese Restriktionen sind:

- In einem Row-Level-Trigger darf nicht das gleiche Objekt modifiziert oder gelesen werden, dessen Modifikation den Trigger ausgelöst hat.
- In einem Statement-Trigger darf keine Tabelle modifiziert oder gelesen werden, die mit der Tabelle, auf der der Trigger liegt, durch ein Fremdschlüsselconstraint verbunden ist und der Trigger durch ein CASCADE ausgelöst wurde.

Trigger in Oracle

Um zu zeigen, wie Trigger funktionieren, nutzen wir die folgenden Tabellen:

```
CREATE TABLE T4 (a INTEGER, b CHAR(10));
CREATE TABLE T5 (c CHAR(10), d INTEGER);
```

Wir erzeugen einen Trigger, der einen Datensatz in Tabelle T5 einfügt, wenn ein Datensatz in die Tabelle T4 eingefügt wird und dabei der Wert von Spalte a <= 10 ist.

```
CREATE TRIGGER trig1
    AFTER INSERT ON T4
    FOR EACH ROW
    WHEN (NEW.a <= 10)
    BEGIN
        INSERT INTO T5 VALUES(:NEW.b, :NEW.a);
    END trig1;
/
```

Die speziellen Variablen NEW und OLD werden benutzt, um die Werte des neuen, bzw. alten Datensatzes zu referenzieren. Bitte beachten Sie, dass im Trigger-Body den Schlüsselworten NEW und OLD ein Doppelpunkt vorangestellt werden muß. In der WHEN Bedingung des Triggers steht kein Doppelpunkt! Wir erzeugen den Trigger mit dem CREATE TRIGGER Statement, durch das Erzeugen wird der jedoch nicht auch ausgeführt.

Nur ein auslösendes Ereignis, wie hier das Einfügen eines Datensatzes in T4, würde den Trigger-Body zur Ausführung bringen.

Um Informationen über Trigger zu erhalten, nutzen Sie:

```
select trigger_type, table_name, triggering_event from user_triggers
where trigger_name = '<trigger_name>';
```

Um einen Trigger zu löschen:

```
drop trigger <trigger_name>;
```

Trigger lassen sich auch deaktivieren, ohne sie löschen zu müssen:

```
alter trigger <trigger_name> {disable|enable};
```

Bedenken Sie, dass durch das Löschen der Tabelle alle mit dieser Tabelle verbundenen Trigger mit gelöscht werden.

Der Before- und After-Zeitpunkt

Wozu genau sind nun die Zeitpunkt Before und After gedacht?? Bei ersten Versuchen scheint es völlig unerheblich, ob ein Before- oder After-Trigger genutzt wird. Z.B. findet immer ein Rollback der kompletten Anweisung statt, wenn ein Trigger mit einem Fehler terminiert.
Im Falle eines Row-Triggers ist der Zeitpunkt BEFORE aber auffällig. Denn nur bei einem BEFORE-ROW-Trigger hat dieser die Möglichkeit, die geänderten Werte z.B. im Zuge eines Updates noch zu verändern.

Folgendes Beispiel verhindert, dass das Gehalt eines Mitarbeiters auf mehr als 5000 geändert werden kann. (ACHTUNG, es handelt sich um einen UPDATE-Trigger, im Falle eines Inserts sind also Gehälter größer 5000 zulässig):

```
CREATE TRIGGER bu_mitarbeiter
    BEFORE UPDATE ON mitarbeiter
    FOR EACH ROW
    BEGIN
        IF :new.gehalt > 5000 then
            :new.gehalt = 5000;
        END IF;
    END trig1;
```

Obiges Beispiel ist kein gutes, denn das gleiche hätte man mit einem CHECK-Constraint erreichen können, was weitaus performanter ist, aber man sieht, dass ein Trigger die Werte eines Datensatzes nicht nur untersuchen, sondern auch ändern kann. Wie gesagt, der ROW-Trigger kann dies nur im Falle eines BEFORE-Triggers, also bevor die eigentlich DML-Anweisung ausgeführt wird.

