Datenbanken II

Tagesthema:	Gesamtdokument
Dozent:	Herr Fuß, IT-systemworks
Datum / Dauer:	25.05.04 / -
Raum:	-
Dokument Nr.	db2_full

Marktübersicht:

Universell

- Oracle
- DB2

Bis 50 User / 2-3 GB

- SQLServer/Access
- MySQL
- Gupta SQL Base

- Fast DB
- Lotus
- Clipper / dBase

Literatur

- Oracle Essentials, O'Reilly

1. Kurzübersicht SQL

Structured Query Language

- SQL-1
- SQL-2
- SQL-3

1.1. Data Definition Language (DDL)

DDL-Statements sind alle Statements, die Objekte in der DB anlegen oder ändern.

```
CREATE TABLE Buch
(     Buch_ID     number,
      Titel       varchar(200),
      Autor       varchar(200),
      Verlag      varchar(200));
```

Bei der Gruppe der DDL Statements gibt es eine hohe Herstellerabhängigkeit bei der Syntax.
→ Handbuch lesen.

1.2. Data Manipulation Language

Alle Statements, die Inhalte in den Objekten ändern oder anzeigen

```
SELECT
INSERT
UPDATE
DELETE
```

1.2.1. INSERT

→ Einfügen neuer Daten

```
INSERT INTO Buch
(     Buch_ID, Titel, Autor, Verlag)
VALUES
(     1, ‚Der Herr der Ringe', ‚Tolkien', ‚Klett');
```

Buch_ID	Titel	Autor	Verlag
1	Der Herr der Ringe	Tolkien	Klett
15	*FAZ*		
16	*Bild*		*Springer*

Via INSERT … SELECT

Einfügen von mehreren Werten mit einem INSERT Statement:

Buch

Buch_ID
…

Zeitung

Zeitung_ID
Titel
Verlag

```
INSERT INTO Buch
(     Buch_ID, Titel, Verlag)
SELECT
      Zeitung_ID, Titel, Verlag
FROM Zeitung;
```

1.2.2. UPDATE

→ an den bestehenden Daten

```
UPDATE Buch
SET Autor = ‚niemand'
WHERE Autor is null;
```

Ändert alle Zeilen, bei denen Autor nicht befüllt ist. Dabei wird Autor auf den festen Wert ‚niemand' gesetzt.

1.2.3. DELETE

→ löscht Zeilen

```
DELETE FROM Buch
WHERE Autor = 'Tolkien';
```

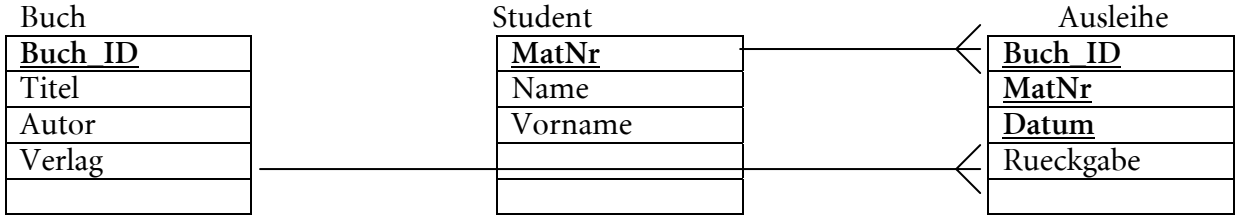
1.2.4. SELECT

(je nach Autor DQL, Data Query Language)
→ Abfragen von Dateninhalten

```
SELECT * FROM Buch;
```

Liefert alle Inhalte (in allen Spalten) in Buch

```
SELECT Buch_ID, Verlag
FROM Buch
WHERE Autor = ‚Tolkien';
```




```
SELECT      Buch.Titel
            Buch.Autor,
            Ausleihe.Datum
FROM        Buch,
            Ausleihe,
            Student
WHERE       Student.Name = 'Maier'
AND         Student.Vorname = 'Hans'
AND         Student.MatNr = Ausleihe.MatNr
AND         Ausleihe.Rueckgabe is null
AND         Buch.Buch_ID = Ausleihe.Buch_ID;
```

→ liefert alle Bücher, die Hans Maier zurzeit ausgeliehen hat.

JOIN: Verbindung mehrere Tabellen in einem Statement

Logisch wird beim JOIN das Kreuzprodukt aller beteiligten Tabellen gebildet und darauf die WHERE Bedingung angewendet.

INNER JOIN: Beim Vergleich zweier Spalten aus zwei Tabellen muss in beiden Tabellen der Wert vorhanden sein.

```
SELECT      s.MatNr,
            a.Buch_ID
FROM        Student s,
            Ausleihe a
WHERE       s.MatNr = a.MatNr (+)
```

Das (+) bezeichnet die Spalte, die leer bleiben darf (outer join)

MatNr	Buch_ID
123	1
123	2
456	Null
789	47

Alternative Syntax:

```
SELECT      s.MatNr,
            a.buch_id
FROM        student
LEFT OUTER JOIN
            Ausleihe a
```

OUTER JOIN: Join, bei dem auch Rows der einen Tabellen aufgenommen werden, zu denen es in der anderen Tabelle kein Äquivalent gibt.

Operatoren:

\+ - * / ( )

```
SELECT      gehalt * 10
FROM        gehalt;

SELECT      titel * 10
FROM        buch;
```
→ Gefahr von Laufzeitfehlern

Vergleiche:
=, >, <, >=, <=, =>, <=, !=, <>, is null, is not null

Logische Verknüpfungen:
AND, OR, NOT

```
SELECT      *
FROM        Buch
WHERE       Titel = 'Der Herr der Ringe'
OR          autor = ,Tolkien';
```

LIKE
→ Suche mit Wildcard

```
SELECT      *
FROM        Buch
WHERE       Autor LIKE 'T%';
```
→ liefert alle Bücher, deren Autor mit (Gross) T beginnt, der Autor „tolkien" wird nicht gefunden.

% eine beliebige Anzahl von Zeichen
_ genau ein beliebiges Zeichen

UPPER, LOWER, INITCAP

```
SELECT UPPER (,Tolkien') …
```
→ Liefert TOLKIEN

```
SELECT      *
FROM        Buch
WHERE       UPPER (Autor) = UPPER (,tolkien');
```

Vorsicht bei Sonderzeichen!

```
SELECT INITCAP ('der herr DER ringe');
```
→ Der Herr Der Ringe

IN

```
SELECT      *
FROM        Buch
WHERE       Autor IN (,Tolkien', ,Konsalik');

SELECT      *
FROM        Buch
WHERE       Buch_ID IN
      (SELECT buch_ID FROM ausleihe
      WHERE Rueckgabe is null);
```

EXISTS

```
SELECT      *
FROM        Buch
WHERE EXISTS
      (SELECT * FROM ausleihe
      WHERE Buch.buch_ID = ausleihe.buch_id);
```

Aggregierende Funktionen

```
SELECT      COUNT(*)
FROM Buch;
```
→ Anzahl der Bücher

```
SELECT      COUNT(DISTINCT Autor)
FROM Buch;
```
→ Anzahl unterschiedlicher Autoren

```
SELECT autor, COUNT(*)
FROM Buch
GROUP BY Autor;
```
→ Tolkien 5
Konsalik 10

```
SELECT      COUNT (DISTINCT Buch.buch_id) –
            COUNT(DISTINCT Ausleihe.Buch_id)
FROM        Buch, Ausleihe
WHERE       Buch.buch_id = Ausleihe.buch_id(+)
```
→ liefert die Anzahl der nicht ausgeliehenen Bücher.

```
SELECT      Autor, COUNT(*)
FROM        Buch
GROUP BY    Autor
HAVING      COUNT(*) > 1;
```
→ alle Autoren, die mehr als ein Buch haben.

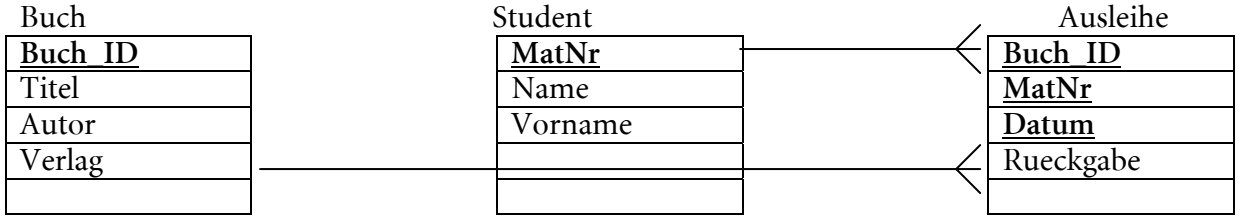
```
SELECT      Autor, MIN(Titel)
FROM        Buch
GROUP BY    Autor;
```
→ je Autor den kleinsten Titel ausgeben.

Übung:

a) Geben Sie je Student (MatNr, Name, Vorname) die ausgeliehenen Bücher (Autor, Titel, Verlag) mit einem SQL Statement aus.
b) Bestimmen Sie je Student (MatNr, Name, Vorname) die Anzahl der ausgeliehenen Bücher.
c) Besimmen Sie je Student die mittlere Ausleihedauer.

```
SELECT      to_date(‚02.01.2004’) –
            to_date(‚01.01.2004’) …
```

Liefert 1 Tag Differenz



Lösung:
a) SELECT

1.3. Transaktionskontrolle

Bestellung

Best_Nr
Datum
Kunde_Nr

Best_Artikel

Best_Nr
Art_Nr
Anzahl

```
INSERT INTO Bestellung
      (Best_Nr, Datum, Kunde_Nr)
VALUES
      (123, ,01.04.04', 4711);
```

← Bestellung von Hans Müller am 01.04.2004

```
INSERT INTO Best_Artikel
      (Best_Nr, Art_Nr, Anzahl)
VALUES
      (123, 456, 10);
```

← Hans Müller hätte gerne 10 CD-Rohlinge

```
INSERT INTO Best_Artikel
      (Best_Nr, Art_Nr, Anzahl)
VALUES
      (123, 789, 5);
```

← und zusätzlich 5 x 100 GB Festplatten

`Commit`
→ Bestätigt eine Transaktion und schreibt alle Daten der Transaktion fest in die Datenbank.

`Rollback`
→ verwirft die letzte Transaktion.

Die erste Transaktion beginnt mit dem Verbindungsaufbau des Frontendprozesses mit der Datenbank. Jedes commit/rollback beendet die aktuelle Transaktion und beginnt eine neue Transaktion.

Viele Hersteller kennen einen „Autocommit“. Dieser wird in der Regel auf Sessionbasis aktiviert. Durch den „Autocommit“ wird durch die Datenbank nach jeder Datenveränderung ein „commit“ selbständig durchgeführt.
Über „begin work“ lässt sich bei Autocommit eine echte Transkation aufbauen.

ACHTUNG: Autocommit birgt die Gefahr von statistischen Programmfehlern.
Im Normalfall sind die Veränderungen in der aktuellen Transaktion nur für die eigene Session sichtbar. Alle anderen Sessions sehen die veränderten Daten erst nach dem commit.

