



## *Inhalt:*

- *Überblick*
- *Datentypen*
- *Einfache Befehle*
- *Erweiterungen*
- *Anfragen über mehrere Relationen*
- *Operator JOIN*
- *Anfragebearbeitung*
- *Indizes*



*DDL = Data Definition Language*

*DML = Data Manipulation Language*

*DCL = Data Control Language*

*TCL = Transaction Control Language*

## DDL

- ✓ ALTER
- ✓ COMMENT
- ✓ CREATE
- ✓ DROP
- ✓ RENAME
- ✓ TRUNCATE

## DML

- ✓ CALL
- ✓ DELETE
- ✓ EXPLAIN
- ✓ INSERT
- ✓ LOCK
- ✓ MERGE
- ✓ SELECT
- ✓ UPDATE

## DCL

- ✓ GRANT
- ✓ REVOKE

## TCL

- ✓ COMMIT
- ✓ ROLLBACK
- ✓ SAVEPOINT
- ✓ SET  
TRANSACTION



Ein Datentyp stellt die Werte und die darauf definierten Operatoren dar.

*Datentyp = Werte + Operationen*

*Beispiele:*

- *Reelle Zahlen (real) [ +, -, \*, / ]*
- *Ganze Zahlen (integer) [ +, -, \*, /, mod ]*
- *Text (string, text) [ substr, length, +, pad ]*
- *Datum, Uhrzeit (date, time) [ -, + ]*
- *Unstrukturierte Objekte (video, mem)*
- *Elektrischer Widerstand [  $R^+$ , Reihenschaltung, Parallelschaltung ]*



*Oracle und viele andere Datenbanken unterstützen folgende Datentypen:*

- *VARCHAR (n), VARCHAR2 (n) – Zeichenkette variabler Länge.*
- *CHAR (n) – Zeichenkette fester Länge.*
- *NUMBER (p, s) – Dezimale Zahl, p ist Anzahl von genauen Stellen,  $p=1..38$ , s ist Anzahl von Nachkommastellen,  $s=-84..127$ , Werte von  $-10^{125}$  bis  $+10^{125}$ .*
- *DECIMAL (p, s) – Dezimale Zahl, p ist Genauigkeit,  $p=1..38$ , s ist Anzahl von Nachkommastellen,  $s=-84..127$ , Werte von  $-10^{308}$  bis  $+10^{308}$ .*
- *INTEGER – Ganze Zahl,  $-2147483648..2147483647$ .*
- *DATE – Gültiges Datum/Uhrzeit (sekundengenau).*



- *RAW (n) – Binärdaten der Länge n, n zwischen 1 und 2000 Bytes.*
- *LONG RAW – Binärdaten bis zu 2 GiB.*
- *CLOB – Zeichenketten bis 4 GiB.*
- *BLOB – Binärdaten bis 4 GiB.*
- *CFILE, BFILE – Zeiger auf Dateien (Text, Binär).*



## Tabelle anlegen (DDL):

```
CREATE TABLE Professoren  
(  
    PersNr INTEGER NOT NULL,  
    Name    VARCHAR(30) NOT NULL,  
    Rang    CHARACTER(2),  
    PRIMARY KEY(PersNr)  -- is for integrity only!  
);
```

## Tabelle insgesamt löschen (DDL):

```
DROP TABLE Professoren;
```



## Tupel (Zeile) einfügen (DML):

```
INSERT INTO Professoren (PersNr, Name, Rang)
VALUES (30314, 'Cantor', 'W2');
```

```
INSERT INTO Professoren (PersNr, Name, Rang) VALUES
(30314, 'Cantor', 'W2'),
(30315, 'Leibniz', 'W1'),
(30316, 'Plank', 'W1');
```

## Einfügen mit einer verschachtelten Abfrage:

```
INSERT INTO hoeren
SELECT MatrNr, VorLNr
FROM Studenten, Vorlesungen
WHERE Titel = 'Datenbanken';
```