Hat man nun mehrere Trigger, die bei einer DML-Anweisung auf einer Tabelle reagieren, ist die Ausführungsreihenfolge wie folgt:

- BEFORE-Statement-Trigger wird einmal ausgeführt
- Für jeden manipulierten Datensatz wird der BEFORE-ROW-Trigger ausgelöst.
- Datensatz wird durch die DML-Operation geändert
- Für jeden manipulierten Datensatz wird der AFTER-ROW-Trigger ausgelöst.
- AFTER Statement-Trigger wird einmal ausgeführt

Und wozu können nun diese verschiedenen Zeitpunkte verwendet werden? Z.B. um Trigger miteinander kommunizieren zu lassen!
Nehmen wir folgende Problemstellung an: Es darf maximal die Hälfte der Datensätze, die in einer Tabelle stehen, in einem Delete-Statement gelöscht werden.
Wir müssen also die Anzahl Datensätze vor und nach der Delete-Operation vergleichen, um festzustellen, wieviel nun tatsächlich gelöscht worden sind.
Ein Before-Delete-Trigger soll die Anzahl Datensätze zählen und sich den Wert in einer Package merken. Ein After-Delete-Trigger vergleicht den Wert mit der nun vorhandenen Anzahl Datensätze. Wenn mehr als die Hälfte gelöscht wurde, soll ein Fehler ausgelöst werden.

```
create package gedaechtnis is
          anzahl_vor_delete number;
     end;
     /
```

```
create trigger bd_tab before delete on tab
     begin
        select count(*) into gedaechtnis.anzahl_vor_delete from tab;
     end;
     /
```

```
create trigger ad_tab after delete on tab
         anzahl_nach_delete number;
     begin
         select count(*) into anzahl_nach_delete from tab;
         if anzahl_nach_delete < anzahl_vor_delete / 2 then
           raise_application_error(-20001,'Mehr als die Hälfte löschen ist verboten');
         end if;
      end;
      /
```

Ausserdem gibt es in ORACLE nicht nur Trigger, die auf DML-Ereignisse reagieren, sondern auch solche, die auf DDL- oder gar Systemereignisse (SHUTDOWN, LOGON o.ä.) reagieren. Damit werden Trigger schnell sehr mächtig, man macht sich aber das Debuggen u.U. schwer, da man über die Aktivitäten von Triggern schnell den Überblick verliert. Also setzen Sie Trigger wohlüberlegt und sparsam ein.

Physikalische Speicherorganisation von Datenbanken

Inhaltsverzeichnis

Physikalische Speicherorganisation
Ungeordnetes File mit b Blöcken
Geordnetes, sequentielles File
Externes Hashing
Indexstrukturen
B-Bäume und B*-Bäume

Copyright: Martin Hulin

Physikalische Speicherorganisation

Informationen zu diesem Lernmodul

Informationen zu diesem Lernmodul

Dieses Lernmodul ist Teil des Kapitels 5 der Vorlesung Datenbanksysteme. Wie in der Vorlesung angesprochen, sollen Sie die physikalische Speicherorganisation zum Teil selbständig erarbeiten. Dazu benötigen Sie insgesamt ca. 4 Stunden.

Wenn Sie Fragen haben, während Sie das Lernmodul bearbeiten, können Sie diese im Forum stellen oder erst mal schauen, ob ein Kommilitone die gleiche Frage schon gestellt hat. Sie können mich aber auch gern in der nächsten Vorlesungsstunde persönlich fragen.

Und jetzt viel Erfolg und hoffentlich auch ein bisschen Spaß mit dem Lernmodul,
Martin Hulin

Physikalische Speicherorganisation

Situation:
DB-Daten auf Sekundärspeicher, Ausschnitt in Hauptspeicher
Zugriffszeit auf Sekundärspeicher ist 1000 mal langsamer als die Zugriffszeit auf den Hauptspeicher.
Dieser extreme Unterschied wird mit der technologischen Weiterentwicklung immer noch größer.

Folgerung: Spare Zugriffe auf Sekundärspeicher!