Zeit	Session #1	Session #2
10:00	`SELECT COUNT(*) FROM Artikel;` → 0 Rows	

10:01	`INSERT INTO Artikel`
	`  (Art_Nr, Bezeichnung, Preis)`
	`VALUES`
	`  (4711, 'CD-Rohling', 1,50);`	

10:02	`SELECT * FROM Artikel;` → 4711, CD-Rohling, 1,50	
10:03		`SELECT COUNT(*) FROM Artikel;` → 0 Rows
10:04	`Commit;`	

10:05		`SELECT *`
		`FROM Artikel;`
		→ 4711, CD-Rohling, 1,50

10:06		`INSERT INTO Artikel`
		`  (…)`
		`VALUES`
		`  (123, 'Festplatte', 100);`

10:07	`UPDATE Artikel` `SET preis = preis * 0,8;`	
10:08	`Commit;`	`Commit;`
10:09	`SELECT * FROM Artikel;` → 4711, CD-Rohling, 1,20 → 123, Festplatte, 200	`SELECT * FROM Artikel;`

Savepoint:
Das Setzen eines Markers bis zu dem eine Transaktion mit einem *rollback to savepoint* rückgängig gemacht werden kann.

```
SELECT COUNT(*)
FROM Artikel;
→ 0 Rows

INSERT INTO Artikel
      (…)
VALUES
      (4711, 'Rohling', 1,50);

SAVEPOINT vor_update;

UPDATE Artikel SET preis = 0;

ROLLBACK TO SAVEPOINT vor_update;

COMMIT;

SELECT * FROM Artikel;
→ 4711, Rohling, 1,50
```

Verwerfen der Änderungen bis zum Savepoint

2. Data Dictionary

Im Data Dictionary werden in Form von Tabellen die Objekte der Datenbank verwaltet. Häufig liegen die Tabellen des Data Dictionary in einem eigenen User. Der Aufbau des Data Dictionary ist nicht standardisiert.
Für Oracle gilt:

- Data Dictionary liegt im User SYS
- Zur Abfrage des Dictionarys gibt es zwei Gruppen von Views:
  - dba_
    dba_*: alle Objekte
  - user_
    user_*: Tabellen, Indizes des aktuellen Benutzers

Artikel

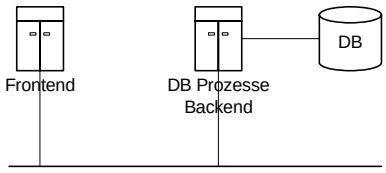
Art_Nr
Bezeichnung
Preis

```
SELECT * FROM user_tables;
```
→ Artikel …

```
SELECT * FROM user_tab_colums
WHERE table_name = 'ARTIKEL'
```
→ ArtNr NUMBER
Bezeichnung VARCHAR(80)
Preis NUMBER

Analog: dba_tables
dba_tab_columns

3. Integration von DB Abfragen im Frontend



Session:
Verbindung mit der Datenbank. Die Session beginnt mit der Anmeldung (User/Password) und hält den Kontext für die nachfolgenden Statements.

Cursor:
Der Cursor dient der Abarbeitung von einem Statement. In einer Session können mehrere Cursor gleichzeitig offen sein.

DB	Filesystem
Session	Netzwerkanmeldung
Cursor	Filehandle

3.1. JAVA

```
Import java.sql.*;
DriverManager.registerDriver (new CTreiber);
```
→ Bekanntgabe, welcher Datenbank-Treiber benutzt werden soll.

```
Connection con = DriverManager.getConnection (<Datenbank>, <User>, <Pwd>);
```
→ Sessionaufbau

```
Statement stmt = con.createStatement();
```
→ Cursor anmelden

```
Resultset rset = stmt.executeQuery ("SELECT Bezeichnung FROM Artikel");
```
→ Abfrage an DB schicken, Ergebnis in *rset* speichern.

```
While (rset.next())
      System.out.println (rset.getString(1));
```
→ Zeilenweise das Resultset holen und ausgeben.

```
rset.close();
```
→ Resultset schließen

```
stmt.close();
```
→ Cursor schließen

```
con.close();
```
→ Session schließen

3.2. PHP

```
$link = mysql_connect (<Datenbank>, <User>, <Pwd>);
```
→ Session aufbauen

```
$query = "SELECT Bezeichnung FROM Artikel;";
```
→ Stringzuweisung

```
$result = mysql_query($query);
```
→ Cursor anmelden und Statement ausführen

```
While ($row = mysql_fetch_array($result))
{ print $row[0]; }
```
→ Zeilenweise Resultset lessen und ausgeben

```
Mysql_free_result($result);
```
→ Resultset freigeben

```
Mysql_close($link);
```
→ Session schließen

3.3. C / C++

Zwei Varianten

a) Bibliothek mit Funktionsaufrufen
   - ADO, OLEDB (Microsoft)
   - DLL des DB-Herstellers

b) Präprozessoren
Vorverarbeitung des Quelltextes vor dem eigentlichen Compilieren. Dabei werden Teile des Quelltextes durch Bibliotheksaufrufe ersetzt.

```
int ArtNr;
SELECT ArtNr
INTO ArtNr
FROM Artikel;
```

← Wird durch den Präprozessor in korrektes C umgesetzt.

```
While (fetchnext()) {…}
```

Vorteil: Das explizite Binden der Variable entfällt. "Bindvariablen" können direkt den Zusammenhang zwischen SQL-Statement und Variablen des Programms herstellen.
Viele Präprozessoren prüfen die Syntax des SQL-Statements gegen eine Testdatenbank.

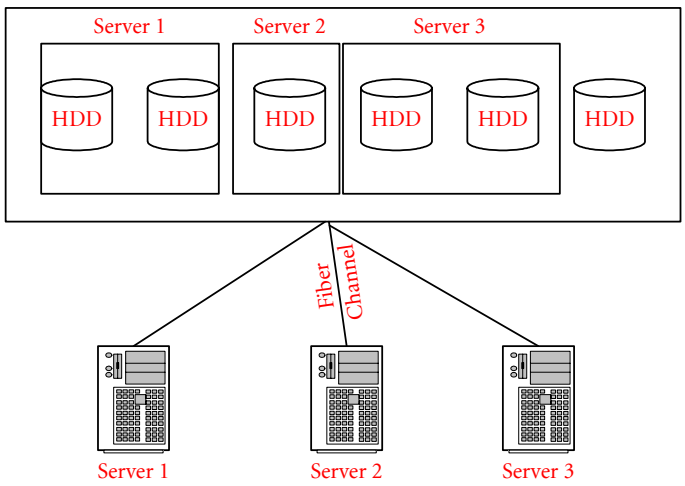
Nachteil: Zeilennummern des Compilerlaufs hat nichts mit der echten Quelltextzeile zu tun. Debugging wird erschwert.

3.4. 4GL Sprachen (4th Generation Language)

- Oracle Forms
- Centura/Gupta Team Developer

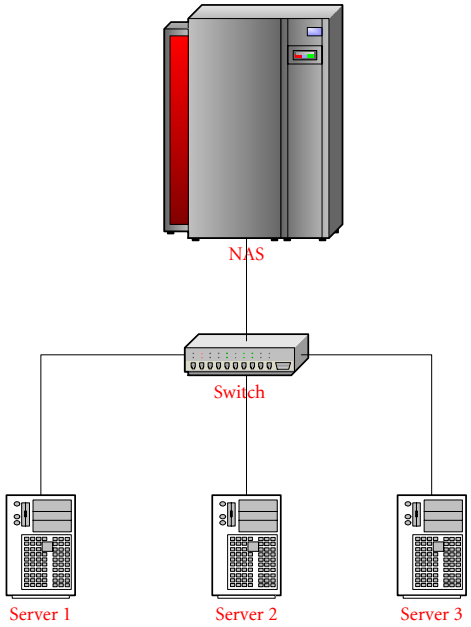
4. Storagesysteme

SAN: Storage Area Network



Konzept:
Plattenkapazität nicht direkt in den Servern einzubauen, sondern in einem getrennten Gerät. Die Server werden an das SAN angeschlossen und die Platten im SAN den einzelnen Server „on demand" zugeordnet.

NAS: Network Attached Storage
Vorteil NAS: einfache Installation; nutzt bestehende Verkabelung
Vorteil SAN: schneller

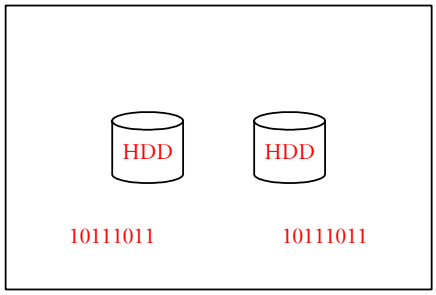


4.1. Fehlertoleranz

RAID: Redundant Array of Independent Discs

RAID 0:
Array besteht aus einer Platte. → häufig mit „Striping“ gemischt
Fehlertoleranz: 0

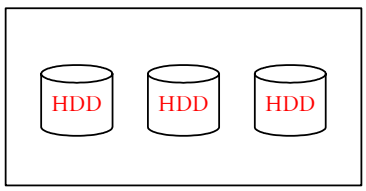
RAID 1: Spiegelung
Logische Platte / Array



Platte 2 enthält 1:1 die Daten von Platte 1

Fehlertoleranz: Fällt eine Platte aus, so ist das Array immer noch arbeitsfähig.
Nutzkapazität: Plattenkapazität / 2

RAID 5:



Redundanz über Parityverfahren

Paritybit bei serieller Übertragung

1	0	1	0	+	1	1	1	0

→ ermöglicht das Erkennen von 1-Bit Fehlern

1	0	1

Jedes dieser Bits liegt auf verschiedenen Platten.
Erstes Bit auf Platte 1, zweites Bit auf Platte 2, drittes Bit auf Platte 3. Dadurch ist eine Rückrechnung der Daten von zwei Platten auf die Information der dritten Platte möglich.

Ein RAID 5 wird aus mindestens drei Platten aufgebaut.

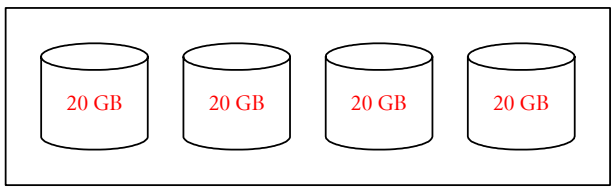
Fehlertoleranz: Ausfall einer Platte wird ohne Datenverlust verkraftet
Nutzkapazität: (n-1) * Plattenkapazität
n = Plattenanzahl

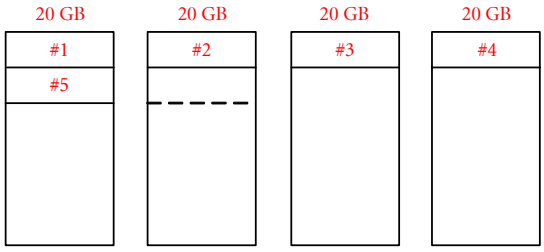
Beispiel: 3 x 20 GB → 40 GB

Vorteile / Nachteile:
RAID 0: - keine Fehlertoleranz
RAID 1: + Fehlertoleranz
- Hohe Kosten mit geringer Nutzkapazität
- hohe Lesegeschwindigkeit bei mehreren Zugriffsprozessen
RAID 5: + Fehlertoleranz
+ hohe Nutzkapazität im Vergleich zu RAID 1 geringere Kosten
- langsame Lesegeschwindigkeit, da von mehreren Platten gelesen wird.