## Tupel teilweise einfügen (DML):

```
INSERT INTO Professoren (PersNr, Name)
VALUES (30317, 'Feuerbach'); -- Rang ist mit NULL belegt
```



*Inhalt der Tabelle löschen (DML):*

*DELETE FROM Professoren;*

*DELETE FROM Professoren WHERE Range = 'W3';*

*Tupel ändern (DML):*

*UPDATE Studenten SET Semester = Semester + 1;*



*Inline-View ist eigentlich kein View. Das ist eine SELECT-Anweisung, die eine weitere SELECT-Anweisung in der Klausel FROM hat.*

*Professoren, die Assistenten haben:*

```
SELECT p.Name, p.Raum
FROM Professoren p,
(
    SELECT DISTINCT Boss
    FROM Assistenten
) a
WHERE p.PersNr = a.Boss;
```

```
SELECT p.Name, p.Raum
FROM Professoren p, Assistenten a
WHERE p.PersNr = a.Boss;
```



*Noch eine Inline-View-Anweisung.*

*Studenten, die Vorlesung "Ethik" besuchen:*

```
SELECT s.Name, s.Semester
FROM Studenten s,
(
  SELECT h.MatrNr
  FROM hoeren h,
  (
    SELECT VorLNr
    FROM Vorlesungen
    WHERE Titel = 'Ethik'
  ) v
  WHERE h.VorLNr = v.VorLNr
) a
WHERE s.MatrNr = a.MatrNr
;
```



*Allgemeine Syntax des Befehls SELECT:*

```
SELECT column1, column2  
  FROM table1, table2  
  WHERE condition  
  GROUP BY column1, column2  
  HAVING condition  
  ORDER BY column1, column2;
```

```
SELECT column1 [AS] alias_col_1, column2 [AS] alias_col_2  
  FROM table1 [AS] alias_tab_1, table2 [AS] alias_tab_2  
  WHERE condition  
  GROUP BY column1, column2  
  HAVING condition  
  ORDER BY column1, column2;
```

```
SELECT * FROM Dual;
```



*SQL ist durch mehrere Komponenten erweitert, die in der relationalen Algebra nicht vorhanden sind.*

*Sortieren der Ausgabe:*

```
SELECT PersNr, Name, Rang FROM Professoren  
ORDER BY Rang DESC, Name ASC; – – absteigend, aufsteigend
```

*Duplikate bei der Ausgabe eliminieren (zählt das ganze Tupel):*

```
SELECT DISTINCT Rang FROM Professoren;
```

*Vergleiche mit den Platzhaltern (nur mit LIKE zu verwenden):*

- \_ steht für genau ein Zeichen;*
- % steht für beliebige Anzahl der Zeichen (auch für keine).*

```
SELECT * FROM Professoren WHERE Rang LIKE 'W_';
```

```
SELECT * FROM Professoren WHERE Name LIKE 'T%eophrastos';
```



*Standard-Funktionen IN, NOT IN.*

*Anfrage: Welche Professoren lesen keine Vorlesungen.*

```
SELECT Name  
FROM Studenten  
WHERE Semester IN (1, 2, 3);
```

```
SELECT VorlNr  
FROM Vorlesungen  
WHERE Titel IN ('Ethik', 'Logik');
```

```
SELECT Name  
FROM Professoren  
WHERE PersNr NOT IN  
    (SELECT gelesenVon FROM Vorlesungen);
```



## Standard-Funktionen ALL, ANY (SAME).

```
SELECT Name FROM Studenten  
WHERE Semester >= ALL (7, 8, 9);
```

– – the same as

```
SELECT Name FROM Studenten  
WHERE Semester >= 7 AND Semester >= 8 AND Semester >= 9;
```

```
SELECT Name FROM Studenten  
WHERE Semester = ALL (SELECT Semester FROM Studenten);
```

```
SELECT Name FROM Studenten  
WHERE Semester >= ANY (7, 8, 9);
```

– – the same as

```
SELECT Name FROM Studenten  
WHERE Semester >= 7 OR Semester >= 8 OR Semester >= 9;
```



