

Datenbanken

Kapitel 6: Anwendungsentwicklung



1

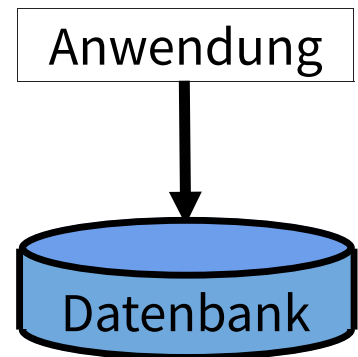
In diesem Kapitel erstellen wir...

- ... Java-Anwendungen, die mit JDBC mit der Datenbank kommunizieren,
- ... Funktionen und Prozeduren direkt in der Datenbank,
- ... Trigger,
- ... Indexe.

2

Anwendungsentwicklung

- Lauffähige Anwendung in Java, C++, Python, PHP, ...
- Konsolenprogramm, GUI, App, Serverprozess, ...
- Komponenten:
 - Connection (Aufbau einer Verbindung zur DB)
 - Statement (Ausführung einer SQL-Anfrage)
 - PreparedStatement (Statement mit Platzhaltern)
 - ResultSet (Ergebnis einer ausgeführten Anfrage)



Im ersten Teil dieses Kapitels betrachten wir Anwendungen, die nicht in der Datenbank laufen, sondern eigenständig sind. Sie stellen eine Verbindung zur Datenbank her, um an diese Anfragen zu schicken und die Ergebnisse dieser Anfragen zu verarbeiten.

3

Beispielanwendungen

GUI / App

```
bash
bash-3.2$ python3 dbtest.py
Produkte des Webshops:
* Schokoriegel (0.89 EUR)
* Müsliriegel (1.19 EUR)
* Spülmaschinentabs (3.99 EUR)
* Katzenfutter (4.99 EUR)
* Maschinenbau-Lehrbuch (22.90 EUR)
* Regal (100.00 EUR)
bash-3.2$
```

A screenshot of a terminal window titled 'bash'. It shows the execution of a Python script 'dbtest.py' which outputs a list of products and their prices. The products are: Schokoriegel (0.89 EUR), Müsliriegel (1.19 EUR), Spülmaschinentabs (3.99 EUR), Katzenfutter (4.99 EUR), Maschinenbau-Lehrbuch (22.90 EUR), and Regal (100.00 EUR). The terminal prompt is 'bash-3.2\$'.



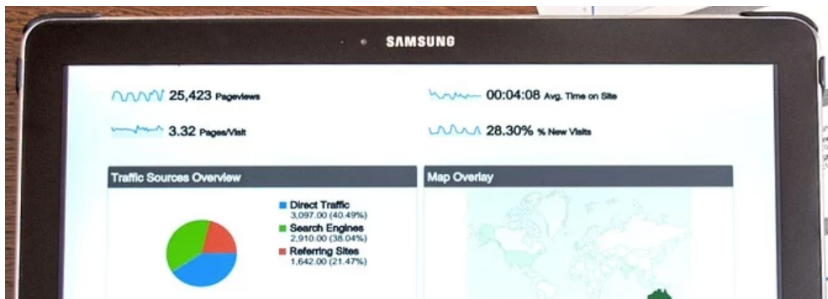
Das Python-Programm links stellt eine Verbindung mit einer Datenbank her, stellt eine SELECT-Anfrage und zeigt die Ergebnisse auf der Konsole an. Die Android-Anwendung rechts ist eine Todo-App (Bildquelle: Antonio Pardo, <https://www.flickr.com/photos/apardo/3323321813>). Diese Anwendung läuft lokal ohne Internetzugriff und speichert die Daten in eine SQLite-Datenbank, die auf dem Handy gespeichert ist.

4

Anwendungsserver

Java EE, JSP, Java Servlets, .NET, PHP, Django, ...

Endanwendung (z. B. Webbrowser) → Anwendungsserver → Datenbank



Das hier gezeigte Tablet stellt keine direkte Verbindung zu einer Datenbank her, um die Daten für die Diagramme zu laden. Ansonsten könnte der Nutzer der App eventuell sogar die DB-Benutzerdaten auslesen. Stattdessen läuft auf einem Webserver eine Anwendung, welche mit der Datenbank interagiert und eine HTML-Seite erzeugt. Diese ruft der Browser des Tablets auf und zeigt sie an. Auch die Todo-App der vorherigen Folie könnte Server-basiert realisiert werden. In dem Fall reicht es, dass der Anwendungsserver die Daten als JSON oder XML bereitstellt, sodass die Smartphone-App diese dann entsprechend darstellen kann.

5

Embedded SQL

Ansatz: SQL-Anfragen und Programmcode mischen,
dann erzeugt ein Precompiler das eigentliche lauffähige Programm.

Beispiel: SQL Object-Language Bindings (OLB) für Java (vormals SQLJ)

```
ProdukteIterator iter;  
#sql iter = { SELECT bezeichnung, preis FROM produkte };  
do {  
    #sql { FETCH :iter INTO :bezeichnung, :preis };  
    System.out.println(bezeichnung+" kostet "+preis+" EUR");  
} while (!iter.endFetch());  
iter.close();
```

Vorteil: Syntaxüberprüfung der Anfrage und Prüfung auf gültige Tabellen- und Spaltennamen kann bereits bei der Compile-Zeit erfolgen.

Die Zeilen, die im Beispielcode mit #sql beginnen, werden vom Precompiler in Java-Code übersetzt. Gleichzeitig erfolgt eine Überprüfung, ob die Anfrage syntaktisch korrekt ist. Neben dem Java-Programm wird auch eine Profil-Datei erzeugt, die in die Datenbank gespeichert werden kann. Die Datenbank würde dabei einen Fehler liefern, wenn z. B. ein ungültiger Spaltenname verwendet wurde.

6

JDBC

SQL-Anfragen werden API-Methoden als Strings übergeben.

```
ResultSet rs = st.executeQuery("SELECT bezeichnung, preis" +  
                                " FROM webshop.produkte");  
while (rs.next()) {  
    bezeichnung = rs.getString(1);  
    preis = rs.getBigDecimal(2);  
    System.out.println(bezeichnung+" kostet "+preis+" EUR");  
}  
rs.close();
```

Vorteile: Flexibel (Anfragen können dynamisch zur Laufzeit generiert werden), kein Precompiler nötig, kompatibel mit allen Java-IDEs und vielen Frameworks.

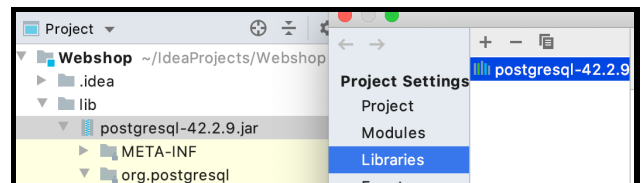
JDBC (Java Database Connectivity) ist eine universelle API-Schnittstelle für die Programmiersprache Java.

7

Driver

Jedes DBMS hat seinen eigenen JDBC-Treiber: MySQL, PostgreSQL, Oracle, ...

⇒ Entsprechende Jar herunterladen und dem Java-Projekt zur Verfügung stellen



Nun kann über die Klasse `DriverManager` eine Verbindung aufgebaut werden.

Bis Java 1.6 war es nötig, den Treiber mittels `Class.forName("org.postgresql.Driver");` zu laden. Mittlerweile unterstützen die meisten JDBC-Treiber den Java Service Provider-Mechanismus, sodass der Treiber automatisch geladen wird, den die JVM im Classpath findet.