Daumenregel:
RAID 5: Fileserver
RAID 1: DB-Server

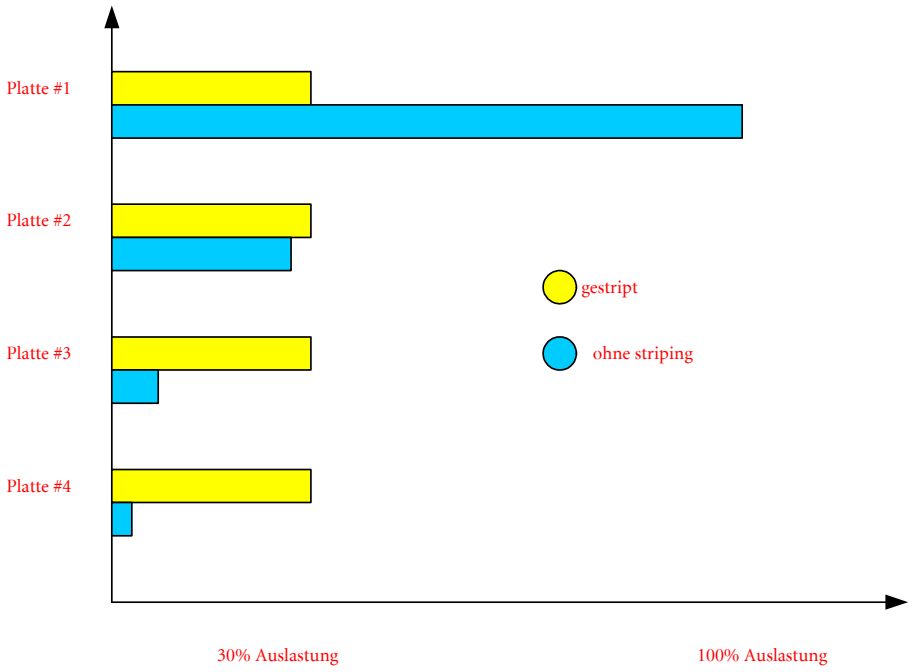
4.2. Striping





Slice: Ein zusammenhängender Block auf einer Platte. Alle Slices haben die gleiche Größe.

Beim Striping wird der logische Speicherraum gleichmäßig auf alle Platten verteilt. Ein Leseprozess wechselt sehr schnell von einer Platte auf die nächste. Greifen viele Prozesse auf die Platten zu, so werden alle Platten gleichmäßig belastet. Beim Schreiben werden die anfallenden Daten ebenfalls gleichmäßig verteilt.
Es wird so verhindert, dass eine Platte zum Flaschenhals des Systems wird.



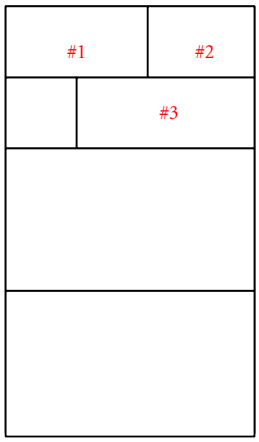
5. Physikalischer Aufbau von Datenbank-Objekten

Person
Name Vorname

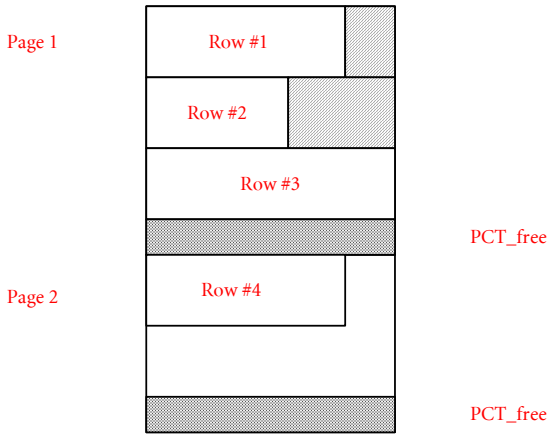
- CHAR - String fester Länge
- VARCHAR - String variabler Längen
- NUMBER - Zahlen
- DATE/TIME

```
PersNr NUMBER
Name VARCHAR(80)
Vorname VARCHAR(80)
```

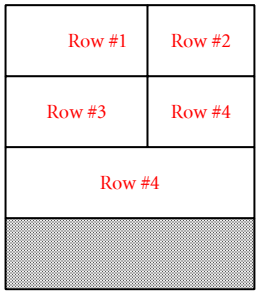
PersNr	**Name**	**Vorname**	*Platzbedarf*
1	Müller	Kurt	4 + 7 + 5 = 16 Byte
2	Becher	Heinz	4 + 7 + 6 = 17 Byte
3	Schmidt	Rudolph	4 + 8 + 8 = 20 Byte
			Gesamt: 53 Byte

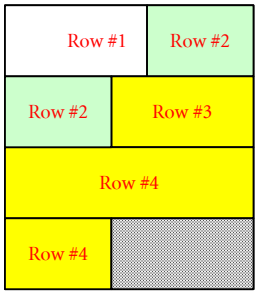


Jedes Objekt wird in gleichmäßig große Blöcke (Databaseblock / Databasepage) aufgeteilt. Das Speichermanagement für die einzelnen Rows findet in den DB-Blöcken statt.



Beim Befüllen durch INSERTs werden die Rows in die DB-Page geschrieben. Aufgrund variabler Datentypen sind die Rows unterschiedlich lang.
Ist eine Page bis zum PCT_free Wert gefüllt, so finden keine weiteren INSERTs auf die Page statt. (Die „volle“ Page wird aus den Freelist genommen.) Für weitere INSERTs wird der nächste Block aus der Freelist genommen.



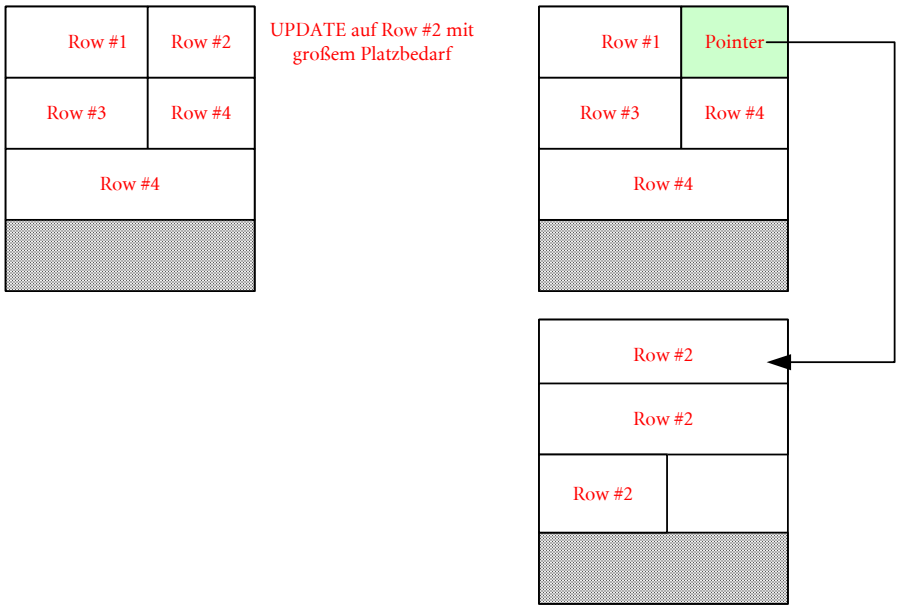


Beim Update einer Row wird diese an ihrer Position ggf verlängert und alle nachfolgenden Rows nach hinten verschoben.

Es wird somit der wegen PCT_Free frei gelassene Speicherbereich aufgebraucht.

5.1. Rowchaining

Ist für den Update der freie Platz zu gering, so wird über Rowchaining auf einen Hilfsblock zurückgegriffen.



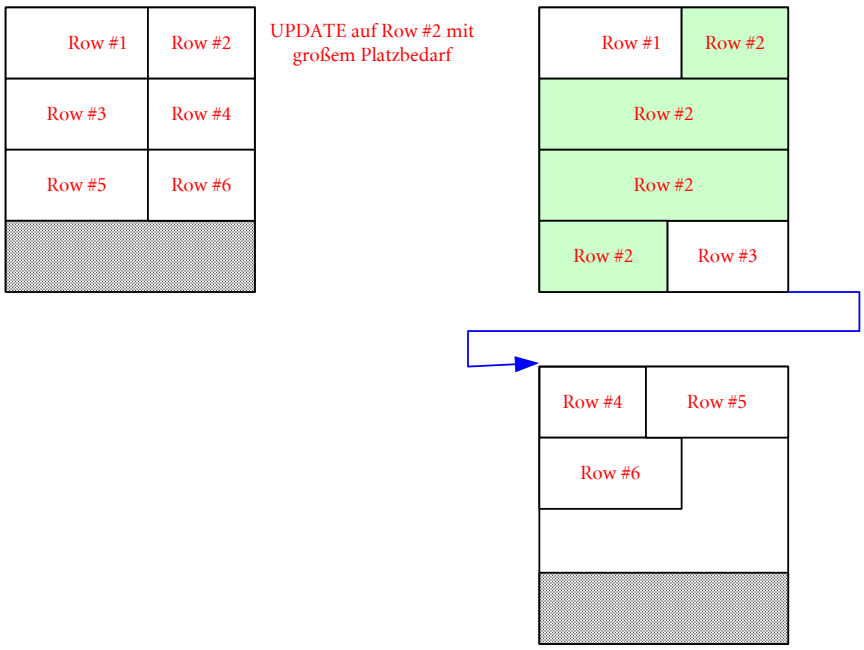
An die Stelle von Row2 wird ein Pointer auf einen neuen Block mit der Row2 gestellt.

Vorteil: Update selbst ist schnell

Nachteil: Zusatzaufwand beim Lesen von Row2.

5.2. Block Chaining

Beim Block Chaining wird Row2 in der ersten Page bleiben. Alle nachfolgenden Rows werden nach hinten geschoben. Alle Rows, die keinen Platz mehr finden, werden in einen neuen Block geschoben.



→ Langsamer

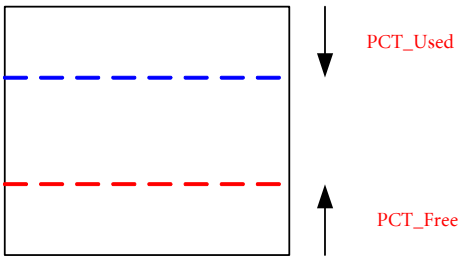
Vorteil: Update selbst ist schnell

Nachteil: Zusatzaufwand beim Lesen von Row2

Am besten: Chaining vermeiden, besser Rowchaining als Blockchaining. PCT_Free-Wert passend wählen, je nach UPDATE-Häufigkeit größer oder kleiner.