*Oft muss man Informationen aus mehreren Tabellen auswerten. Solche Anfragen über mehrere Relationen (Tabellen) heißen Joins.*

*Vorgehensweise dafür in allen relationalen Datenbanken ist gleich:*

- Kreuzprodukt aus Tabellen bilden.*
- Notwendige Zeilen und Felder aus dem Kreuzprodukt ausschneiden.*

*Wichtigste Voraussetzung dabei ist die Information über Verbindung zwischen den Tabellen – wie, durch welche Felder sind die Tabellen untereinander verbunden.*

<i>SELECT t1.Feld1, t2.Feld2</i>	<i>– – nicht obligatorisch</i>
<i>FROM Tab1 t1, Tab2 t2</i>	<i>– – Kreuzprodukt</i>
<i>WHERE t1.Feld1 = t2.Feld2;</i>	<i>– – Verbindung zwischen den</i>
	<i>– – Tabellen</i>
	<i>– – access- oder join-Prädikat</i>



*Anfrage: Welche Professoren haben welche Assistenten.*

*Beide Tabellen sind hier in voller Größe dargestellt:*

PersNr	Name	Rang	Raum
-----	-----	-----	-----
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

PersNr	Name	Fachgebiet	Boss
-----	-----	-----	-----
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2134

*SELECT \**  
*FROM Professoren, Assistenten;*



*Kreuzprodukt (Auszug), ohne die zusammengehörigen Zeilen auszuschneiden, sieht so aus:*

PersNr	Name	Rang	Raum	PersNr	Name	Fachgebiet	Boss
-----	-----	----	----	-----	-----	-----	----
2125	Sokrates	C4	226	3002	Platon	Ideenlehre	2125
2125	Sokrates	C4	226	3003	Aristoteles	Syllogistik	2125
2125	Sokrates	C4	226	3004	Wittgenstein	Sprachtheorie	2126
2125	Sokrates	C4	226	3005	Rhetikus	Planetenbewegung	2127
2125	Sokrates	C4	226	3006	Newton	Keplersche Gesetze	2127
2125	Sokrates	C4	226	3007	Spinoza	Gott und Natur	2134
2126	Russel	C4	232	3002	Platon	Ideenlehre	2125
2126	Russel	C4	232	3003	Aristoteles	Syllogistik	2125
2126	Russel	C4	232	3004	Wittgenstein	Sprachtheorie	2126
2126	Russel	C4	232	3005	Rhetikus	Planetenbewegung	2127
2126	Russel	C4	232	3006	Newton	Keplersche Gesetze	2127
2126	Russel	C4	232	3007	Spinoza	Gott und Natur	2134
2127	Kopernikus	C3	310	3002	Platon	Ideenlehre	2125
2127	Kopernikus	C3	310	3003	Aristoteles	Syllogistik	2125
2127	Kopernikus	C3	310	3004	Wittgenstein	Sprachtheorie	2126
2127	Kopernikus	C3	310	3005	Rhetikus	Planetenbewegung	2127
2127	Kopernikus	C3	310	3006	Newton	Keplersche Gesetze	2127
2127	Kopernikus	C3	310	3007	Spinoza	Gott und Natur	2134
2133	Popper	C3	52	3002	Platon	Ideenlehre	2125
2133	Popper	C3	52	3003	Aristoteles	Syllogistik	2125
2133	Popper	C3	52	3004	Wittgenstein	Sprachtheorie	2126
2133	Popper	C3	52	3005	Rhetikus	Planetenbewegung	2127
2133	Popper	C3	52	3006	Newton	Keplersche Gesetze	2127
2133	Popper	C3	52	3007	Spinoza	Gott und Natur	2134



*Nach dem die zusammengehörigen Zeilen ausgeschnitten sind, sieht Ergebnis so aus:*