8

Connection

```
final String url = "jdbc:postgresql://localhost/meine_db";
final String user = "test";
final String password = "test";
Connection conn = null;
try {
    // Verbindung aufbauen
    conn = DriverManager.getConnection(url, user, password);
    System.out.println("Verbunden mit der PostgreSQL-DB!");
    // Mit der DB interagieren...

    // Am Ende Verbindung wieder schließen
    conn.close();
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```

Eine JDBC-URL besteht aus `jdbc:`, dem Treiber-Namen, der Adresse des DB-Servers, dem Datenbanknamen und evtl. weiteren optionalen Properties. `DriverManager.getConnection` stellt eine Verbindung zur Datenbank her und liefert ein `Connection`-Objekt zurück. Mittels dieses Objekts kann nun mit der Datenbank gearbeitet werden. Tritt ein Fehler beim Verbinden auf (z. B. falsches Passwort), wird eine `SQLException` geworfen. Auch andere Methoden, z. B. diejenigen, die Anfragen an die Datenbank schicken, werfen im Fehlerfall ebendiese Exception.

9

Statement und ResultSet

```
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT bezeichnung, preis +
                                " FROM webshop.produkte");

while (rs.next()) {
    bezeichnung = rs.getString(1);
    preis = rs.getBigDecimal(2);
    System.out.println(bezeichnung+" kostet "+preis+" EUR");
}
rs.close();
st.close();
```

Um eine Anfrage an die Datenbank zu schicken, kann auf der vorhandenen `Connection` ein `Statement`-Objekt erzeugt werden. Eine Anfrage wird mit der `executeQuery`- oder `executeUpdate`-Methode an die Datenbank geschickt. Erstere liefert ein `ResultSet` zurück, zweite dient zur Ausführung von `INSERT`, `UPDATE`, `DELETE`, usw. und liefert die Anzahl der betroffenen Zeilen zurück.

Über ein `ResultSet` kann mittels der `next()`-Methode iteriert werden. `next()` liefert `false`, wenn wir am Ende der Ergebnismenge angekommen sind. Wie im Beispiel gezeigt, kann man mit einer `While`-Schleife Zeile für Zeile über das Ergebnis einer ausgeführten Anfrage iterieren. In jeder Iteration stehen einem Datentyp-spezifische Methoden bereit, um auf die Spaltenwerte der aktuellen Zeile zuzugreifen; entweder über die Position (beginnend mit 1) oder über das Spaltenlabel: `rs.getString(1)` und `rs.getString("bezeichnung")` liefert beides den Wert der Ergebnisspalte "bezeichnung".

10

SQL-Injections



Vorsicht beim Erzeugen von Query-Strings,
die Benutzereingaben beinhalten! (→ xkcd.com/327)

```
ResultSet rs = st.executeQuery("SELECT bezeichnung, preis FROM" +
    " webshop.produkte WHERE hersteller = '"+scan.nextLine()+"'");
```

Geben Sie einen Hersteller ein:

Mit SQL-Injections lassen sich Sicherheitslücken ausnutzen, die es dem Benutzer ermöglichen, Datenbankbefehle einzuschleusen. Das hier gezeigte Programm ist gut gemeint, aber unsicher. Wenn der Benutzer Hersteller Calgonte eingibt, soll `SELECT ... WHERE hersteller = 'Calgonte'` ausgeführt werden. Dadurch, dass der Benutzerinput ohne Überprüfung und Maskierung von Sonderzeichen einfach so als Teil der Anfrage an den String drangehangen wird, ermöglicht man dem Anwender beliebige Anfragen auf der Datenbank auszuführen. Ersterer Hack würde die Produkte-Tabelle dropen. Dies kann man mit der JDBC-Parameter-Einstellung `allowMultiQueries=false` verhindern. Der zweite Hack würde dennoch funktionieren. Er ermöglicht dem Angreifer die Einsicht der E-Mail-Adressen und Passwörter aller Kunden. Durch die Eingabe von `' OR 1=1 --` findet man alle Produkte.

11

SQL-Injections



Anderes Beispiel: Login-Formular

E-Mail:

Passwort:

Einloggen

```
ResultSet rs = st.executeQuery("SELECT true FROM" +
    " webshop.kunden WHERE email = '"+email+"'");
```

Dadurch dass der Hacker hier mit dem Anführungszeichen den String abschließt und mittels `--` den Rest der Anfrage auskommentiert, ergibt sich eine gültige Anfrage, welche lediglich überprüft, ob es den Benutzer gibt und nicht, ob das Passwort stimmt. Im Endeffekt wird ausgeführt: `SELECT COUNT(*) FROM webshop.kunden WHERE email = 'peter@example.com' -- AND password = md5('')`.

Alles ab `--` wird ignoriert.

Die md5-Funktion erzeugt einen Hash aus dem Passwort (z. B. `d41d8cd98f00b204e9800998ecf8427e`), welcher mit dem gespeicherten Hash verglichen wird. Dies verhindert das Abspeichern von Klartext-Passwörtern. Da md5 jedoch unsicher ist, sollte stattdessen eher sha256 verwendet werden.

12

PreparedStatement

Ein PreparedStatement beinhaltet ?-Platzhalter, die vor der Ausführung der Anfrage mit ihren Werten belegt werden.

```
PreparedStatement st = conn.prepareStatement(
    "SELECT bezeichnung, preis FROM webshop.produkte " +
    "WHERE hersteller = ?");
System.out.print("Gib einen Hersteller ein: ");
st.setString(1, scan.nextLine());
ResultSet rs = st.executeQuery();
```

Vorteile: Verhindert SQL SQL-Injections, weniger fehleranfällig, Wiederverwenden von Anfragen. Sonderzeichen (z. B. ') werden automatisch maskiert.

Bevor ein PreparedStatement mittels executeQuery oder executeUpdate ausgeführt wird, muss jeder ?-Platzhalter mit einem Wert belegt werden. Dazu werden Datentyp-spezifische Methoden eingesetzt, z. B. hier setString. Dadurch, dass in diesem gesetzten String Sonderzeichen wie ' automatisch maskiert werden (z. B. zu ' '), sind keine SQL Injections mehr möglich. Ein und dasselbe PreparedStatement kann mehrfach wiederverwendet werden, was auch eine erhöhte Performance zur Folge haben kann. Ein weiterer Vorteil ist, dass man sich unschöne und fehleranfällige Anfrageelemente wie WHERE hersteller = '+' + scan.nextLine() + "'" erspart. Hier vergisst man gerne mal ein '.

13

executeUpdate

```
Scanner scan = new Scanner(System.in);
PreparedStatement st = conn.prepareStatement(
    "INSERT INTO webshop.kunden (name, email, passwort) " +
    "VALUES (?, ?, md5(?))");
System.out.print("Name: ");
st.setString(1, scan.nextLine());
System.out.print("E-Mail: ");
st.setString(2, scan.nextLine());
System.out.print("Passwort: ");
st.setString(3, scan.nextLine());
int zeilenEingefuegt = st.executeUpdate();
```

Anders als executeQuery liefert executeUpdate kein ResultSet zurück, sondern einen Integer: die Anzahl der betroffenen Zeilen. Beim hier gezeigten INSERT ist dies stets eine 1, da nur eine Zeile eingefügt wird. Im Fehlerfall wird eine Exception geworfen. Bei einem UPDATE- oder DELETE-Kommando kann man am Rückgabewert der Methode sehen, wie viele Zeilen geändert oder gelöscht wurden. Neben executeUpdate und executeQuery gibt es auch noch die Methode execute. Diese liefert keinen Rückgabewert. Man verwendet sie z. B. für CREATE TABLE oder andere DDL-Kommandos.

14

DB-Metadaten-Zugriff mit JDBC

DatabaseMetaData

```
DatabaseMetaData md = conn.getMetaData();
//          cata. schema    tabelle typ
ResultSet rs = md.getTables(null, "webshop", "%", null);
while (rs.next()) {
    System.out.println(rs.getString(3));
}
```

ResultSetMetaData