5.3. PCT_Free, PCT_Used

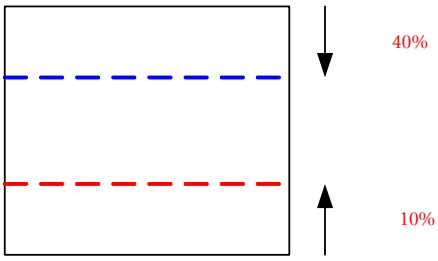
PCT steht für „per cent“



PCT_Free:
Prozentualer Anteil der Page, der durch INSERTs frei bleibt.
Typische Werte: 5 – 40%, Default häufig: 10%

PCT_Used:
Eine Page, deren Befüllung wieder unter den PCT_Free-Wert sinkt, wird erst dann wieder für INSERTs genutzt, wenn auch PCT_Used unterschritten ist (Hysterese). Typischer Wert 40%
→ Lastverteilung

PCT_Free sollte vor Anlage der Tabelle zur Vermeidung von „chained rows“ bzw. „chained blocks“ bestimmt werden.
PCT_Used ist unkritisch und kann zur Optimierung der Speicherausnutzung verändert werden.



Wird eine Tabelle mit Bewegungsdaten befüllt und werden alte Daten regelmäßig reorganisiert, so wird die mittlere Befüllung der Page sich zwischen PCT_Free und PCT_Used einpendeln.

Beispiel:
Regelmäßig reorganisierte Bewegungsdatentabelle

- PCT_Free 10%
- PCT_Used 40%

→ mittlere Befüllung je Page: $\frac{90\% - 40\%}{2} + 40\% = 65\%$

→ 35% des Objektes ist leer.

PCT_Free 5%
PCT_Used 60%

→ mittlere Befüllung je Page: $\frac{95\% - 60\%}{2} + 60\% = 77{,}5\%$

→ 22,5% des Objektes ist leer.

Soll eine Row eingelesen werden, so wird ihre ganze Page von Platte gelesen. Die gelesene Page wird zunächst im DB-Cache gehalten.
Wird eine weitere Row aus der Page benötigt, so kann die Page aus dem Cache genutzt werden. Der Zugriff wird schneller. Je dichter die Rows in den Pages liegen, desto besser ist die Cache-Buffer Hit-Ratio; die Abarbeitung des Statements wird schneller.
Die Wahl von PCT_Free bzw. PCT_Used ist somit entscheidend für die Chained-Row Statistik und der Cache-Buffer Hit-Ratio.

Übung:
Die Tabelle Person

PersNr:	NUMBER
Name:	VARCHAR(80)
Vorname:	VARCHAR(80)
GebDatum:	DATE

NUMBER:	belegt im Mittel 4 Byte
VARCHAR:	belegt echte Länge plus 1 Byte
DATE:	belegt fix 7 Byte

- Wie groß ist die mittlere Rowlänge aller Personen im Kurs?
- Wenn statistisch 50% der Herren heiraten und einen Doppelnamen bilden und alle in die Großfamilie Müller einheiraten, welchen PCT_Free-Wert ist dann sinnvoll?

Lösung der Übung:

Durchschnittliche Länge vor Update = 26 Byte
→ Pagegröße: 2k – 64k
Jede Row wird im Durchschnitt um 3 Byte verlängert. Nach Update durchschnittliche Länge 29 Byte.

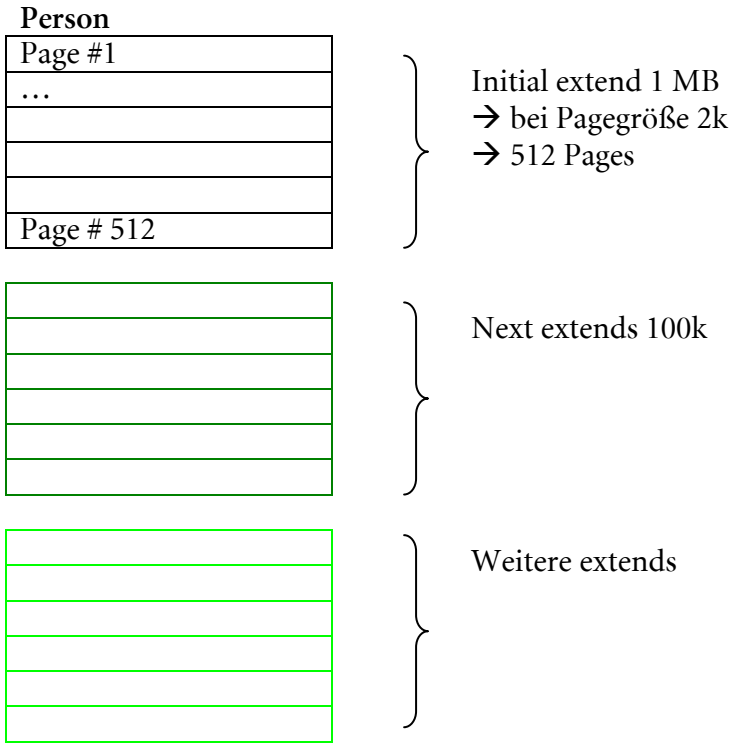
$$\frac{26\ (vorher)}{29\ (nachher)} = 89\%$$

→ PCT_Free: 11 %

Alternativ:
780 Byte Nutzdaten + 94,5 Byte Update = 874,5 Byte

$$\frac{780\ (vorher)}{874{,}5\ (nachher)} = 89\%$$

5.4. Extend



Beim Anlegen eines DB-Objekts wird ein zusammenhängender Speicherblock allokiert (initial extend). Dieser wird ausschließlich für das DB-Objekt benutzt. Ist der „initial extend" voll, so wird ein weiterer Extend automatisch allokiert.
Die Größen für die Extends werden durch drei Parameter bei der Objektanlage definiert.

Initial:	Größe des ersten Extends
Next:	Größe des nächsten Extends
PCTINCREASE:	Prozentuale Wachstum beim Ziehen weiterer Extends.

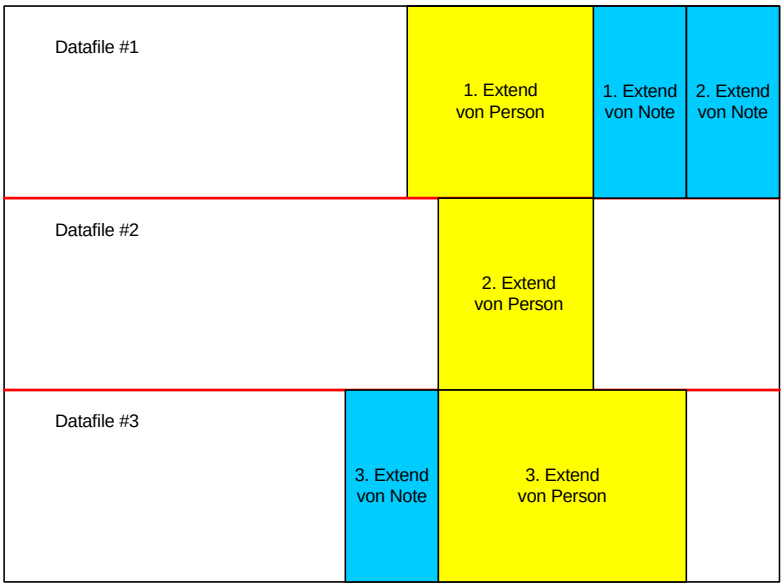
```
CREATE TABLE person
(<Columns>)
Pctfree 5
Pctused 40
Storage     (initial    1M
             next       100k
             pctincrease 50);
```

1. extend:	1M
2. extend:	100k
3. extend:	150k
4. extend:	225k

Ein allokierter Extend wird durch die Datenbank nicht wieder automatisch freigegeben. Über die Parameter *MINEXTENDS* und *MAXEXTENDS* können beim Anlegen einer Tabelle eine feste Anzahl von Extends belegt werden. Beim Erreichen von MAXEXTENDS allokiert die Datenbank keine weiteren Extends mehr. Konsequenz: Der einfügende Prozess erhält einen Fehler.

5.5. Tablespace

Tablespace ist ein logischer Speicherraum. Er setzt sich aus mehreren Datenfiles zusammen und stellt so die Summe der Größen der DB-Files an Speicherplatz zur Verfügung.
Normalerweise muss ein Objekt immer in einem Tablespace liegen.



Ein Extend muss immer in genau einer Datafile liegen. Somit ist die maximale Extendgröße durch die Datafiles begrenzt. Datafiles können jederzeit zu einem Tablespace ergänzt werden.

Für jeden Tablespace kann eine automatische Extendverwaltung aktiviert werden. Die Storageparameter werden dann durch die DB festgelegt.

5.6. Datafile

Ein Datafile ist ein physikalisches File auf der Festplatte. Beim Anlegen eines Datafiles wird der komplette angeforderte Speicherplatz allokiert und formatiert. Beim Formatieren werden interne Verwaltungsdaten im Datafile hinterlegt.
Datafiles können als echte Dateien im Dateisystem liegen oder als RAW-Devices Teile des Betriebssystems umgehen und so ca. 5 % Performance gewinnen. Bei RAW-Devices ist die Administrator schwieriger.

5.7. Andere Verwaltungsstrategien

5.7.1. Ein File je Objekt

Jedes Objekt erzeugt bei der Anlage eine gleichnamige Datei im Dateisystem.

Vorteil: + Einfache Administration beim Einrichten
→ keine Tablespaces
→keine Extendprobleme

Nachteil: - Maximale Dateigröße (BS-Parameter) bestimmt die max. Objektgröße
- Bei vielen Objekten entstehen viele Dateien
→ Verwaltungsoverhead des BS steigt
- Schwierigkeiten bei der Performance
→ MySQL, dBase

5.7.2. Eine Datenbankdatei

Die komplette Datenbank wird in einem File gespeichert. Die Speicherverwaltung wird ausschließlich von der Datenbank durchgeführt.

Vorteil: +„easy-to-use“
→ Administrationsaufwand wird minimiert

Nachteil: - bei großen Datenmengen (> 1GB) schlecht zu administrieren
- max. Objektgröße durch Betriebssystem beschränkt
→ Access

6. Logging

Die Datenbank speichert alle Transaktionen als fortlaufendes Log in den *Redo*-Segmenten. Die *Redo*-Segmente werden kontinuierlich auf Platte geschrieben. Die Veränderung an den Datafiles geschehen zunächst in einem Cache und werden nur bei Bedarf auf Platte geschrieben.

Fällt die Datenbank aus (z. B. Stromausfall) so kann der Cache nicht mehr auf Platte geschrieben werden. Die Inhalte der Datafiles sind beim Absturz nicht konsistent. Beim Neustart der Datenbank stellt diese die Inkonsistenz anhand der Zeitstempel der Datafiles fest.

Anhand der Information aus den *Redos* bildet die Datenbank die nicht oder nur teilweise im Datafile gespeicherten letzten Transaktionen nach (ROLLFORWARD).

Ist das *Redo* bis zum Zeitpunkt des Absturzes verarbeitet, so werden alle noch offenen Transaktionen verworfen (ROLLBACK).
Danach ist der Datenbestand in der Datenbank wieder konsistent.

7. Backup & Recovery

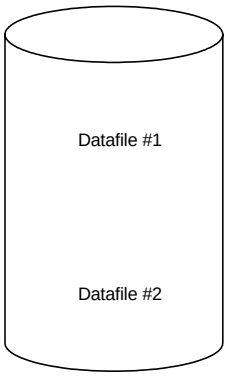
7.1. OFFLine-Backup

Vor dem Sichern wird die Datenbank beendet. Somit haben alle Datenbankdateien einen konsistenten Stand. Da kein Zugriff auf die Datenbank mehr stattfindet, können die Dateien mit normalen Backup-Programmen gesichert werden. Nach der Sicherung wird die Datenbank neu gestartet.