PersNr	Name	Rang	Raum	PersNr	Name	Fachgebiet	Boss
-----	-----	----	----	-----	-----	-----	----
2125	Sokrates	C4	226	3002	Platon	Ideenlehre	2125
2125	Sokrates	C4	226	3003	Aristoteles	Syllogistik	2125
2126	Russel	C4	232	3004	Wittgenstein	Sprachtheorie	2126
2127	Kopernikus	C3	310	3005	Rhetikus	Planetenbewegung	2127
2127	Kopernikus	C3	310	3006	Newton	Keplersche Gesetze	2127
2134	Augustinus	C3	309	3007	Spinoza	Gott und Natur	2134

```
SELECT *  
FROM Professoren p, Assistenten a  
WHERE p.PersNr = a.Boss;  -- join-Prädikat  
  
-- Warum sind hier die Aliasse wichtig?
```



*Jetzt kann man hier noch einen vertikalen Ausschnitt machen:*

Professor	Assistent
-----	-----
Sokrates	Platon
Sokrates	Aristoteles
Russel	Wittgenstein
Kopernikus	Rhetikus
Kopernikus	Newton
Augustinus	Spinoza

```
SELECT p.Name Professor, a.Name Assistent
FROM Professoren p, Assistenten a
WHERE p.PersNr = a.Boss;
```

*Aliasse muss man nicht verwenden, stattdessen kann man die Tabellennamen einsetzen, falls die Felder mit gleichen Namen aus unterschiedlichen Tabellen in einer Anfrage involviert sind.*

*Aliasse sind sehr praktikabel.*



*Beispiele für Anfragen:*

*Welche Studenten hören welche Vorlesungen.*

*Welche Vorlesungen haben keine Nachfolger.*

*Welche Professoren lesen welche Vorlesungen.*

*Studenten, ihre Noten in Vorlesungen und entsprechende Professoren auflisten.*



*Bei Abfragen über mehrere Tabellen stehen die join- und filter-Prädikate zusammen in der WHERE-Klausel.*

```
SELECT p.Name Professor, a.Name Assistent
FROM Professoren p, Assistenten a
WHERE p.PersNr = a.Boss      – – join-Prädikat
      AND p.Name = 'Russel'; – – filter-Prädikat
```

*Dem Optimizer ist es schwer zwischen diesen zwei Typen von Prädikaten zu unterscheiden. Wäre es möglich, dann würde man zuerst die filter-Prädikate an entsprechende Tabellen anwenden, um diese kleiner zu machen, und nur danach würde man das kartesische Produkt aus relativ kleineren Tabellen bilden.*

*Man hat den JOIN-Operator (JOIN-Klausel) eingeführt, um solche Entscheidungen zu erleichtern: in dem JOIN-Operator schreibt man das join-Prädikat und in der WHERE-Klausel schreibt man das filter-Prädikat.*



*Relationaler Verbund (Operator JOIN) ist ein kartesisches Produkt, bei dem die Auswahl der relevanten Tupel sofort in der Operation vorgenommen wird. Technisch gesehen, macht es enorme Zeit-Ersparnisse.*

*Man unterscheidet innere und äußere JOIN-Operatoren. In beiden Fällen werden die Tupel aus einer Relation mit den Tupeln aus der zweiten Relation nach bestimmten Regeln und Bedingungen verknüpft. Unterschied ist nur in der Behandlung der Tupel, für die keine entsprechenden Tupel aus der anderen Relation vorhanden sind.*

*Außerdem gibt es noch die Semi-JOIN-Operatoren.*

*Alle JOIN-Operatoren sind binär, d.h. sie verbinden zwei Relationen.*



*Bei den inneren JOIN-Operatoren werden die Tupel, für die keine entsprechenden Partner-Tupel gefunden wurden, nicht in das Ergebnis übernommen, also – sie gehen einfach verloren.*

*Es gibt folgende Arten von inneren JOIN-Operatoren:*

- *Natürlicher Verbund.*
- *Allgemeiner Verbund oder Theta-JOIN.*

*Man kann den natürlichen Verbund als eine Spezialisierung des allgemeinen Verbundes betrachten.*



*Damit diese Art des Verbundes verwendet werden kann, müssen die beiden Relationen die gleich benannten Attribute haben, wobei diese Attribute den selben Datentyp aufweisen müssen. Das sind die JOIN-Attributen.*