```
ResultSetMetaData meta = rs.getMetaData();
int numberColumns = meta.getColumnCount(); int i;
for (i = 1; i <= numberColumns; i++) {
    String cName = meta.getColumnName(i);
    String dType = meta.getColumnTypeName(i);
    System.out.println("Spalte: "+cName+", Datentyp: "+dType);
}
```

JDBC stellt Methoden bereit, um auf die Metadaten der Datenbank (z. B. Tabellen, Spalten einer Tabelle, ...) und auf Metadaten eines ResultSets zuzugreifen.

15

Python-Anwendung

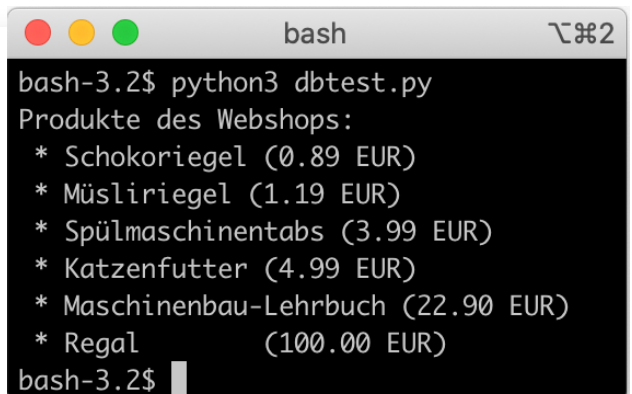
```
#!/usr/bin/python
import psycopg2

conn = psycopg2.connect(
    database="u1", user="u1",
    password="u1", host="localhost",
    port="5432",
)

cursor = conn.cursor()

print("Produkte des Webshops:")

cursor.execute("SELECT bezeichnung, preis FROM webshop.produkte")
while True:
    row = cursor.fetchone()
    if not row:
        break
    print("* %s (%.2f EUR)" % (row[0], row[1]))
```



```
bash
bash-3.2$ python3 dbtest.py
Produkte des Webshops:
* Schokoriegel (0.89 EUR)
* Müsliriegel (1.19 EUR)
* Spülmaschinentabs (3.99 EUR)
* Katzenfutter (4.99 EUR)
* Maschinenbau-Lehrbuch (22.90 EUR)
* Regal (100.00 EUR)
bash-3.2$
```

Das hier gezeigte Python-Programm verwendet die PostgreSQL-Treiber-Library psycopg2. Ähnlich wie in JDBC wird zunächst eine Verbindung mit der Datenbank aufgebaut, dann ein SELECT-Statement ausgeführt und über das Ergebnis der Anfrage Zeile für Zeile iteriert.

16

Python-Anwendung

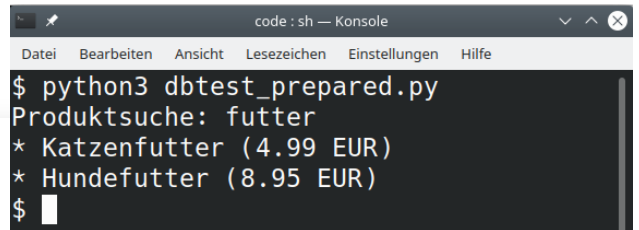
Parametrisierte Anfragen

```
cursor = conn.cursor()

suche = input("Produktsuche: ")

cursor.execute("SELECT bezeichnung, preis FROM webshop.produkte " +
               "WHERE bezeichnung LIKE '%%' || %(suche)s || '%%'",
               {"suche":suche})

while True:
    row = cursor.fetchone()
    if not row:
        break
    print("* %s (%.2f EUR)" % (row[0], row[1]))
```



psycopg2 unterstützt zwar nicht auf direktem Wege Prepared Statements, aber es unterstützt parametrisierte Anfragen. In der hier gezeigten Anfrage wurde ein Parameter %(suche) vom Typ String (s) eingeführt. Der Wert wird in einem Dictionary als zweiten Parameter der execute-Methode übergeben. Python kümmert sich dann um das escapen von ' und anderen Zeichen, um SQL-Injections zu verhindern. Da wir in der gezeigten Anfrage das Prozent-Symbol brauchen, müssen wir es escapen, indem wir %% schreiben.

17

🔗 102

Routinen

Benutzerdefinierte Routinen sind in der DB gespeicherte Datenbankobjekte (wie Tabellen und Views):

- **Prozeduren** tun etwas
- **Funktionen (UDF)** liefern einen Ergebniswert
- **Tabellenfunktionen** liefern eine Tabelle
- **Methoden** gehören zu einem User-defined Datatype

Routinen haben einen Namen und Eingabeparameter.

Die meisten Datenbankmanagementsysteme erlauben die Erstellung von Routinen in einer Programmiersprache. In PostgreSQL heißt diese Sprache PL/pgSQL (Prozedurale Spracherweiterung für PostgreSQL). Neben SQL-Anfragen (SELECT, INSERT, ...) gibt es auch IF-Blöcke, FOR-Schleifen, usw. Methoden werden in dieser Vorlesung nicht behandelt. Durch in der Datenbank gespeicherte Routinen werden unnötige Anfragen zwischen Client-Anwendung und DB-Server vermieden. Vor allem solche Anfragen, die nur Zwischenergebnisse abrufen, welche in Folgeanfragen verwendet werden.

18

PL/pgSQL

Block:

```
[ DECLARE
-- hier können Variablen definiert werden ]
BEGIN
-- Anweisungen ...
END
```

Kontrollstrukturen:

```
IF ... THEN ... [ELSE ...] END IF;
WHILE ... LOOP ... END LOOP;
LOOP ... EXIT WHEN ...; END LOOP;
FOR ... IN ... .. LOOP ... END LOOP;
FOR ... IN 'SELECT ...' LOOP ... END LOOP;
```

19

 103

Stored Procedures

```
CREATE OR REPLACE FUNCTION alles_leeren() RETURNS void AS
$$ BEGIN
  TRUNCATE bewertungen CASCADE;
  TRUNCATE produkte CASCADE;
  TRUNCATE hersteller CASCADE;
  TRUNCATE kunden CASCADE;
END $$ LANGUAGE plpgsql;
```

```
SELECT alles_leeren();
```

In diesem einfachen Beispiel hat unsere Prozedur `alles_leeren` keine Eingabeparameter (daher die leeren Klammern `()`), wir definieren keine Variablen und verwenden auch keine Kontrollstrukturen oder Schleifen. Es werden lediglich vier `TRUNCATE`-Kommandos ausgeführt. In PostgreSQL sind Prozeduren `FUNCTION`-Objekte mit dem Rückgabebetyp `void`. Man ruft sie in einem `SELECT`-Kommando ohne `FROM`-Klausel auf. Seit PostgreSQL 11 gibt es jedoch auch `CREATE PROCEDURE`, welche dann mit einem `CALL`-Befehl aufgerufen werden (siehe übernächste Folie). In anderen DBMS werden Prozeduren mit einem `EXECUTE`-Befehl ausgeführt.

Schreibt man `CREATE OR REPLACE` statt einfach nur `CREATE`, wird die Funktion überschrieben, falls sie schon existiert.

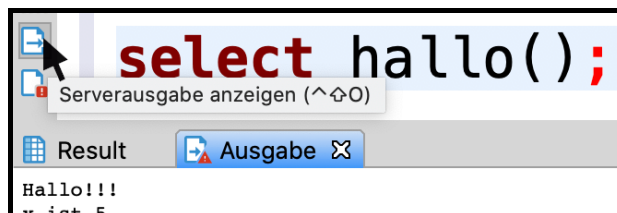
20

RAISE NOTICE

RAISE NOTICE gibt einen Infotext auf der Konsole aus.