Für das Recovery wird nur die Vollsicherung zurückgespielt.

Vorteil: + preiswert
+ einfach zu administrieren
Nachteil: - es kann nur auf den letzten Backup-Zeitpunkt zurückgespielt werden
- Backup nur offline

7.2. ONline-Backup



Backup-prozess

1.) Hinweis an DB → Sicherheitsmodus
2.) DB: erstes Datafile freigeben, Datafile 1 wird gesichert
3.) DB: zweites Datafile freigeben
4.) …
5.) Backup fertig

Beim Online-Backup werden nacheinander die Datafiles durch den Backup-prozess bei der Datenbank gelockt. Für den Zeitraum des Backups eines Files liest die Datenbank nur noch darin, schreibt aber die Änderungen erst nach Ende der Sicherung des Files.

Ist ein Redo voll, so wird es nicht direkt überschrieben, sondern es wird erst eine Kopie in das Filesystem gelegt (Archiv-Log). Sind alle Datafiles gesichert, werden zusätzlich die angefallenen Archiv-Logs gesichert und anschließend gelöscht.

Auf Band sind nun Datafiles, die nicht zueinander konsistent sind. Die Archiv-Logs beinhalten aber die Redo-Information über den Backup-Zeitraum. Nach dem Rückspielen nutzt die DB beim Start die Archiv-Logs für einen Rollforward / Rollback.

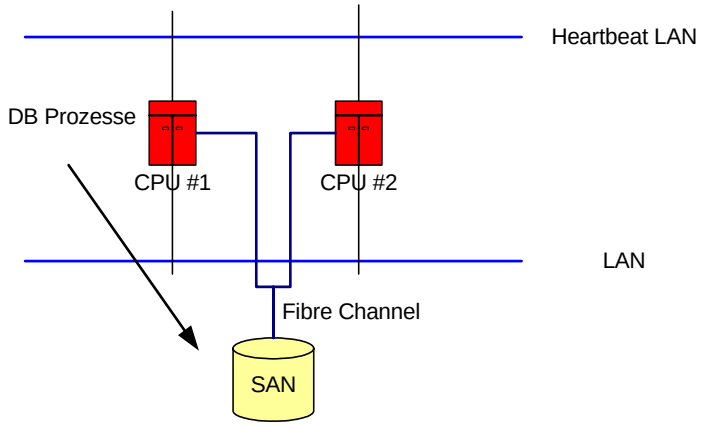
Die Archiv-Logs werden immer geschrieben und können so regelmäßig als inkrementelles Backup gesichert werden.

Beispiel:

- Onlinebackup jeden Samstag 15:00 Uhr
- Alle vier Stunden werden angefallene Archiv-Logs gesichert.
- Systemausfall: Donnerstag 16:00 Uhr
- Recovery
  - Rückspielen der Datafiles vom letzten Samstag
  - Rückspielen der Archiv-Logs vom Samstag 15:00 Uhr bis Donnerstag 16:00
  - DB starten

8. Fehlertolerante Systeme

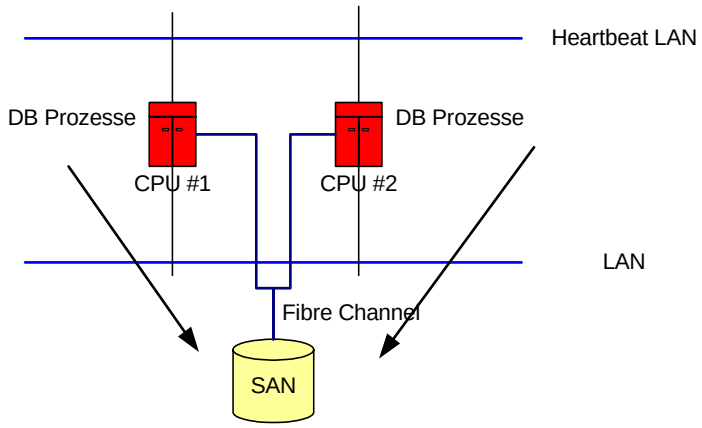
8.1. Betriebssystem-Cluster



Beim Betriebssystem-Cluster wird die Datenbank auf einem der beteiligten Nodes gestartet. Fällt dieser Node aus, so erkennen die verbleibenden Nodes den Ausfall und übernehmen den Dienst. Das heißt eine der verbleibenden Nodes startet die DB-Prozesse.
Die Prozesse der Datenbank verarbeiten die Redos nach und stellen ihren Dienst unter der gleichen IP-Adresse wie zuvor zur Verfügung.

Vorteil: + Beim Ausfall einer Komponente steht Service nach wenigen Minuten wieder zur Verfügung
+ „Preiswert“, da Clusterlizenzen nur auf BS-Ebene anfallen.
Nachteil: - nur Fehlertoleranz, kein Load-Balancing

8.2. Datenbank-Cluster

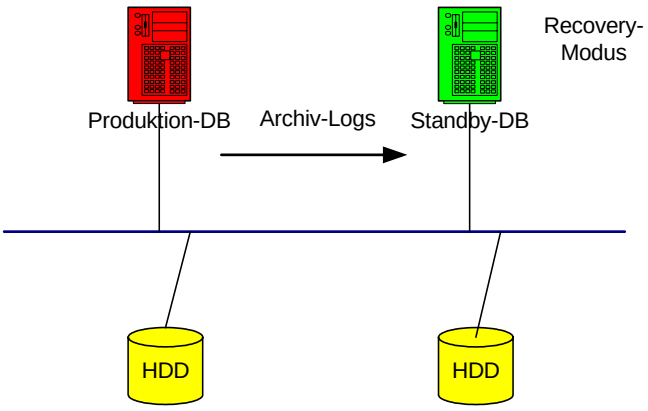


Ein DB-Cluster setzt auf einen BS-Cluster auf. Die DB-Prozesse werden auf allen Nodes gestartet. Benötigt nun ein Node eine Page, so muss bei allen anderen Nodes nachgefragt werden, ob diese dort verändert im Cache vorliegt. Diese wird dann entweder auf Platte geschrieben oder direkt zwischen den Nodes über das LAN ausgetauscht.

Beim Ausfall einer Node recovern die verbliebenden Nodes die Redo-Files des Ausgefallenen.

Vorteil: + fehlertolerant & Lastverteilung
Nachteil: - höhere Datenbank Lizenzkosten
- Administration ist aufwendiger

8.3. Standby-Datenbanken



Bei einer Standby-Datenbank wird eine Kopie (Online-Sicherung) der Produktions-Datenbank auf einem getrennten System installiert. Die Datenbank arbeitet im Recovery-Modus und verarbeitet kontinuierlich die Archiv-Logs der Produktion nach. Fällt die Produktions-DB aus, so werden auf der Standby-DB die restlichen Archiv-Logs nachverarbeitet und anschließend die DB normal gestartet.

Hält man zwischen Produktion- und Standby-System einen Zeitversatz (z. B. drei Stunden), so können administrative Fehler (z. B. DROP TABLE) aufgefangen werden.

Vorteil: + Es können administrative Fehler beseitigt werden.
+ Es steht schnell ein rückgespieltes Backup zur Verfügung.
Nachteil: - Hohe Hardwarekosten

Terminaldienst

Server: 141.72.5.187

SQL Talk

Connect dbvl2 <name> geheim; <STRG> +<Enter>

SELECT username, count(*) from v$session group by username;

Aufgabe 1:

- Tabellen anlegen

```
Create table student (matrnr number not null, name varchar2(80), vorname
varchar2(80));
```

- Unique Indizes auf den PK-Spalten anlegen

```
Create unique index ausleihe_pk on ausleihe (matrnr, buch);
```

- PK-Constraints auf den Tabellen anlegen

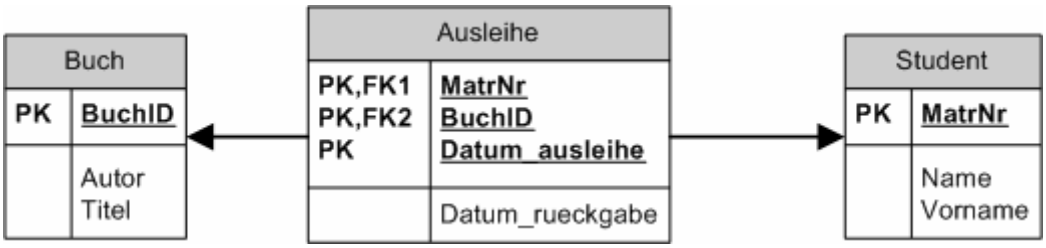
```
Alter table student add constraint student_pk primary key (matrnr);
```

- FK-Constraints auf den Tabellen anlegen

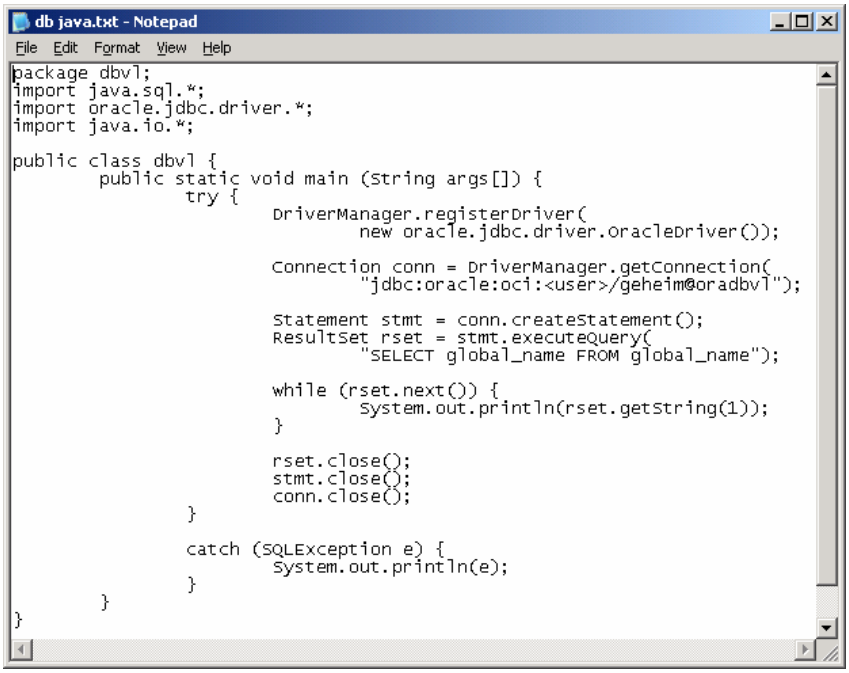
```
Alter table ausleihe add constraint ausleihe_student_fk foreign key
(matrnr) references student (matrnr);
```

- Sequenzen für inkrementellen Primary Key erzeugen:

```
Create sequence student_seq;
```



Aufgabe 4:



9. Stored Procedures & Trigger

$t_{Antwort} = t_{Netz} + t_{Abarbeitung} + t_{Netz}$

```
SELECT * FROM student
```

Schleife über Resultset

```
SELECT COUNT(*) FROM Ausleihe
WHERE MatrNr = ?
```

n Studenten:
$t_{Antwort} = (n+1) * (2 * t_{Netz} + t_{Abarbeitung})$
Normalfall LAN: $t_{Netz} << t_{Abarbeitung}$ → $t_{Antwort} = (n+1) * 2 * t_{Abarbeitung}$
t_{Netz} = 1000 ms, 100 Studenten →

$t_{Antwort}$ = 101 * 2 * 1s = 202 s

9.1. Stored Procedures

Stored Procedures sind Programme, die in der DB abgelegt werden. Sie können innerhalb von SQL benutzt werden. Je nach Hersteller stehen dafür verschiedene Sprachen zur Verfügung.