*In das Ergebnis werden die Tupel übernommen, die gleiche Werte in den JOIN-Attributen haben. Die JOIN-Attribute selbst werden in das Ergebnis nur einmal übernommen.*

*Der natürliche Verbund wird so bezeichnet:*

$$R \bowtie S$$



## Der natürliche Verbund (Join)

Gegeben seien zwei Relationen:

$R(A_1, \dots, A_m, B_1, \dots, B_k)$

$S(B_1, \dots, B_k, C_1, \dots, C_n)$

$$R \bowtie S = \Pi_{A_1, \dots, A_m, R.B_1, \dots, R.B_k, C_1, \dots, C_n}(\sigma_{R.B_1=S.B_1 \wedge \dots \wedge R.B_k=S.B_k}(R \times S))$$

$R \bowtie S$											
$R - S$				$R \cap S$				$S - R$			
$A_1$	$A_2$	...	$A_m$	$B_1$	$B_2$	...	$B_k$	$C_1$	$C_2$	...	$C_n$



## Beispiel Join

hören	
MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
28106	5043

Vorlesungen			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125

hören ⋈ Vorlesungen

hören		Vorlesungen			
MatrNr	VorlNr	VorlNr	Titel	SWS	gelesenVon
26120	5001	↔ 5001	Grundzüge	4	2137
27550	5001	↔ 5001	Grundzüge	4	2137
28106	5043	↔ 5043	Erkenntnistheorie	3	2126

Einmal wird die Spalte VorlNr in der Relationalen Algebra eliminiert



## 3-Wege-Join

(Studenten ⋈ hören) ⋈ Vorlesungen							
MatrNr	Name	Semester	VorlNr	Titel	SWS	Gelesen Von → PersNr	...
26120	Fichte	10	5001	Grundzüge	4	2137	...
27550	Jonas	12	5022	Glaube und Wissen	2	2134	...
28106	Carnap	3	4052	Wissenschaftstheorie	3	2126	...
...	...	...	...	...	...	...	...

Wenn Attribute mit unterschiedlichen Name verknüpft werden sollen, müssen diese umbenannt werden (z.B. PersNr gelesenVon wenn die lesenden Professoren verknüpft werden sollen)

Studenten ⋈ Vorlesungen ⋈  $\rho_{\text{gelesenVon} \leftarrow \text{PersNr}}(\text{Professoren})$



*Diese Art des Verbundes kann für die Relationen mit beliebigen Attributen verwendet werden. In das Ergebnis werden die Tupel übernommen, die ein Prädikat (Bedingung) erfüllen. Keine Attribute werden aus dem Ergebnis eliminiert.*

*Der allgemeine Verbund wird so bezeichnet:*

$$R \bowtie_{\theta} S$$

*Hier ist  $\theta$  ein Prädikat, z.B.*

$$R \bowtie_{A1 > B1 \wedge A2 = B2 \vee A3 < B3} S$$



## Allgemeiner Join (Theta-Join)

Gegeben seien folgende Relationen(-Schemata)

- $R(A_1, \dots, A_n)$  und
- $S(B_1, \dots, B_m)$

$$R \bowtie_{\theta} S = \sigma_{\theta} (R \times S)$$

$$R \bowtie_{\theta} S$$

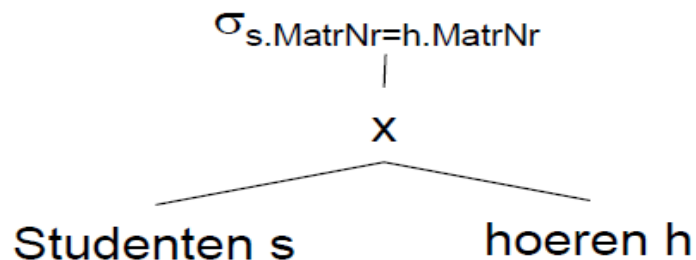
Angabe eines  
beliebigen Join-  
Prädikats  $\theta$