1. Maßnahme:
Organisiere Daten in Blöcken
Hole nicht einzelne Daten, sondern ganzen Block in Hauptspeicher
übliche Blockgrößen: 512 Byte - 64 KByte

Die Hoffnung ist, dass benachbarte Datensätze mit großer Wahrscheinlichkeit auch bald benötigt werden und dann schon im Hauptspeicher sind.

2. Maßnahme: Optimierung von File-Organisation und Zugriffsmethoden

Untersuche Kosten für die DB-Operationen Suche, Eintragen, Ändern und Löschen von Daten

In den folgenden Abschnitten werden **Verschiedene Zugriffsmethoden** untersucht.

Ungeordnetes File mit b Blöcken

Suche: lineare Suche, Zugriff im Durchschnitt auf b/2 Blöcke, wenn Suchbegriff vorhanden;
auf alle b Blöcke, wenn Suchbegriff nicht vorhanden.

Ändern: wie Suche

Eintragen: 1 bis 2 Blöcke

Löschen: wie Suche

Geordnetes, sequentielles File

Sortierung nach einem Attribut.

Suche: binäre Suche (Bisektion) auf Blöcken statt auf Datensätzen.

Zugriffe für Suche nach Ordnungsfeld: ld (b)

sonst b/2 Blöcke.

Ändern: wie Suche

Eintragen: wie Suche + b/2 Blöcke umkopieren, wenn Block voll ist.

Löschen: wie Suche + b/2 Blöcke umkopieren, wenn Block leer ist

Externes Hashing

Aus einem Datenfeld wird mittels einer *Hashfunktion* eine Nummer abgebildet, die sog. *Bucket-Nr.* In einem Array gibt es für jede Bucket-Nr. einen Zeiger auf einen Speicherblock auf der Festplatte. Alle Datensätze, die die gleiche Bucket-Nr. bekommen, kommen auch in den gleichen Block. Ist der Block voll müssen *Konfliktlösungsstrategien* eingesetzt werden.

Suche: 2 Blockzugriffe, bei Suche nach Hashfeld

Einfügen: 2 Blockzugriffe, mehr bei Konflikten

Löschen: 2 Blockzugriffe

Ändern: 2 Blockzugriffe, wenn Hashfeld nicht verändert wird, sonst Suche, Löschen und Einfügen

Nachteile:

Zugriff auf Hashfeld in Sortierreihenfolge nur durch Sortieren möglich.

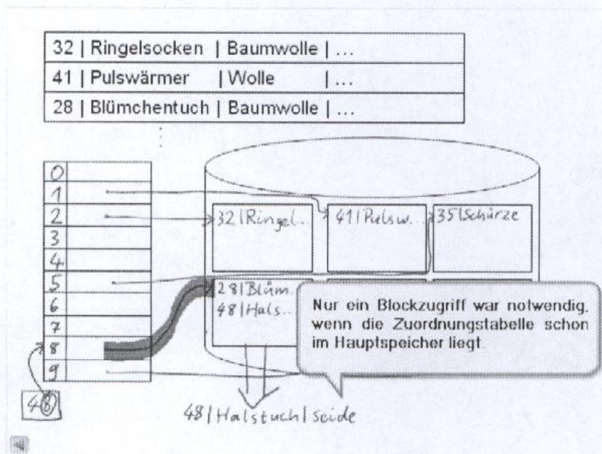
Gesamtfilegröße festgelegt durch Hashfunktion

Alternative: dynamisches Hashing

Schauen Sie sich nun die folgende **Flash-Animation** an, in der externes Hashing an einem Beispiel erläutert wird. Um zum nächsten Schritt der Animation zu kommen, klicken Sie bitte auf das **kleine Dreieck** jeweils rechts unten in den Animationen.