- eigene SQL Erweiterung (PLSQL)
- C++
- Java

```
Create or replace
Function Alter (dtGebDatum in date)
Return number
Is
Begin
      Return trunc ((sysdate-dtGebDate) / 365);
End;

Select name, vorname, Alter (GebDatum) from person;
```

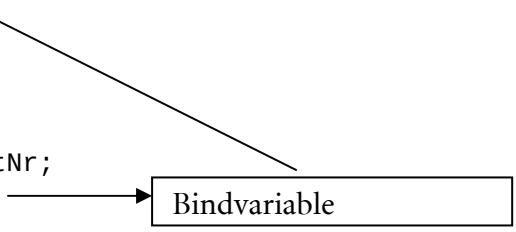
Vorteile:

- Es kann ein Abstraktionsniveau innerhalb der DB gebildet werden.
- Der Code wird auf der DB-Server ausgeführt. → Anforderung an Client sinkt
- Minimierung von Übertragungen über das Netzwerk → Zeitersparnis bei langsamen Netzwerken

```
dtMorgen := dtHeute + 1;
if dtMorgen < dtHeute then
      dbms_output.put_line (,kleiner');
else if dtMorgen = dtHeute then
      dbms_output.put_line ('gleich');
else
      dbms_output.put_line ('größer');
end if;

while (dtMorgen > dtHeute) loop
      dtHeute := dtHeute + 1;
end loop;

loop
      dtHeute := dtHeute -1;
      exit when dtHeute < dtMorgen;
end loop;
```

Die Funktion "Anzahl":

```
create or replace
function Anzahl (nMatNr in number)
return number
is
cursor hSqlAnzahl is
      SELECT COUNT (*)
      FROM Ausleihe
      WHERE MatNr = nMatNr;
nAnzahl number;

begin
      open hSqlAnzahl;
      fetch hSqlAnzahl into nAnzahl;
      close hSqlAnzahl;
      return nAnzahl;
end;
```



SELECT name, vorname, Anzahl (MatNr)
FROM student;

Übung

```
Function Zweiundvierzig
      Return number is
Begin
      Return 42;
End
```

9.2. Trigger

```
INSERT INTO person
(Name, Vorname, GebDatum)
VALUES
('Müller', 'Heinz', '1.1.1900');

CREATE OR REPLACE TRIGGER Person_Korrektur
Before insert on Person
For each row
Begin
      :new.alter := sysdate - :new.GebDatum;
End;
```

Trigger sind Programme, die durch die Änderung einer Row in einer Tabelle ausgeführt werden. Es ist nicht möglich, die Ausführung zu verhindern. Der Trigger kann vor der eigentlichen Änderung (dem Schreiben in die Tabelle) oder danach ausgeführt werden. Liegen mehrere Trigger auf einer Tabelle, so werden zuerst die Before-Trigger und anschließend die After-Trigger ausgeführt.

Innerhalb der beiden Ausführungsblöcke ist die Reihenfolge zufällig.

Anwendungsgebiete:

- Konsistente Befüllung abhängiger Spalten
- Zwangsbefüllung von DEBUG Spalten (siehe Übung)
- Befüllung von Performancetabellen
- Befüllung von Historietabellen
- Replikation

Übung mit Trigger

```
Create or replace trigger student_korrektur
Before insert on student
For each row
Begin
      :new.insdate := sysdate;
End;

INSERT INTO student
(matrnr, name, vorname)
VALUES (matrnr_seq.nextval, 'name', 'vorname') ;

Alter table … add (…)
```

- OOR DB
- **Tuning**
- Replikation 1 Vorlesungsblock
- Fragestunde
- Alte Klausur

Klausur: Dienstag in der Klausurwoche

10. Tuning

10.1. Hardware

Bei der Auswahl der Hardware auf
- ausreichend CPU, RAM
- Storagesystem
- Backup / Recovery achten

Wichtig:
- sinnvolle Dimensionierung
- Nachrüstung häufig schwierig

10.2. Datenbankparameter

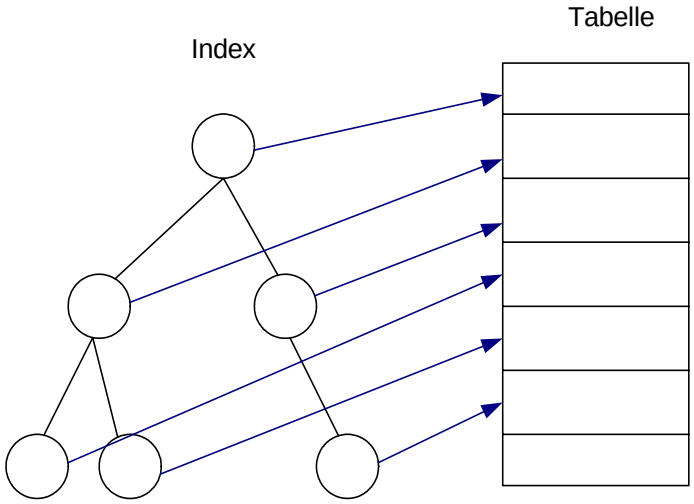
- Größe diverser Caches
- Anzahl genutzte CPUs
- Anzahl von Backendprozessen

10.3. Datenbankdateien

- RAID ?
- Striping?
- Eventuell Lastverteilung (beide TS auf getrennten Platten)
  - Tablespace Tabellen
  - Tablespace Indizes
- Filesystem oder RAW Device

10.4. Indizes

Suche in n Elementen ohne Heuristik O(n) = n
Bei einer Suche in einem B* Baum beträgt der Aufwand O(n) = log n
Ein Index wird auf n Spalten einer Tabelle definiert. Alle Werte aus den n Spalten der Tabelle werden in einem sortierten Baum als eigenes Speicherkonstrukt abgelegt. Zu jeder Kombination der Spalten werden Pointer auf die Rows der Tabelle abgelegt.



Jede Veränderung in der der Tabelle (Inserts, Updates, Deletes) werden online im Index gepflegt. Beim Zugriff auf den Index werden so n Pages aus dem Index angelesen. Anschließend werden über die gefundenen Pointer die eigentlichen Rows aus der Tabelle gelesen (in Pages).

Für die Anlage von Indizes gilt:

- nur Spalten indizieren, bei denen die Restmenge durch die Abfragen besser als 20 % ist. (Selektivität)
- Stehen mehrere Spalten als Indexkandidat zur Verfügung, so sollte die Spalte mit der höchsten Selektivität genutzt werden.

Übung:

Tabelle Paket

Barcode
PLZ
Gewicht

```
SELECT *
FROM Paket
WHERE PLZ between ‚10000' and ‚19999'
AND Gewicht between 1 and 2;
```

- Alle PLZ-Gebiete haben gleiches Volumen
- Es werden Pakete zwischen ein und zwanzig Kilo transportiert. Die Gewichte sind gleich verteilt.

Frage:

- Welche Spalten sind Indexkandidaten?
- Auf welche Spalten legen Sie den Index?

PLZ → 10 % Teilresultset
Gewicht → 5 % Teilresultset

Index auf Gewicht wegen der hohen Selektivität.

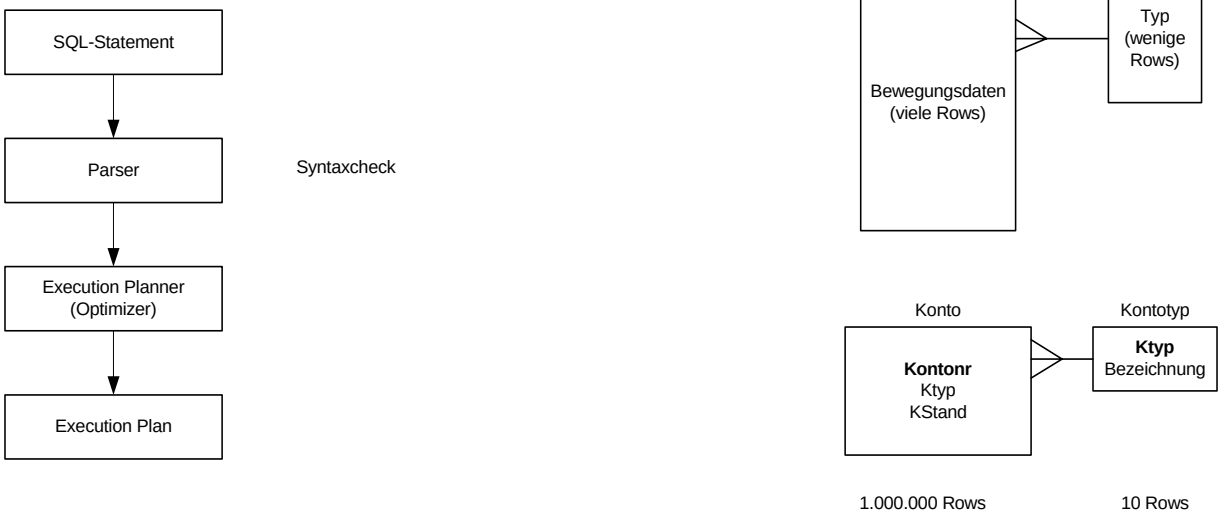
Vorteile von Indizes

- Performanceverbesserung beim Erstellen des Resultset
- Eventuell Verringerung des Sortierungsaufwandes

Nachteile

- DMLs werden langsamer (Data Manipulation Language)
- Indizes benötigen Platz
- Gefahr von „Killerindizes“
  - schlechte Selektivität (20% Teilresultset)
  - Indizes auf kleine Tabellen, Beispiel:
    Tabelle: 1 Page
    Index: 1 Page
    “Table access full : 1 Page
    “Indexzugriff: 2 Pages
  - Indizes mit großer Spaltenanzahl
    Jede Spalte muss das 20%-Kriterium erfüllen.

10.5. Application Layer




```
SELECT      *
FROM        Konto, Kontotyp
WHERE       Konto.KTyp = Kontotyp.KTyp
AND         KStand < 0
AND         Bezeichnung = ‘Giro’
```

Indizes:

- alle Pks
- alle Fks
- KStand

Zwei mögliche Ausführungspläne

- (1) Index KStand auswerten
  (2) über Tabellenzugriff auf Konto den KTyp bestimmen
  (3) In Kontotyp die Nicht-Girokonten filtern.