```
CREATE OR REPLACE FUNCTION hallo() RETURNS void AS $$
DECLARE
    x INT := 5;
BEGIN
    RAISE NOTICE 'Hallo!!!';
    RAISE NOTICE 'x ist %', x;
END $$ LANGUAGE plpgsql;
```

```
SELECT hallo();
```



Verwendet man im Notice %-Platzhalter, kann man deren Werte als zusätzliche Parameter übermitteln. Diese Folie zeigt auch, wie man Variablen deklariert. Bei `x INT := 5;` hat x den Wert 5. Mit `x INT` würde man lediglich die Variable x ohne Wert definieren, damit man diesen später zuweisen kann.

21

Stored Procedures

```
CREATE PROCEDURE bestelle_produkt(_kundenr INT,
                                   _produktnr INT) AS $$
DECLARE
    _bestellnr INT;
BEGIN
    SELECT MAX(bestellnummer)+1 INTO _bestellnr FROM bestellungen;
    RAISE NOTICE 'Bestellnummer: %', _bestellnr;
    INSERT INTO bestellungen(bestellnummer, kundennummer, zeit,
                             preis) VALUES (_bestellnr, _kundenr, current_timestamp,
                             (SELECT preis FROM produkte WHERE produktnr=_produktnr));
    INSERT INTO bestellungen_positionen(bestellnummer,
                                         produktnummer, anzahl) VALUES (_bestellnr, _produktnr, 1);
    COMMIT;
END $$ LANGUAGE plpgsql;
```

```
CALL bestelle_produkt(5, 17);
```

Anders als bei CREATE FUNCTION werden bei PostgreSQL im CREATE PROCEDURE auch Transaktionen (COMMIT / ROLLBACK) unterstützt. Damit wird garantiert, dass beide hier dargestellten INSERTs ausgeführt werden (oder keines) und dass die mit MAX ermittelte neue Bestellnummer auch in jedem Fall gültig ist, auch wenn andere Transaktionen parallel Bestellungen einfügen (→ Phantomproblem).

22

Eingabeparameter

Zugriff über den Namen oder die Position (\$1, ...)

```
CREATE OR REPLACE PROCEDURE test(i int, s VARCHAR) AS $$
BEGIN
  RAISE NOTICE 'i ist %', i;      -- i ist 99
  RAISE NOTICE 's ist %', $2;    -- s ist Tomate
END $$ LANGUAGE plpgsql;

CALL test(99, 'Tomate');
```

\$2 steht hier für den zweiten Parameter, also für s. Wir verwenden in den Beispielen aber auch oft Variablennamen, die mit einem Unterstrich beginnen (z. B. `_produktnr`), damit man sie nicht mit Spaltennamen oder ähnlichem verwechselt.

23

 103

Funktionen

UDF = User-defined Function

```
CREATE OR REPLACE FUNCTION addieren(int, int) RETURNS INT AS
$$ BEGIN RETURN $1+$2; END $$ LANGUAGE plpgsql;

SELECT addieren(5,4);

SELECT *, addieren(sterne, 1) FROM bewertungen;
```

Funktionen haben beliebig viele Eingabeparameter und liefern genau einen Rückgabewert zurück. Der Datentyp des Rückgabewerts steht oben hinter RETURNS. Mit einem RETURN-Kommando terminiert die Funktion und erzeugt die Rückgabe. Man kann die Funktion genau wie eingebaute SQL-Funktionen (UPPER, MD5, ...) in SELECT-Anfragen mit und ohne FROM-Klausel verwenden, aber auch in INSERT, UPDATE, etc. Die hier dargestellte Funktion addiert zwei Zahlen. Die Eingabeparameternamen kann man wie hier gezeigt einfach weggelassen, wenn man auf sie lediglich mittels \$1 etc. zugreift.

24

Exceptions werfen und behandeln

```
CREATE FUNCTION dividieren(float, float) RETURNS float AS
$$ BEGIN
RETURN $1/$2;
EXCEPTION WHEN OTHERS THEN
    RAISE EXCEPTION 'Divisor darf nicht 0 sein!';
END $$ LANGUAGE plpgsql;
```

Hinter EXCEPTION WHEN steht der Name einer Exception, die abgefangen werden soll. OTHERS fängt alle sonstigen Fehler ab. Mit RAISE EXCEPTION wird ein Programm unterbrochen und die entsprechende Fehlermeldung an den Client zurückgegeben.

25

Volatilitäts-Kategorien

```
CREATE OR REPLACE FUNCTION addieren(int, int) RETURNS INT AS
$$ BEGIN RETURN $1+$2; END $$ LANGUAGE plpgsql IMMUTABLE;
```

VOLATILE (Standard)

Funktion darf alles: DB verändern und bei gleichen Parameterwerten unterschiedliche Ergebnisse liefern.

STABLE

Funktion darf die DB nicht verändern und muss innerhalb des gleichen Statements bei gleichen Parameterwerten das gleiche Ergebnis liefern.

IMMUTABLE

Funktion darf die DB nicht verändern und muss bei gleichen Parameterwerten das gleiche Ergebnis liefern.

Wenn eine Funktion als STABLE oder IMMUTABLE deklariert wird, kann der Datenbank-Optimierer Ergebnisse der Funktion wiederverwenden, ohne die Funktion nochmals aufzurufen. Eine Funktion `get_hersteller_land`, die beispielsweise eine SELECT-Anfrage an eine Tabelle schickt, um das Land eines Herstellers zu ermitteln, ist STABLE, da sie innerhalb einer Anfrage, in der sie aufgerufen wird, für den gleichen Hersteller das gleiche Land liefern würde.

26

Tabellenfunktionen

```
CREATE OR REPLACE FUNCTION produkte_von(VARCHAR) RETURNS TABLE
(produktnr INT, bezeichnung VARCHAR(100),
 preis DECIMAL(9,2), hersteller VARCHAR(50)) AS $$
SELECT produktnr, bezeichnung, preis, hersteller
FROM produkte WHERE hersteller = $1
$$ LANGUAGE sql;
```

```
SELECT * FROM produkte_von('Calgonte');
```

Tabellenfunktionen liefert eine Tabelle zurück mit beliebig vielen Zeilen. Die Spalten und deren Typen werden im CREATE FUNCTION-Kommando im RETURNS TABLE-Teil definiert. Genau wie eine View wird eine Tabellenfunktion in der FROM-Klausel einer SELECT-Anfrage aufgerufen. Eine Tabellenfunktion ist quasi eine View mit Parametern.

27

 104

Trigger

Wenn ...

- BEFORE
- AFTER
- INSTEAD OF
- INSERT
- UPDATE
- DELETE
- ON <tabelle>
- ON <view>

dann ...

Mittels Triggern können Aktionen ausgeführt werden, jedesmal wenn das festgelegte Ereignis eintritt. Nach einem INSERT in die Produkttabelle kann automatisch der Hersteller eingefügt werden, falls es ihn noch nicht gibt. Bevor der Preis eines Produkts geändert wird, wird überprüft, ob diese Änderung auch in Ordnung ist. Wenn der Preis um mehr als 20% erhöht wird, soll ein Fehler kommen. Wenn ein INSERT auf einer View gemacht wird, sollen stattdessen INSERTs in gewisse Tabellen gemacht werden. Solche Abläufe können mittels Trigger definiert werden.

28

Trigger in PostgreSQL

Triggerfunktion

- Definiert die auszuführende Aktion
- **CREATE FUNCTION ... RETURNS TRIGGER ...**

Trigger

- Definiert den eigentlichen Trigger (die Wenn-Falls-Dann-Bedingung)
- **CREATE TRIGGER ... {BEFORE | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE} ON ... FOR EACH {ROW | STATEMENT} [WHEN (...)] EXECUTE PROCEDURE ...();**

In PostgreSQL wird zunächst eine Triggerfunktion definiert. Diese ist ähnlich zu einer Funktion mit dem Returns-Type void. In dieser Triggerfunktion hat man jedoch Zugriff auf die eingefügte, geänderte, bzw. gelöschte Zeile und man kann diese sogar noch modifizieren oder die Operation noch aufhalten. Trigger können auch sogenannte ECA-Regel implementieren. ECA steht für Event, Condition, Action. Wenn also ein Ereignis eintritt, soll - falls die Bedingung erfüllt ist - das Ereignis ausgeführt werden. Die Bedingung lässt sich in PostgreSQL mittels einer optionalen WHEN-Bedingung festlegen.

29

AFTER INSERT

Nachdem das neue Tupel eingefügt wurde, soll noch etwas gemacht werden.