Das kleine Dreieck sieht so aus:

Animation externes Hashing



Indexstrukturen

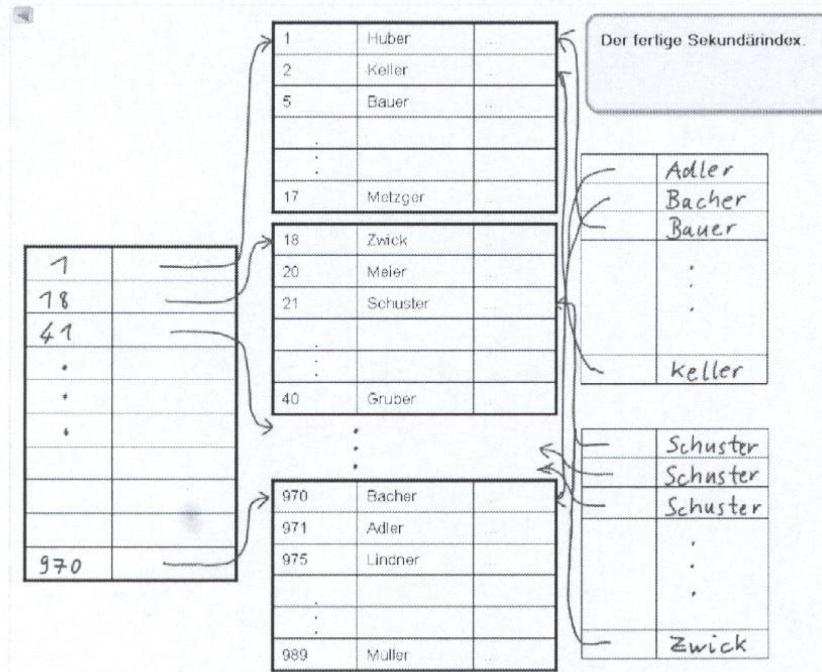
Nachteil aller Sortier- oder Hashing-Strategien: Nur Suche nach Ordnungsfeld wird beschleunigt.

Daher zusätzliche Strukturen, die nach jeweils anderen Attributen sortiert sind: **Indizes** oder **Indexe**. Diese kann man sich vorstellen wie den Indexeintrag in einem Buch: Im Index des Buches findet man den Suchbegriff schnell und folgt dann dem Verweis auf die richtige Seite (eindeutiger Index) oder Seiten (mehrdeutiger Index).

Für das Ordnungsfeld nach dem das File sortiert ist, wird ein **Primärindex** angelegt, für andere Felder jeweils ein **Sekundärindex.**

Auf der nächsten Seite ist ein Beispiel für ein File mit Primärindex PNr und Sekundärindex Nachname zu sehen. Klicken Sie wieder auf das kleine Dreieck (diesmal rechts oben), um zum nächsten Schritt der Animation zu kommen. Je nach Bildschirmauflösung müssen Sie etwas nach unten scrollen, um alle Teile der Animation zu sehen.

Animation Indexstrukturen



Aufwand Suche bei Index

Suche: nochmals reduziert gegenüber sortiertem File, Sortierung auch bei Nicht-Schlüsselfeldern möglich.

Auf der nächsten Seite wird dies für ein **Beispiel** für einen Primärindex durchgerechnet:

Blockgröße 1024 Byte, pro Datensatz 100 Byte, 30 000 Datensätze
Ordnungsfeld 9 Byte, Blockzeiger 6 Byte, also Index-Eintrag 15 Byte

Berechnung Blockzugriffe bei Index

Beispiel für die Berechnung der Blockzugriffe bei der Suche mit Hilfe eines Primärindex

Blockgröße 1024 Byte, Datensatzgröße 100 Byte, 30 000 Datensätze
Ordnungsfeld 9 Byte, Blockzeiger 6 Byte, also Index-Eintrag 15 Byte

Wie viele Datensätze passen auf einen Block? $\left\lfloor \frac{1024}{100} \right\rfloor = 10 \frac{DSe}{Block}$

Wie viele Blöcke braucht man für alle Datensätze? $\left\lceil \frac{30000}{10} \right\rceil = 3000$ Blöcke

Suche **ohne Index** nach Ordnungsfeld mit Bisektion: $\lceil ld\ 3000 \rceil = 12$ Blockzugriffe

Wie viele Indexeinträge passen auf einen Block? $\left\lfloor \frac{1024}{15} \right\rfloor = 68 \frac{Indexeinträge}{Block}$

Wie viele Blöcke braucht der Primärindex?
Man braucht einen Indexeintrag pro Datenblock, also 3000 Indexeinträge: $\left\lceil \frac{3000}{68} \right\rceil = 45$ Blöcke

Suche **mit Index** $\lceil ld\ 45 \rceil = 6$ Blockzugriffe für Index
+ 1 Blockzugriff für Datensatz
= 7 Blockzugriffe

Primärindex reduziert Blockzugriffe von 12 auf 7!