- (1) Alle Kontotypen ‚Giro' bestimmen
  (2) Über Index auf KTyp in Konto die passenden Rows suchen
  (3) In Tabelle Konto den KStand filtern

Es gibt zwei Arten von Optimizern:

- Regelbasierte Optimizer
- Kostenbasierte Optimizer

Regelbasierte Optimizer entscheiden anhand fester Regeln und vorhandener Indizes über den Ausführungsplan.
Kostenbasierte Optimizer benutzen zusätzlich Informationen über die tatsächliche Befüllung der Tabellen. Die Statistikinformation wird regelmäßig in so genannten Analyze-Läufen bestimmt.

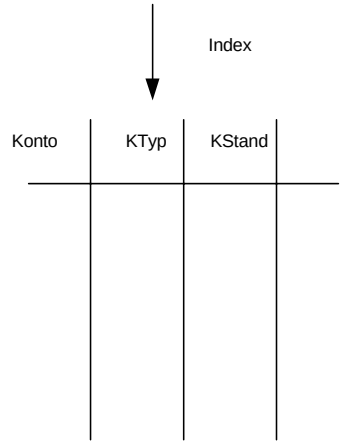
Vorteil kostenbasierte Optimizer:
- schnellere Ausführungszeit

Nachteil:
- regelmäßiger Analyzelauf belastet stark das Plattensystem
- Sind die statistischen Daten zu alt, entscheidet der Optimizer falsch.

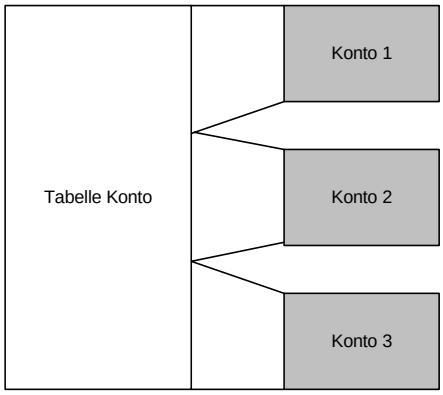
Bei der Software-Entwicklung sollte jeder Execution-Plan durch den Entwickler angesehen und bewertet werden. Ist der Execution-Plan zu schlecht, so kann dieser verändert werden:
- eventuell Index anlegen
- Statement umformulieren
- Statement splitten oder in ein Statement zusammenführen.
- Hint Beispiel: `SELECT /*+RULE*/...` bzw. `SELECT /*+COST*/...`

Alle Maßnahmen, die den Ausführungsplan beeinflussen, sind herstellerspezifisch! → Handbuch lesen.

10.6. Partitionierung



Vertikales Kriterium



Horizontales Kriterium

Bei der Partitionierung einer Tabelle wird diese anhand eines Kriteriums (z. B. das Datum) intern in n Teiltabellen aufgespaltet. Wird auf die Tabelle zugegriffen, so verhält sich diese nach außen wie eine normale Tabelle. Ist allerdings das Partitionierungskriterium in der Abfrage enthalten, so kann die Suche auf wenige Partitionen der Tabelle reduziert werden.

Administrative Maßnahmen (Analyze oder Indexaufbau) können partitionsweise ausgeführt werden. Das Reorganisieren kann bei geeignetem Kriterium auf ALTER TABLE DROP PARTITION reduziert werden.
Nachteile:

- Administrativer Aufwand für Neuanlagen von Partitionen
- Es muss ein sinnvolles Kriterium existieren, das von vielen Anfragen genutzt wird.

10.7. Locking

Um zu verhindern, dass bei der Überprüfung von Constraints ganze Tabellen gelockt werden, ist es ratsam, auf alle PK-Spalten einen unique Index anzulegen. Auf alle FK-Spalten sollte ein non-unique Index angelegt werden.

10.8. Performance-Tabellen

→ Verletzung der Normalformen, um ein häufig benötigtes Ergebnis als einfach zu benutzende Tabelle zur Verfügung zu haben.

Nachteil:

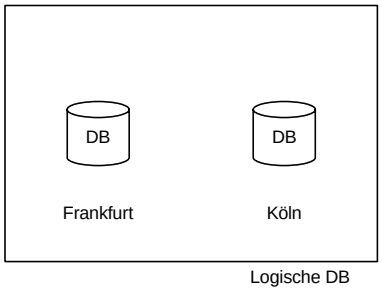
- Dateninkonsistenz zwischen Originaltabelle und Performancetabelle ist möglich
- Platzbedarf
- Abbildungsprozess wird benötigt

Die Befüllung der Performancetabelle kann über

- den Application Layer
- Trigger / Stored Procedures
- Snapshots / Materialized Views

erfolgen.

11. Verteilte Systeme



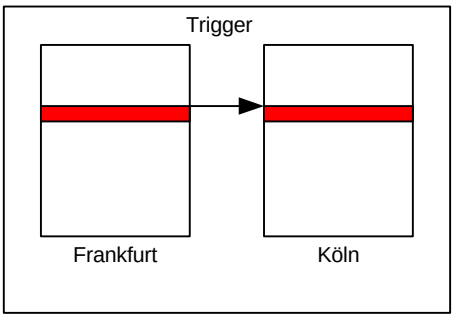
Verteilte DB: Logische Datenbank, deren Daten in mehreren physikalischen DBs liegen.

Database Link: Definition eines Zugriffweges auf eine entfernte Datenbank innerhalb einer Datenbank.

```
SELECT      *
FROM        Umsatz
UNION
SELECT      *
FROM        Umsatz@Frankfurt;
```

Durch den Database Link werden alle Daten aus der entfernten Datenbank innerhalb von SQL zugreifbar. Aus technischen Gründen wird ein Teil der Daten auf mehreren Knoten der verteilten Datenbank vorgehalten werden. Es werden Abgleichprozesse benötigt, die diese Tabellen zueinander konsistent halten. → **Replikation**

11.1. Synchrone Replikation




```
CREATE OR REPLACE TRIGGER
Kunde_nach_Koeln
AFTER INSERT ON Kunde
for each row
begin
      INSERT INTO Kunde@Koeln
      (…)
      VALUES
      (:new …);
end;
```

Bei synchroner Replikation wird jede Veränderung der Tabelle sofort auf alle Kopietabellen weitergegeben.

Vorteil:

- Keine Inkonsistenz zwischen den zu replizierenden Tabellen.
- Einfach zu implementieren

Nachteil:

- Bei der DML Operation müssen alle Zielsysteme erreichbar sein.
- Das Verändern der Daten auf der Originaltabelle wird sehr langsam

11.2. Asynchrone Replikation

Bei asynchroner Replikation wird zunächst in der Quelldatenbank nur die Veränderung der Tabelle in einer Queue vermerkt. Zu einem späteren Zeitpunkt werden die Veränderungen in der Queue auf allen Zielsystemen nachgezogen.

Vorteil:

- Die DML Operation findet nur im Quellsystem statt. Die Zielsysteme müssen nur während dem Datenabgleich erreichbar sein.

Nachteil:

- Zwischen zwei Abgleichen sind die Datenbanken zueinander inkonsistent.
- Implementierung ist komplexer

Der Replikationskonflikt: Wird zwischen zwei Abgleichprozessen eine Row auf zwei Datenbanken geändert, so muss entschieden werden, welche der beiden Operationen korrekt ist.

Frankfurt

ArtNr	Bezeichnung	Preis
1	Nix	1,00
2	Garnix	10,00

Köln

ArtNr	Bezeichnung	Preis
1	Nix	1,00
2	Garnix	10,00

10:00 Uhr	Abgleichprozess: Beide Tabellen haben identische Daten
10:01 Uhr	Frankfurt: UPDATE Artikel SET Preis = 2 * Preis;
10:02 Uhr	Köln: UPDATE Artikel SET Preis = 100 WHERE ArtNr = 1;
10:10 Uhr	Abgleichprozess

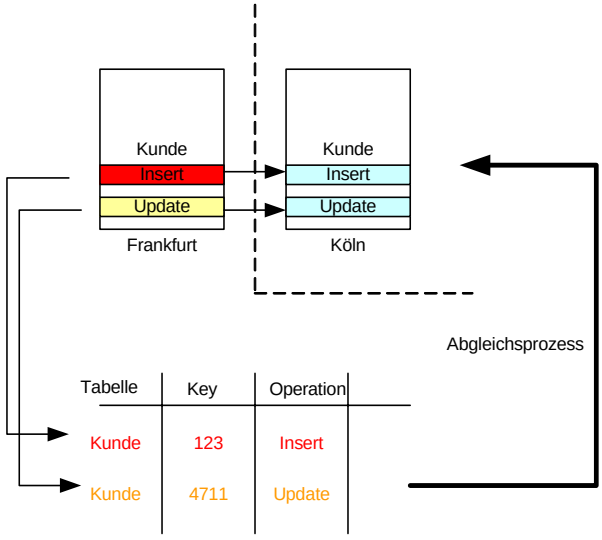
Vermeidung von Replikationskonflikten

- Pflege der Daten findet nur auf einer Seite statt
- Minimierung von DELETEs und UPDATEs durch ein für die Replikation angepasstes Datenmodell

Lösung von Replikationskonflikten

- Letzte Transaktion gewinnt
- Priorisierung von einer Datenbank („Frankfurt hat immer Recht“)
- Funktionale Entscheidung (Max, Min, Avg)
- Eigene Stored Procedure
- Visualisierung und manuelle Korrektur
- …

11.2.1. Snapshotreplikation



Bei der Snapshotreplikation wird jede Veränderung in der Tabelle in einer einfachen Queue abgelegt. Dabei werden nur der Key und die Art der Veränderung gespeichert. Die eigentlichen Daten werden nicht gespeichert. Wird eine Row zwischen zwei Abgleichprozessen mehrfach geändert, so wird nur die höchstpriorisierteste Operation gespeichert.

Beispiel:
INSERT, UPDATE → INSERT in Queue
UPDATE, DELETE → DELETE in Queue

Der Abgleichprozess zieht dann die Zieltabelle anhand der Daten in der Originaltabelle auf dem Zielsystem nach.

Nachteil:

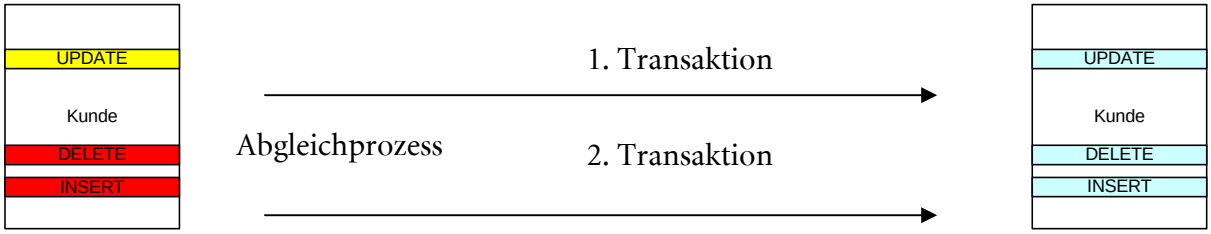
- Es wird nur der Endzustand der Daten transportiert.
- Replikationskonflikte können nur in wenigen Fällen erkannt werden.

Die klassische Snapshotreplikation ist eine Sternverteilung, d. h. nur in einem System können die Daten geändert werden.