```
CREATE FUNCTION produkte_trigger() RETURNS TRIGGER AS
$$ BEGIN
IF (NEW.hersteller IS NOT NULL AND NOT EXISTS
  (SELECT * FROM hersteller WHERE firma = NEW.hersteller)) THEN
  INSERT INTO hersteller (firma) VALUES(NEW.hersteller);
END IF;
RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

NEW bietet Zugriff auf die neu eingefügte Zeile.

```
CREATE TRIGGER produkte_trigger AFTER INSERT ON produkte
FOR EACH ROW EXECUTE PROCEDURE produkte_trigger();
```

Immer wenn ein INSERT auf die Produkttabelle gemacht wird, feuert der Trigger. Er sorgt dafür, dass der Hersteller in die Herstellertabelle eingefügt, wenn er noch nicht drin ist. Über das Objekt NEW hat man innerhalb des Triggers Zugriff auf die Attributwerte der neu eingefügten Zeile.

30

AFTER INSERT

```
INSERT INTO produkte VALUES(1000, 'Kuchen', 2.00, 'Kuchenpeter');
```

[23503]: ERROR: insert or update on table "produkte" violates foreign key constraint "produkte_hersteller_fkey" Detail: Key (hersteller)=(Kuchenpeter) is not present in table "hersteller".

- Entweder Trigger auf BEFORE INSERT ändern
- oder das Fremdschlüssel-Constraint auf DEFERRED setzen:

```
ALTER TABLE webshop.produkte ADD CONSTRAINT
produkte_hersteller_fkey FOREIGN KEY (hersteller)
REFERENCES hersteller(firma) ON UPDATE CASCADE
INITIALLY DEFERRED;
```

Ändern wir den Trigger auf BEFORE INSERT, wird die Prozedur, die den Hersteller einfügt, ausgeführt, bevor die neue Zeile in die Produkttabelle eingefügt wird. Der Hersteller ist hier der Parent-Record und das Produkt der Child-Record. Der Parent muss vorher existieren, bevor der Child-Record existiert. Als Alternative kann die Überprüfung des Fremdschlüssel-Constraints auch auf DEFERRED (verzögert) setzen. Der Standard ist IMMEDIATE - eine sofortige Überprüfung. Bei DEFERRED wird das Constraint erst zum Ende der Transaktion überprüft. In unserem Fall führt dies zum Erfolg, da zum Ende der Transaktion der zuvor nicht existierende Hersteller zwischenzeitlich durch den Trigger eingefügt wurde.

31

BEFORE INSERT / UPDATE / DELETE 105

Mit einem BEFORE-Trigger kann man das einzufügende Tupel modifizieren oder das INSERT / UPDATE / DELETE ablehnen.

```
CREATE OR REPLACE FUNCTION preis_trigger() RETURNS TRIGGER AS $$
DECLARE anz_gratis_produkte INT;
BEGIN
IF (NEW.preis < 0) THEN NEW.preis = 0; END IF;
SELECT COUNT(*) INTO anz_gratis_produkte
FROM produkte WHERE preis = 0;
IF (NEW.preis = 0 AND anz_gratis_produkte >= 3) THEN
RAISE EXCEPTION 'Zu viele kostenlose Produkte!';
END IF;
RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER preis_trigger BEFORE INSERT OR UPDATE ON produkte
FOR EACH ROW EXECUTE PROCEDURE preis_trigger();
```

Bei einem BEFORE INSERT-Trigger kann man einzufügende Tupel noch abändern, bevor es tatsächlich in die Tabelle eingefügt wird. Hier wird der Preis auf 0 gesetzt, sollte jemand versuchen, ein Produkt mit negativen Preis einzufügen. Außerdem überwacht der Trigger, dass es nie mehr als drei kostenlose Produkte gibt. Ein INSERT oder UPDATE wird abgelehnt, wenn es dieses Constraint verletzen sollte.

32

OLD / NEW

- INSERT-Trigger: Zugriff auf NEW
- DELETE-Trigger: Zugriff auf OLD
- UPDATE-Trigger: Zugriff auf OLD und NEW

```
CREATE OR REPLACE FUNCTION preiserhoehung_trigger()  
RETURNS TRIGGER AS $$  
BEGIN  
IF (NEW.preis > OLD.preis*1.1) THEN  
  RAISE EXCEPTION 'Preiserhöhung um mehr als 10%% nicht erlaubt!';  
END IF;  
RETURN NEW;  
END; $$ LANGUAGE plpgsql;
```

Beim INSERT hat man in der Triggerfunktion Zugriff auf NEW (die eingefügte Zeile), beim DELETE auf OLD (die gelöschte Zeile) und beim UPDATE auf OLD und NEW. OLD bietet Zugriff auf den vorherigen Zustand des Tupels vor der Änderung, NEW auf den Zustand nach der Änderung.

Hier wurde 10%% geschrieben, da % als Platzhalter steht, der durch einen Wert ersetzt wird (siehe Folie zu RAISE NOTICE). %% wird durch % ersetzt.

33

FOR EACH ROW / STATEMENT

FOR EACH ROW

- Triggerfunktion wird für jedes eingefügte / gelöschte / geänderte Tupel einmal aufgerufen
- Zugriff auf das Tupel mittels NEW bzw. OLD

FOR EACH STATEMENT

- Triggerfunktion wird für das gesamte INSERT / UPDATE / DELETE-Statement nur einmal aufgerufen
- Kein direkter Zugriff auf die Tupel

Der Trigger, der überprüft, dass es stets nie mehr drei kostenlose Produkte gibt, könnte auch vom Typ FOR EACH STATEMENT sein. Dies würde die Überprüfung bei großen Änderungsmengen deutlich beschleunigen, da sie nur einmal und nicht für jede geänderte Zeile erfolgen muss.

34

INSTEAD OF-Trigger

```
INSERT INTO meine_view VALUES (...)
```

- Nur in simple Views Projektion-Selektion-Views darf ein INSERT/UPDATE/DELETE erfolgen
- Mittels einer CHECK OPTION kann überprüft werden, dass das Tupel auch das WHERE-Prädikat der View erfüllt.
- INSTEAD OF-Trigger ermöglichen INSERT/UPDATE/DELETE auf jeder View.

```
CREATE TRIGGER meine_view_trigger INSTEAD OF INSERT ON meine_view  
FOR EACH ROW EXECUTE PROCEDURE meine_view_trigger();
```

INSTEAD OF-Trigger sind nur für Sichten gedacht. Jedes mal, wenn jemand ein INSERT / UPDATE / DELETE auf eine Sicht macht, soll stattdessen die Trigger-Aktion ausgelöst werden.

35

INSTEAD OF-Trigger

```
CREATE VIEW meine_view AS  
SELECT p.produktnr, p.bezeichnung, p.preis, p.hersteller, h.land  
FROM produkte p JOIN hersteller h ON p.hersteller=h.firma;
```

```
CREATE OR REPLACE FUNCTION meine_view_trigger()  
RETURNS TRIGGER AS $$  
BEGIN  
INSERT INTO hersteller (firma, land)  
  VALUES (NEW.hersteller, NEW.land)  
  ON CONFLICT (firma) DO UPDATE SET land=EXCLUDED.land;  
INSERT INTO produkte (produktnr, bezeichnung, preis, hersteller)  
  VALUES (NEW.produktnr, NEW.bezeichnung, NEW.preis, NEW.hersteller);  
RETURN NEW;  
END; $$ LANGUAGE plpgsql;
```

Die gezeigte View führt eine Denormalisierung der beiden Tabellen Produkte und Hersteller aus, also einen Join. Der Trigger sorgt dafür, dass ein INSERT in diese View aufgeteilt wird in zwei INSERT-Kommandos. Zunächst wird der Hersteller eingefügt. Falls es ihn schon gibt, wird sein Land angepasst. Im Anschluss erfolgt das Einfügen des Produktes.

36

Indexe

Motivation: Wir haben n Produkte. Wie teuer ist die folgende Anfrage?

```
SELECT * FROM produkte WHERE produktnr = 29
```

- Full Table Scan: $O(n)$
- Tabelle intern sortiert nach Produktnummer: $O(\log_2(n))$
- B-Baum-Index auf der Produktnummer: $O(\log_k(n))$
- Hash-Index auf der Produktnummer: $O(1)$

Mit der O-Notation wird der Aufwand in Abhängigkeit der Tabellengröße beschrieben.