$R \bowtie_{\theta} S$							
R				S			
$A_1$	$A_2$	...	$A_n$	$B_1$	$B_2$	...	$B_m$

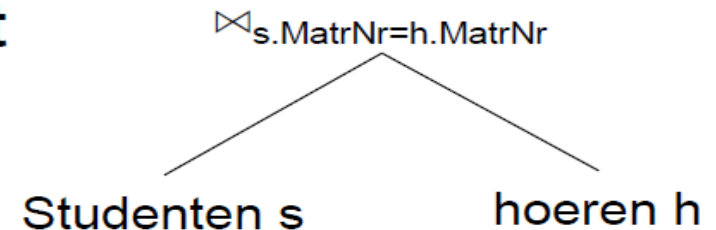
Beispiel: Assistenten, die mehr verdienen als die zugeordneten Profs  
 Professoren  $\bowtie_{\text{Professoren.Gehalt} < \text{Assistenten.Gehalt} \wedge \text{Boss} = \text{Professoren.PersNr}}$  Assistenten



## Vergleich Kartesisches Produkt/Join



**entspricht**



- Der Join Operator entspricht einem Kartesischen Produkt inklusive einer Selektion
- Join-Operatoren, die auf Gleichheit von Attributen selektieren wird Equi-Join genannt
- Join-Operatoren mit beliebigem Selektionsprädikat wird Theta-Join genannt



*Bei den äußeren JOIN-Operatoren werden die Tupel, für die keine entsprechenden Partner-Tupel gefunden wurden, mit den NULL-Werten für fehlendes Partner-Tupel ergänzt und in das Ergebnis übernommen, also, sie gehen nicht verloren.*

*Es gibt folgende Arten von äußeren JOIN-Operatoren:*

- *Linker äußerer Verbund;*
- *Rechter äußerer Verbund;*
- *Vollständiger äußerer Verbund.*



*Der linke äußere JOIN funktioniert ähnlich wie der natürliche Verbund: die beiden Relationen müssen gleich benannte Attribute von den gleichen Datentypen haben.*

*Bei dem linken äußeren JOIN-Operator werden alle Tupel aus der linken Relation in das Ergebnis übernommen. Dabei die Tupel, für die keine entsprechenden Partner-Tupel in der rechten Relation gefunden wurden, werden mit den NULL-Werten (rechts) ergänzt und in das Ergebnis übernommen.*

*Der linke äußere Verbund wird so bezeichnet:*

$$L = \bowtie R$$

*Relation L*

A	B	C
a1	b1	c1
a2	b2	c2

*Relation R*

C	D	E
c1	d1	e1
c3	d3	e3

*Relation L = ⋈ R*

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b2	c2	NULL	NULL



*Der rechte äußere JOIN funktioniert ähnlich wie der natürliche Verbund: die beiden Relationen müssen gleich benannte Attribute von den gleichen Datentypen haben.*

*Bei dem rechten äußeren JOIN-Operator werden alle Tupel aus der rechten Relation in das Ergebnis übernommen. Dabei die Tupel, für die keine entsprechenden Partner-Tupel in der linken Relation gefunden wurden, werden mit den NULL-Werten (links) ergänzt und in das Ergebnis übernommen.*

*Der rechte äußere Verbund wird so bezeichnet:*

$$L \bowtie = R$$

*Relation L*

A	B	C
a1	b1	c1
a2	b2	c2

*Relation R*

C	D	E
c1	d1	e1
c3	d3	e3

*Relation L ⋈= R*

A	B	C	D	E
a1	b1	c1	d1	e1
NULL	NULL	c3	d3	e3



*Der vollständige äußere JOIN funktioniert ähnlich wie der natürliche Verbund: die beiden Relationen müssen gleich benannte Attribute von den gleichen Datentypen haben.*

*Bei dem vollständigen äußeren JOIN-Operator werden alle Tupel aus beiden Relation in das Ergebnis übernommen. Dabei die Tupel, für die keine entsprechenden Partner-Tupel in der linken oder in der rechten Relation gefunden wurden, werden mit den NULL-Werten (links oder rechts) ergänzt und in das Ergebnis übernommen.*