Vorteile:

- asynchrones Verfahren
- einfache Implementierung
- einfache Administration

11.2.2. Multi-Master Replikation



Operation	Reihenfolge	Transaktion	New_Kdnr	Old_KdNr	New_Name	Old_Name
INSERT	1	1	123	NULL	Nase	NULL
DELETE	2	1	NULL	4711	NULL	Bär
UPDATE	1	2	456	456	Ketchup	Heinz

Bei der Multimaster-Replikation werden in den Queue-Tabellen Informationen über die Transaktion, die Reihenfolge der Statements innerhalb der Transaktion, die Operation und für alle Felder der Tabelle die Inhalte der Row vor und nach der Operation gespeichert. Der Abbauprozess führt die so gespeicherten Transaktionen 1:1 auf der Zielseite aus. Dabei werden im Gegensatz zur Snapshotreplikation alle Zwischenstände auch im Zielsystem erreicht.

Mit den gespeicherten alten Werten der Spalten lassen sich Replikationskonflikte erkennen.

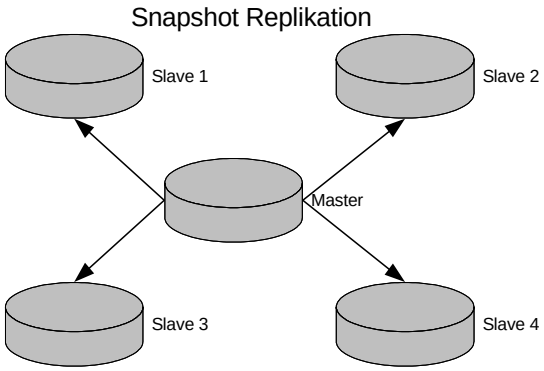
Vorteile:

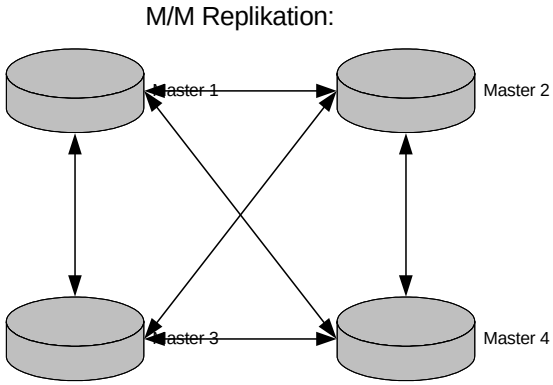
- auf allen Seiten können Daten verändert werden
- Datenaustausch in der Regel Pushverfahren, bei Snapshot in der Regel Poll-Verfahren. Push ist schneller
- Replikationskonflikte können erkannt werden

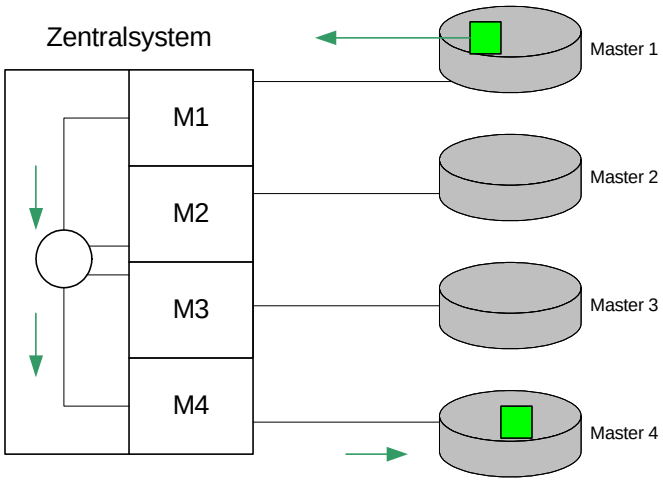
Nachteile:

- Implementierung ist komplex
- Administrationsaufwand → Änderung nur im Wartungsfenster möglich

MM-Replikation wird in der Regel bei Bewegungsdaten eingesetzt. Es werden hier n*(n-1) Prozesse benötigt. Bei der Snapshot-Replikation nur (n-1) Prozesse.





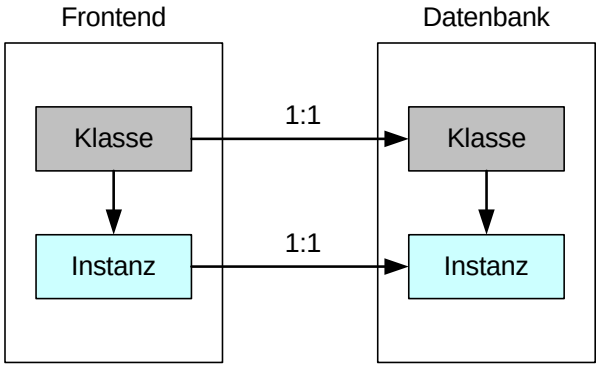


Anwendung z. B. DataWarehouse

12. Objektorientierte Systeme

- Vererbung
- Polymorphie
- Kapselung
- Keine logische Trennung zwischen Daten & Methoden

12.5. Voll objektorientierte Datenbank



Funktionsweise:

- Objekt aus dem Frontend wird 1:1 in der DB gespeichert
- Wegen Code-Inkompatibilitäten kann häufig keine Methode zur Objektsuche in der Datenbank eingesetzt werden.

12.6. Objektrelationale Datenbank

Idee: Relationale DB um objektorientierte Ansätze erweitern

SQL wird um objektorientierte Eigenschaften erweitert. Klassen werden als „Type“ definiert.

```
type  tPerson is
      (Name       varchar2(80)          } Klassische Datenfelder
      Vorname     varchar2(80)          }
      GebDatum    date                  }
      method      alter return number   } Methoden
                  is                    }
                  begin                 }
                  …                     }
                  end;                  }
      );
```

```
CREATE TABLE Person
object of tPerson;
```

```
SELECT *
FROM Person
WHERE Alter between 18 and 25;
```

Die Methoden können in unterschiedlichen Sprachen definiert werden (z. B. C++, Java, PLSQL). Kompilierte Sprachen (C++) werden auf dem CPU-Typ der DB-Maschine kompiliert.

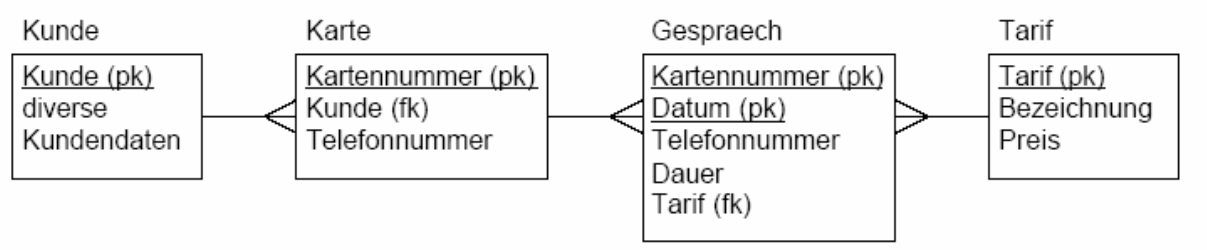
Vorteile:

- Backend ist objektorientiert

Nachteile:

- Objekt aus dem Backend kann nicht direkt auf das Frontend übertragen werden.
- Jeder Hersteller verfolgt eigenes Konzept

Lösung der Übungsklausur



a)

Scheiben Sie ein SQL-Statement das für die Kartennummer 487324563 einen Einzelgesprächsnachweis für den Monat November 2003 (Datum, angerufene Telefonnummer, Dauer, Verbindungspreis) ausgibt. Sortieren Sie das Ergebnis nach dem Datum.

```
SELECT      g.Datum,
            g.Telefonnummer,
            g.Dauer,
            g.Dauer * t.Preis
FROM        Gespraech g,
            Tarif t
WHERE       g.Kartennummer = 487324563
AND         g.Datum BETWEEN ,1.11.2003 0:00:00' AND ,30.11.2003 23:59:59'
AND         g.Tarif = t.Tarif
ORDER BY    g.Datum;
```

b)
Schreiben Sie ein SQL-Statement, das für die Kundennummer 4711 die im Monat November 2003 angefallenen Kosten je Karte ausgibt (Kartennummer, Telefonnummer der Karte, Kosten).

```
SELECT      k.Kartennummer,
            k.Telefonnummer,
            SUM (g.Dauer * t.Preis)
FROM        Karte k,
            Gespraech g,
            Tarif t
WHERE       k.Kartennummer = g.Kartennummer
AND         g.Tarif = t.Tarif
AND         k.Kundennummer = ,4711'
AND         g.Datum BETWEEN ,1.11.2003 0:00:00' AND ,30.11.2003 23:59:59'
GROUP BY    k.Kartennummer, k.Telefonnummer
```

2.
a)
Ein Optimizer bestimmt den Ausführungsplan eines Statements.

b)

- Regelbasiert
- Kostenbasiert

c)
Regelbasierte Optimizer entscheiden anhand fester Regeln und vorhandener Indizes über den Ausführungsplan.
Kostenbasierte Optimizer benutzen zusätzlich Informationen über die tatsächliche Befüllung der Tabellen. Die Statistikinformation wird regelmäßig in so genannten Analyze-Läufen bestimmt.

d)
Regelbasiert: Keine besonderen Voraussetzungen
Kostenbasiert: Statistikdaten, die vom Analyse-Lauf generiert werden

3.
a)
Die Page wird beim INSERT nur bis zum Erreichen des durch den PCT_FREE Wertes begrenzten Kapazität gefüllt.

b)

$$\frac{95\% - 60\%}{2} + 60\% = 77{,}5\%$$

c)

$$\frac{29}{36} = 81\%$$

Alle Blöcke, die vor dem ALTER TABLE 81 % und mehr gefüllt waren, erhalten eine Chained Row. Zu Beginn der Tabelle ist das Risiko für Chained Rows sehr groß, da alle Pages um die 95 % gefüllt sind. Nach einem Jahr beträgt die mittlere Befüllung einer Page nur noch 77,5 % und damit ist das Risiko für Chained Rows wesentlich geringer.

5.

BS Cluster
FailOver, Prozesse wandern auf den Standby-Server

SchattenDB
Absicherung gegen administrative Fehler

6.

Siehe Script

7.

c)

```
create or replace trigger
tarif_to_queue
after insert or update or delete on tariff
for each row
declare
      strOperation varchar2(20);
      begin
      if inserting then
            strOperation := 'INSERT'
      else if updating then
            strOperation := 'UPDATE';
      else
            strOperation := 'DELETE';
      end if;

      INSERT INTO Queue
            (Transaktion, Statement, Operation, neu_tarif, old_tarif,
            neu_Bezeichnung, old_Bezeichnung, new_Preis, old_Preis)
      VALUES
            (GetTransaktion(), GetStatement(), strOperation, :new.Tarif,
            :old.Tarif, …)
end;
```

7.

a)

Nur die Spalten in der WHERE Klausel sind für die Abfrage interessant, also sind nur diese beiden Spalten für einen Index interessant.
Selektivität ist bei der Telefonnummer höher als bei Datum. Datum reduziert auf 1/6, Telefonnummer auf 1/10 → Index auf Telefonnummer

b)

Bei der Betrachtung aller Gespräche außerhalb Deutschlands reduziert sich das Resultset auf 20%, daher ist die Selektivität von Datum in diesem Fall höher → Index auf Datum