37

Full Table Scan

Linearer Aufwand: $O(n)$

Jede einzelne Zeile wird gelesen und für sie wird überprüft, ob das WHERE-Prädikat wahr oder falsch ist.

```
SELECT * FROM produkte WHERE produktnr = 29
```

produktnr	bezeichnung	preis	hersteller
91	Maschinenbau-Lehrbuch	22.9	-
92	Regal	100	-
29	Spülmaschinentabs	3.99	Calgonte
17	Schokoriegel	0.89	Monsterfood
18	Müsliriegel	1.19	Monsterfood

Bei einem Full Table Scan muss jede einzelne Zeile der Tabelle gelesen werden, um die gewünschte Zeile zu finden. Wäre die Tabelle doppelt so groß, wäre auch der Aufwand doppelt so groß.

38

Clustered Index

Logarithmischer Aufwand: $O(\log_2(n))$

Binäre Suche: In die Mitte der Tabelle springen. Hat diese Zeile eine größere Produktnummer als die gesuchte? Oberhalb weitersuchen, usw.

```
CLUSTER produkte USING produkte_pkey
```

```
SELECT * FROM produkte WHERE produktnr = 29
```

produktnr
17
18
19
22
29
88
91
92
998
999

- Es kann nur einen Clustered Index pro Tabelle geben
- Re-Clustering nötig nach INSERT, UPDATE, DELETE

Ein Clustered Index bestimmt die interne Speicherung der Zeilen. Clustert man die Produkttabelle nach der Primärschlüsselspalte produktnr, werden die Datensätze sortiert nach der Produktnummer abgespeichert. In PostgreSQL ist dazu die Ausführung des Kommandos CLUSTER nötig. PostgreSQL macht kein automatisches Reclustering, d. h. nach Änderungen an den Datensätzen muss nochmals CLUSTER ausgeführt werden. Liegen die Daten sortiert, ist eine Suche auf der Index-Spalte, ein GROUP BY und ein Sortieren sehr schnell. Das Re-Clustering ist jedoch eine sehr teure Operation.

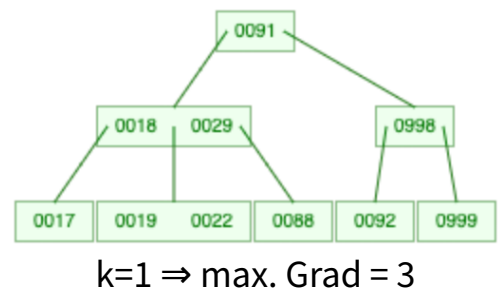
39

B-Bäume

Logarithmischer Aufwand: $O(\log_k(n))$

Simulation: <https://www.cs.usfca.edu/~galles/visualization/BTree.html>

- Jeder Knoten hat maximal den Grad $2k+1$, also max. $2k+1$ Kindknoten
- Jeder Knoten speichert zw. k und $2k$ Einträge
- Suche startet in der Wurzel, dann wird sich durch den Baum navigiert: kleiner: links; größer: rechts entlang



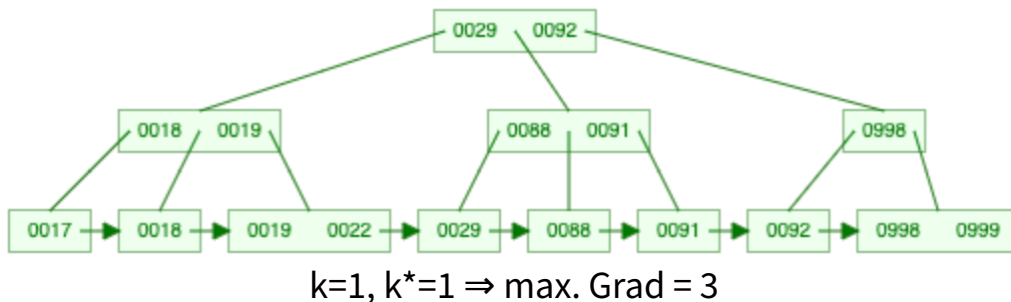
Ein innerer Knoten hat immer einen Kindknoten mehr als Einträge. Ein Eintrag in einem B-Baum kann mit logarithmischem Aufwand gesucht werden. Erst wenn ein Baum k -mal mehr Einträge speichert, erhöhen sich die Suchkosten gerade mal um 1. Im gezeigten Baum können wir jeden Eintrag finden, indem wir höchstens drei Seiten lesen müssen, da die Höhe des Baumes 3 ist. In einem voll besetzten B-Baum der Klasse $k=3$ und der Höhe $h=5$ kann man mehr als $3^5=243$ Einträge abspeichern und jeder davon lässt sich mit höchstens 5 Leseoperationen wiederfinden. In der Realität ist z. B. $k=1024$. Somit lassen sich Milliarden Einträge so organisieren, dass man jeden davon mit nur drei Leseoperationen finden kann.

40

B+-Bäume

Simulation: <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>

- Optimierung des B-Baums; in DMBS häufigst eingesetzte Indexstruktur
- Innere Knoten sind nur Wegweiser (max. $2k$ Schlüssel, max. $2k+1$ Kinder)
- Blattknoten enthalten die Schlüssel und Pointer zu max. $2k^*+1$ Datensätzen
- Blattknoten sind miteinander verkettet \Rightarrow schnelle sequenzielle Suche
- Suche startet in der Wurzel, dann links ($<$) oder rechts (\geq) weitersuchen

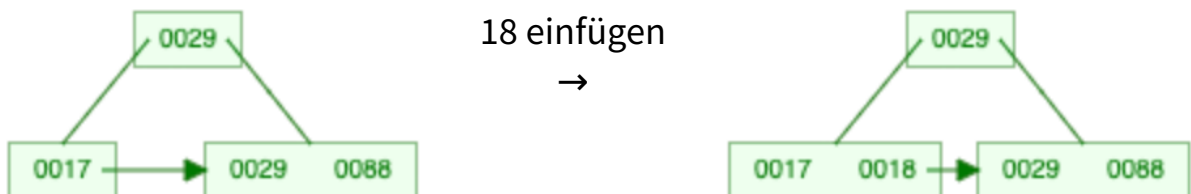


Da man in den Bäumen nicht lediglich Zahlen o.ä. sondern natürlich auch den Pointer zum tatsächlichen Datensatz speichern muss, ist es eine Optimierung, diese Pointer nur in die Blattebene zu speichern, sodass innere Knoten bei gleicher Seitengröße einen viel höheren Grad haben können als Blattknoten.

41

Einfügen in B+-Baum

1. Blattknoten suchen, in welchen der neue Schlüssel gehört
2. a. Noch Platz? \Rightarrow einfach einfügen



Im links gezeigten B+-Baum ($k=1, k^*=1$) soll der Eintrag 18 eingefügt werden. Zuerst wird der Knoten gesucht, in den die 18 gehört. 18 ist kleiner als 29, also gehen wir nach links. Dort ist bisher nur die 17 gespeichert. In jedem Knoten ist Platz für $2k^* = 2$ Einträge, also ist dort noch Platz, sodass wir die 18 einfach einfügen können.

Hat unsere Produkttabelle einen Index auf der Produktnummer-Spalte - die meisten DBMS legen auf Primärschlüsselspalten ohnehin automatisch B+-Baum-Indexte an -, wird in den gezeigten Baum der Wert 18 zusammen mit der Speicheradresse, wo der entsprechende Datensatz auf der Festplatte gespeichert ist, in die Blattebene des B+-Baums eingefügt. Möchte man nun suchen (`SELECT * FROM produkte WHERE produktnr=18`), kann der Index genutzt werden, um schnell den Datensatz zu finden. Auch bei Bereichsanfragen (`WHERE produktnr BETWEEN 18 and 88`) hilft der B+-Baum: Man navigiert sich zur Blattebene zur 18, und folgt dann den verketteten Blättern bis zur 88. Im B-Baum wäre dies komplexer.