Aufwand Ändern, Einfügen, Löschen bei Index

Einfügen: wie Einfügen bei geordnetem File + Einfügen in Indexfiles

Ändern: Suche + Ändern Datensatz + Ändern Indexeintrag

Löschen: wie Löschen bei geordnetem File + Reorganisieren Indexfiles

B-Bäume und B*-Bäume

Durch Indexstrukturen ist zwar die Suche beschleunigt worden, aber Einfügen und Löschen ist wesentlich langsamer geworden. Besser sind dynamische Indexstrukturen: B-Bäume

Idee der B-Bäume

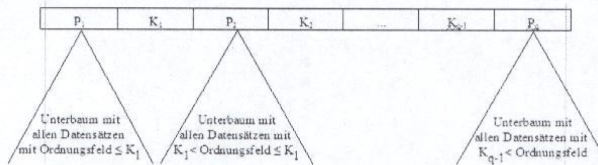
- Bäume sind bekannte Suchstrukturen. Die Suchzeit wächst nur mit dem Logarithmus der Datensätze, wenn der Suchbaum ausgeglichen ist.
- Beim B-Baum sind die Knoten immer ganze Datenblöcke. Dadurch ergibt sich ein hoher Verzweigungsgrad und eine geringe Tiefe.
- Der B-Baum darf nicht entarten, daher gibt es ein Balancekriterium.
- Damit der B-Baum nicht bei jeder Einfüge- oder Lösch-Operation reorganisiert werden muss, erlaubt das Balancekriterium Schwankungen: Jeder Knoten außer der Wurzel muss mindestens p/2 Pointer zu Unterbäumen haben; der Füllgrad kann also zwischen p/2 und p schwanken.

Auf der nächsten Seite sehen Sie die Definition einer Abart des B-Baums – dem B*-Baum – der in den meisten DB-Systemen verwendet wird.

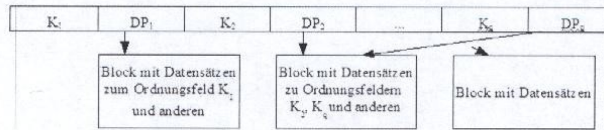
Definition B*-Baum

Definition B*-Bäume der Ordnung p (natürliche Zahl)

- Sei K das Ordnungsfeld. Jeder innere Knoten hat die Form (s. u.) mit $q \leq p$, P sind Pointer



-

- Innerhalb jedes inneren Knotens gilt $K_1 < K_2 < \ldots < K_{q-1}$
- Alle Datensätze im Unterbaum unter P_i liegen zwischen K_{i-1} und K_i
- Jeder innere Knoten hat höchstens p Pointer
- Jeder innere Knoten, außer der Wurzel, hat mindestens p/2 Pointer.
- Die Wurzel hat mindestens 2 Pointer, wenn innere Knoten existieren.
- In den Blattknoten sitzen Pointer auf die Blöcke mit den Datensätzen. Die Größe p kann für die Blattknoten anders gewählt werden als für die inneren Knoten.
- 

- Alle Blattknoten sind auf derselben Ebene.