*Der vollständige äußere Verbund wird so bezeichnet:*

$$L = \bowtie = R$$



Relation L

A	B	C
a1	b1	c1
a2	b2	c2

Relation R

C	D	E
c1	d1	e1
c3	d3	e3

Relation L  $= \bowtie =$  R

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b2	c2	NULL	NULL
NULL	NULL	c3	d3	e3



## Semi-JOIN-Operatoren

*Die Semi-JOIN-Operatoren liefern als Ergebnis die Tupel nur aus einer der beiden Relationen. Sie funktionieren ähnlich wie der natürliche Verbund: die beiden Relationen müssen gleich benannte Attribute von den gleichen Datentypen haben.*

*Es gibt folgenden Operatoren:*

- *Semi-JOIN von L mit R*
- *Semi-JOIN von R mit L*
- *Anti-Semi-JOIN von L mit R*
- *Anti-Semi-JOIN von R mit L*



*Dieser Operator liefert nur die Tupel aus der linken Relation, die die entsprechenden Tupel-Partner aus der rechten Relation haben.*

*Dieser Operator wird so bezeichnet:*

$$L \bowtie R$$

*Dieser Operator funktioniert so:*

$$L \bowtie R = \pi_{L\text{-Attribute}} ( L \bowtie R )$$

*Relation L*

A	B	C
a1	b1	c1
a2	b2	c2

*Relation R*

C	D	E
c1	d1	e1
c3	d3	e3

*Relation  $L \bowtie R$* 

A	B	C
a1	b1	c1



*Dieser Operator liefert nur die Tupel aus der rechten Relation, die die entsprechenden Tupel-Partner aus der linken Relation haben.*

*Dieser Operator wird so bezeichnet:*

$$L \bowtie R$$

*Dieser Operator funktioniert so:*

$$L \bowtie R = \pi_{R\text{-Attribute}} ( L \Join R )$$

*Es gelten folgende einfache Formeln:*

$$L \bowtie R = R \bowtie L$$

$$L \bowtie R = R \bowtie L$$

*Relation L*

A	B	C
a1	b1	c1
a2	b2	c2

*Relation R*

C	D	E
c1	d1	e1
c3	d3	e3

*Relation  $L \bowtie R$* 

C	D	E
c1	d1	e1



*Dieser Operator liefert nur die Tupel aus der linken Relation, die keine entsprechenden Tupel-Partner in der rechten Relation haben.*

*Dieser Operator wird so bezeichnet:*

$$L \triangleright R$$

*Es gilt folgende Formel:*

$$L \triangleright R = L - (L \bowtie R)$$

*Relation L*

A	B	C
a1	b1	c1
a2	b2	c2

*Relation R*

C	D	E
c1	d1	e1
c3	d3	e3

*Relation L  $\triangleright$  R*

A	B	C
a2	b2	c2



*Dieser Operator liefert nur die Tupel aus der rechten Relation, die keine entsprechenden Tupel-Partner in der linken Relation haben.*

*Dieser Operator wird so bezeichnet:*

$$L \triangleleft R$$

*Es gilt folgende Formel:*

$$L \triangleleft R = R - (L \bowtie R)$$

*Relation L*

A	B	C
a1	b1	c1
a2	b2	c2

*Relation R*

C	D	E
c1	d1	e1
c3	d3	e3

*Relation  $L \triangleleft R$* 

C	D	E
c3	d3	e3



*Die Operatoren JOIN sind in vielen SQL-Dialekten implementiert. Wenn sie irgendwo fehlen, kann man sie durch die WHERE-Klausel ersetzen.*

*Die JOIN-Operatoren sind in der FROM-Klausel implementiert. Die Syntax ist folgende:*

*SELECT Attribute FROM Tabelle1 JOIN Tabelle2 ON Bedingung*



*An der Stelle von JOIN können in SQL-Anweisungen stehen:*

- *CROSS JOIN*
- *NATURAL JOIN*
- *INNER JOIN*
- *LEFT OUTER JOIN*
- *RIGHT OUTER JOIN*
- *FULL OUTER JOIN*



*CROSS JOIN* entspricht dem gewöhnlichen kartesischen Produkt. Ein reines kartesisches Produkt (ohne Verknüpfungen zwischen den Tabellen) hat einen Sinn nur in bestimmten Situationen, z.B. in statistischen Auswertungen.