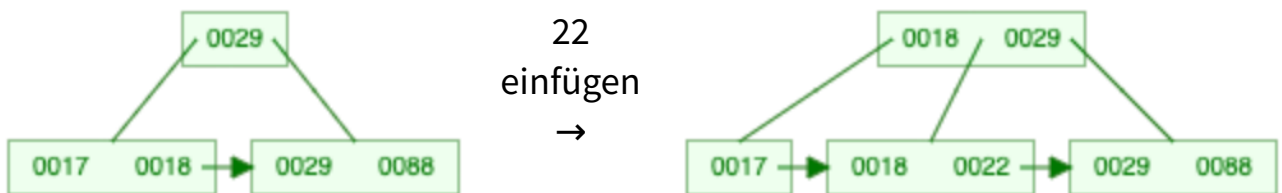
42

Einfügen in B+-Baum

1. Blattknoten suchen, in welchen der neue Schlüssel gehört
2. a. Noch Platz? \Rightarrow einfach einfügen
b. Kein Platz mehr? \Rightarrow einfügen und Knoten splitten

Split

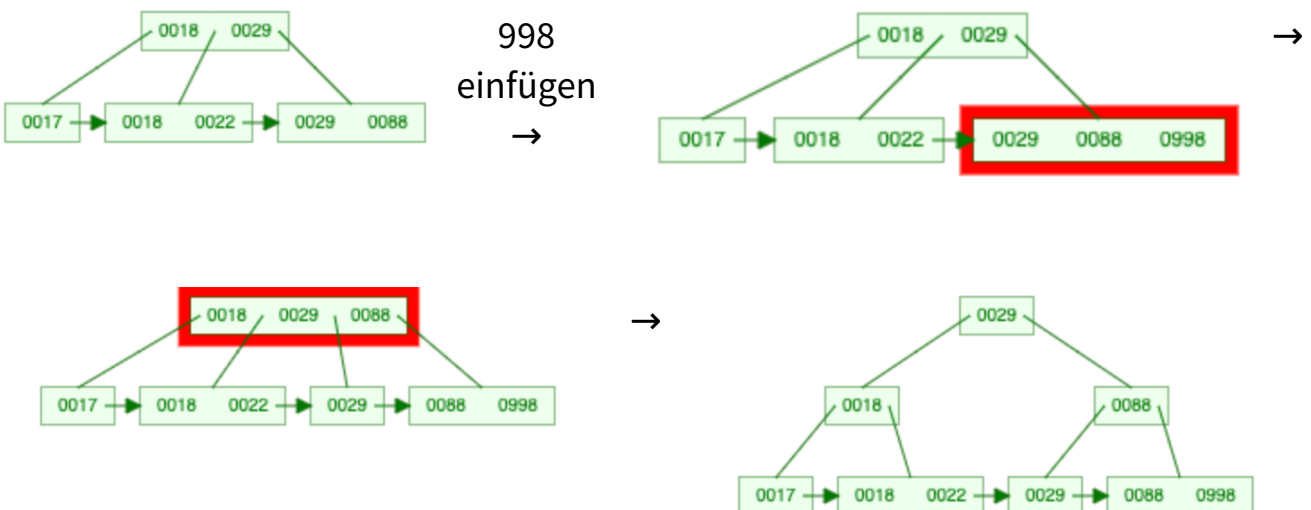
- Knoten wird in zwei Knoten aufgesplittet
- Mittleres Element wandert zusätzlich als Wegweiser in Vaterknoten
- Evtl. ist dort ebenfalls ein Split nötig, wenn er voll ist



Die einzufügende 22 gehört in den Blattknoten links, da sie kleiner als 29 ist. Da dieser Knoten voll ist, erfolgt ein Split. Es gibt nun die 17, 18 und 22. Mittleres Element davon ist die 18. Alles, was kleiner als 18 ist (also die 17) kommt in einen Blattknoten, alles, was größer oder gleich 18 ist (18 und 22) in einen zweiten Blattknoten. Die 18 gelangt als Wegweiser in den bereits vorhandenen Vaterknoten.

43

Einfügen in B+-Baum

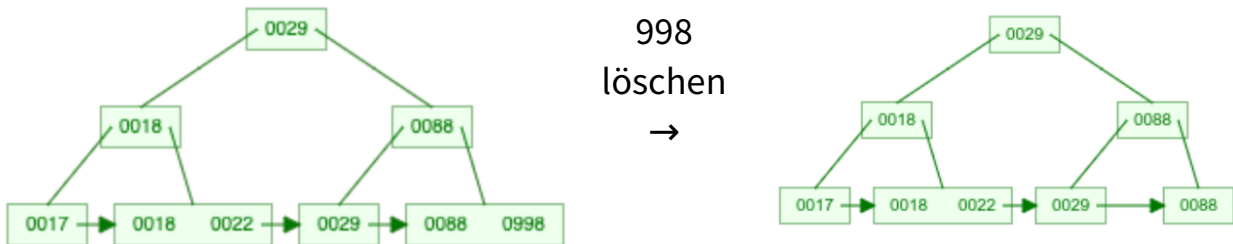


Fügen wir nun die 998 ein, ist ihr Platz im Blatt ganz rechts. Da dieses jedoch voll ist, muss wieder gesplittet werden: Es geht um die 29, 88 und 998. Mittleres Element ist 88, sie wandert in den Vaterknoten. Die beiden neuen Blätter sind [29] und [88, 998]. In deren Vaterknoten wären nun also die Wegweiser 18, 29 und 88. Da dort nur Platz für $2k=2$ Wegweiser ist, erfolgt auch hier ein Split. Das mittlere Element (29) wandert in den Vaterknoten (wird neu erstellt), die 18 links, die 88 rechts davon.

44

Löschen im B+-Baum

1. Blattknoten suchen, in welchem sich der zu löschende Schlüssel befindet
2. a. Danach noch mind. k^* Einträge? \Rightarrow einfach löschen



Eine Löschung ist am einfachsten, wenn dabei kein Unterlauf entsteht. Dies ist der Fall, wenn wir in unserem B+-Baum den Schlüssel 998 löschen. In jedem Blatt müssen mindestens k^* ($=1$) Einträge sein. Das ist hier nach der Löschung der 998 im Blattknoten unten rechts der Fall, also muss nichts weiter unternommen werden.

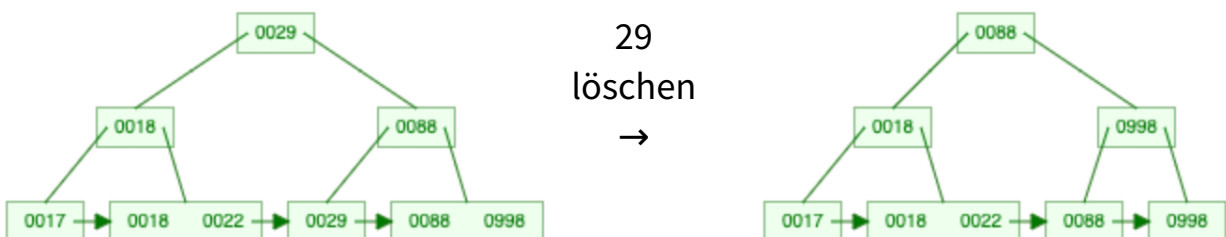
45

Löschen im B+-Baum

1. Blattknoten suchen, in welchem sich der zu löschende Schlüssel befindet
2. a. Danach noch mind. k^* Einträge? \Rightarrow einfach löschen
b. Unterlauf \Rightarrow mit dem Nachbarknoten ausgleichen oder mischen

Ausgleich

- Einträge vom Nachbarknoten übernehmen zum Ausgleichen
- Wegweiser im Vaterknoten entsprechend anpassen



Gibt es in einem Nachbarknoten (erst links schauen, dann rechts) mehr als k^* Einträge, kann ein Ausgleich mit diesem vorgenommen werden. Löschen wir in unserem B+-Baum die 29, haben wir einen Unterlauf. Da es im rechten Nachbarblatt genügend Einträge gibt, können wir die 88 einfach nach links verschieben. Der Wegweiser im Vaterknoten muss angepasst werden, weil rechts Einträge ≥ 998 sind.