Beispiel für B*-Baum: Daten

Beispiel für Aufbau von B*-Baum: p = 4

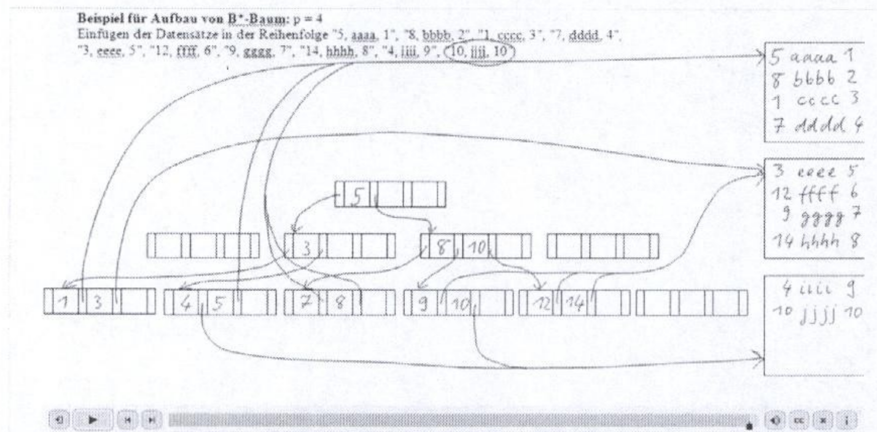
Einfügen der Datensätze in der Reihenfolge

"5, aaaa, 1", "8, bbbb, 2", "1, cccc, 3", "7, dddd, 4", "3, eeee, 5",

"12, ffff, 6", "9, gggg, 7", "14, hhhh, 8", "4, iiii, 9", "10, jjjj, 10"

Schauen Sie sich in der folgenden **Flash-Animation** den Aufbau dieses B*-Baums an.

Aufbau B*-Baum: Animation



Links zu Java-Applets für B-Bäume

Im Internet sind Java-Applets verfügbar, mit denen man mit B-Bäumen spielen kann, und zwar Einfügen und Löschen. Meist sind es keine B*-Bäume sondern B-Bäume, aber das Prinzip ist ähnlich. Probieren Sie es doch einfach mal aus. Wenn Sie im Internet noch bessere Applets zur Animation finden, können Sie einen Link dazu im Forum veröffentlichen.

B-Baum Applet

B+-Baum Applet

Animation: Berechnung Blockzugriffe bei Suche im B-Baum

Allgemeine Formel für Blockzugriffe bei **d** Datenblöcken:

Anzahl **b** von Blättern im B*-Baum

$$b = \left\lfloor \frac{d}{p/2} \right\rfloor = \frac{2d}{p}$$

Damit gilt für die Maximalhöhe **h**:

$$2 \leq \frac{b}{(p/2)^{h-2}} < p$$

$$2\left(\frac{p}{2}\right)^{h-2} \leq b < p\left(\frac{p}{2}\right)^{h-2}$$

$$2\left(\frac{p}{2}\right)^{h-2} \leq b < 2\frac{p}{2}\left(\frac{p}{2}\right)^{h-2}$$

$$\left(\frac{p}{2}\right)^{h-2} \leq \frac{b}{2} < \left(\frac{p}{2}\right)^{h-1}$$

$$(h-2)\ln\frac{p}{2} \leq \ln\frac{b}{2} < (h-1)\ln\frac{p}{2}$$

$$h-2 \leq \frac{\ln b/2}{\ln p/2} < h-1$$

$$\frac{\ln b/2}{\ln p/2} + 1 < h \leq \frac{\ln b/2}{\ln p/2} + 2$$

$$\boxed{\frac{\ln d/p}{\ln p/2} + 1 < h \leq \frac{\ln d/p}{\ln p/2} + 2}$$

ADOBE® CAPTIVATE™

Aufwand Einfügen, Ändern bei B-Baum

Einfügen: Beim Einfügen braucht man pro Ebene maximal 2 Blockzugriffe, wenn alle Knoten überlaufen, einen neuen Wurzelknoten und einen oder zwei Datenblöcke, d. h. bei Höhe *h* insgesamt

Löschen: ähnlich wie Einfügen.

Ändern: Bei Index-Feld wie Suche, Löschen und Einfügen, sonst wie Suche.

Für eine Datentabelle sind gleichzeitig mehrere B*-Baum Indizes für verschiedene Attribute möglich.

Die Technik der **B*-Bäume** ist das **Standardverfahren** in Datenbanken.

Bearbeiten Sie nun die Aufgaben des Selbsttests.