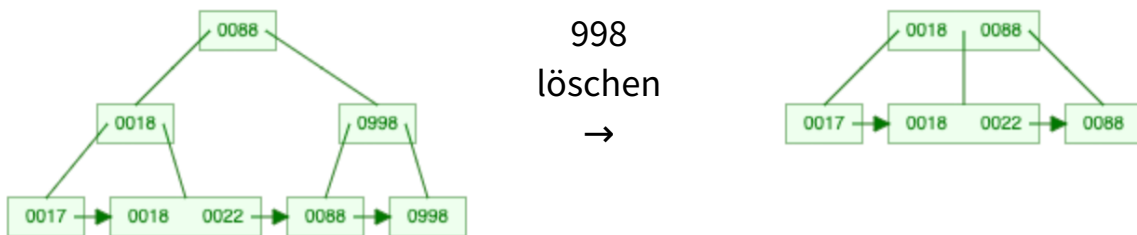
46

Löschen im B+-Baum

1. Blattknoten suchen, in welchem sich der zu löschende Schlüssel befindet
2. a. Danach noch mind. k^* Einträge? \Rightarrow einfach löschen
b. Unterlauf \Rightarrow mit dem linken Nachbarknoten ausgleichen oder mischen

Mischen (Merge)

- Falls Nachbarknoten auch unterlaufen würden \Rightarrow mischen
- Wegweiser aus dem Vater entfernen
- Evtl. ist dort ebenfalls ein Ausgleich / Merge nötig



Löschen wir die 998, entsteht ein Unterlauf und es ist kein Ausgleich möglich. Daher mischen wir 88 und 998 zu einem neuen Blattknoten und entfernen den Wegweiser 998 aus dem Vaterknoten. Da dieser nun unterläuft, wird hier wieder versucht auszugleichen - ohne Erfolg. Daher erfolgt eine Mischung der Knoten [18], [88], [] zu [18, 88].

47

CREATE INDEX

🔊 108

CREATE INDEX erstellt in den meisten DBMS einen B+-Baum-Index.

```
CREATE INDEX produkte_pkey ON produkte(produktnr);
```

Kosten

- Bei INSERT, UPDATE, DELETE muss der Index entsprechend gepflegt werden

Nutzen

- Schnellere exakte Suche (=) und Bereichssuche (<, <=, >, >=, BETWEEN)
- Index kann auch bei ORDER BY, GROUP BY und Joins helfen.

Die meisten DBMS legen Indexe auf Primärschlüsselspalten automatisch an.

Indexe lohnen sich besonders auf Spalten, auf denen oft eine Filterung in einem WHERE-Prädikat erfolgt, z. B. WHERE produktnr=17 oder WHERE bestelldatum BETWEEN '2020-01-01' AND '2020-01-31'. Auf DATE- und TIMESTAMP-Spalten erfolgen oft solche Anfragen, zudem tauchen diese Spalten oft im ORDER BY auf, daher sind hier Indexe oft sinnvoll. Auch bei Prefix-Suchen (LIKE 'A%') hilft ein Index.

48

Mehrdimensionale Indexe

```
CREATE INDEX ON meine_tabelle(a, b, c);
```

Kann genutzt werden bei:

- WHERE a = 5 AND b = 7 AND c = 2
- WHERE a = 5 AND b = 7
- WHERE a = 5
- Bei B+-Baum-Indexen auch bei <, <=, >, >=, BETWEEN, LIKE '...%'

Ein kombinierter Index, der aus mehreren Spalten besteht, ist günstiger als das Erstellen und die Wartung mehrerer einzelner Indexe. Wichtig ist jedoch die Reihenfolge, in der man die Spalten angibt. Der hier gezeigte Index auf den Spalten a, b und c kann z. B. nicht für eine Anfrage WHERE b=7 verwendet werden.

49

Join-Algorithmen

```
SELECT * FROM produkte p JOIN hersteller h
      ON p.hersteller=h.firma;
```

Es gibt viele Arten, einen Join auszuführen:

- *Nested-Loop-Join*: Für jede Zeile der ersten Tabelle wird mit jeder Zeile der zweiten Tabelle überprüft, ob das Join-Prädikat wahr ist.
- *Sort-Merge-Join*: Beide Tabellen werden nach der Join-Spalte sortiert, dann beide von oben nach unten durchgescannt, um Join-Partner zu suchen.
- *Hash-Join*: Hash-Tabelle wird für kleinere Tabelle angelegt, dann die größere Tabelle durchlaufen und Join-Partner mittels der Hash-Tabelle gesucht.
- *Index-Join*: Während eine Tabelle komplett durchlaufen wird, werden für jede Zeile die Join-Partner in der anderen Tabelle mittels eines Indexes gesucht.

Nested-Loop-Join

```
SELECT * FROM produkte p JOIN hersteller h
      ON p.hersteller=h.firma;
```

Für jede Zeile der ersten Tabelle R wird mit jeder Zeile der zweiten Tabelle S überprüft, ob das Join-Prädikat wahr ist.

```
for each row p in PRODUKTE:
  for each row h in HERSTELLER:
    if p.hersteller == h.firma:
      emit(p, h)
```

Aufwand: $O(|R| \cdot |S|)$

51

Sort-Merge-Join

```
SELECT * FROM produkte p JOIN hersteller h
      ON p.hersteller=h.firma;
```

Beide Tabellen werden nach der Join-Spalte sortiert, dann beide von oben nach unten durchgescannt, um Join-Partner zu suchen.

produktnr	bezeichnung	preis	hersteller	firma	land
29	Spülmaschinentabs	3.99	Calgonte	Calgonte	Italien
17	Schokoriegel	0.89	Monsterfood	Holzkopf	Österreich
18	Müsliriegel	1.19	Monsterfood	Monsterfood	USA
88	Katzenfutter	4.99	-		
91	Maschinenbau- Lehrbuch	22.9	-		

Aufwand: $O(|R| \cdot \log(|R|) + |S| \cdot \log(|S|))$

Der Aufwand beim Sort-Merge-Join ist im Wesentlichen der Aufwand, die beiden Tabellen zu sortieren.

52

Hash-Join

```
SELECT * FROM produkte p JOIN hersteller h  
      ON p.hersteller=h.firma;
```

Hash-Tabelle wird für kleinere Tabelle angelegt, dann die größere Tabelle durchlaufen und Join-Partner mittels der Hash-Tabelle gesucht.

```
hash_tabelle = []  
  
for each row h in HERSTELLER:  
    hash_tabelle.put(hash(h.firma), h)  
  
for each row p in PRODUKTE:  
    h = hash_tabelle.get(hash(p.hersteller))  
    emit(p, h)
```

Aufwand: $O(|R| + |S|)$

53

Index-Join

🔊 109

```
SELECT * FROM produkte p JOIN hersteller h  
      ON p.hersteller=h.firma;
```

Während eine Tabelle komplett durchlaufen wird, werden für jede Zeile die Join-Partner in der anderen Tabelle mittels eines Indexes gesucht.

```
for each row p in PRODUKTE:  
    h = hersteller_firma_idx.get(p.firma)  
    emit(p, h)
```

Aufwand hängt von den Index-Kosten ab, z. B. $O(|R| \cdot \log(|S|))$, wenn es einen B+-Baum-Index auf S gibt, oder $O(|R|)$, wenn ein Hash-Index existiert.

Der Index-Join ist ähnlich zum Hash-Join, bis auf den Fakt, dass hier der Index bereits vorher existieren muss. Man kann einen Hash-Join auch als Index-Join sehen, bei dem der Hash-Index extra für den Join erstellt und danach wieder weggeworfen wird.

54

Kapitelzusammenfassung

- DB-Anwendungen: Embedded SQL, JDBC
- JDBC: DriverManager, Connection, Statement, PreparedStatement, ResultSet
- SQL-Injections
- Benutzerdefinierte Routinen: Stored Procedures, (Tabellen-)Funktionen
- Trigger: BEFORE / AFTER / INSTEAD OF
- Indexe: Clustered Index, B-Baum, B+-Baum
- Join-Algorithmen: Nested-Loop-Join, Sort-Merge-Join, Hash-Join, Index-Join