*Anfrage: Wie viel Prozent des gesamten Umfangs aller Vorlesungen (aller SWS) die SWS einer einzelnen Vorlesung umfasst.*

```
SELECT Titel, SWS, tmp.Summe,  
       SWS*100/tmp.Summe as ProzAnteil  
FROM Vorlesungen,  
     ( SELECT sum(SWS) as Summe FROM Vorlesungen ) tmp;
```

```
SELECT Titel, SWS, tmp.Summe,  
       SWS*100/tmp.Summe as ProzAnteil  
FROM Vorlesungen CROSS JOIN  
     ( SELECT sum(SWS) as Summe FROM Vorlesungen ) tmp;
```

- – Einfach JOIN funktioniert unter Oracle nicht,
- – nur JOIN ON.



*NATURAL JOIN wird für die Tabellen ausgeführt, die Spalten mit gleichen Namen (und Datentypen) haben. Die Spaltennamen werden dabei nicht geschrieben.*

*Anfrage: Welche Studenten hören welche Vorlesungen.*

```
SELECT MatrNr, Name, Titel  
FROM Studenten NATURAL JOIN hoeren  
      NATURAL JOIN Vorlesungen;
```

```
SELECT MatrNr, Name, Titel  
FROM Studenten NATURAL JOIN Vorlesungen  
      NATURAL JOIN hoeren;
```

*– – Ohne NATURAL JOIN*

```
SELECT s.MatrNr, s.Name, v.Titel  
FROM Studenten s JOIN hoeren h ON s.MatrNr=h.MatrNr  
      JOIN Vorlesungen v ON v.VorLNr=h.VorLNr;
```



*Anfrage: Welche Professoren welche Assistenten haben.*

```
SELECT p.Name Professor, a.Name Assistant  
FROM Professoren p FULL OUTER JOIN Assistenten a  
ON p.PersNr = a.Boss;
```

*– – a little different*

```
SELECT p.Name Professor, a.Name Assistant  
FROM Professoren p JOIN Assistenten a  
ON p.PersNr = a.Boss;
```

*– – the same without JOIN*

```
SELECT p.Name Professor, a.Name Assistant  
FROM Professoren p, Assistenten a  
WHERE p.PersNr = a.Boss;
```



Relativ einfach sehen die SQL-Anweisungen für die Semi-Operatoren aus.

*Semi-JOIN von L mit R:*

```
select L.* from L inner join R on L.x = R.x
```

```
select L.* from L inner join R using (x)
```

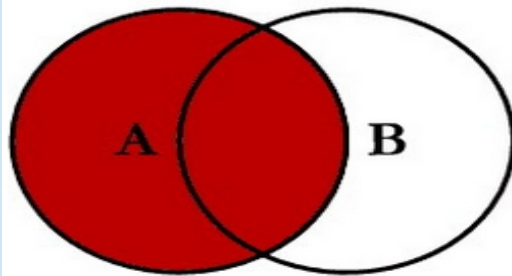
*Semi-JOIN von R mit L:*

```
select R.* from R inner join L on L.x = R.x
```

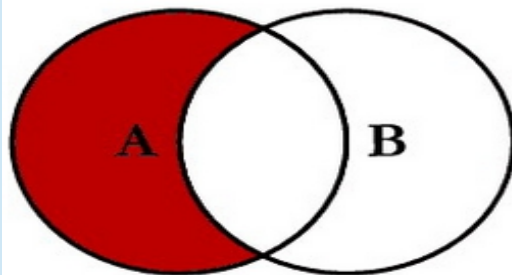
```
select R.* from R inner join L using (x)
```



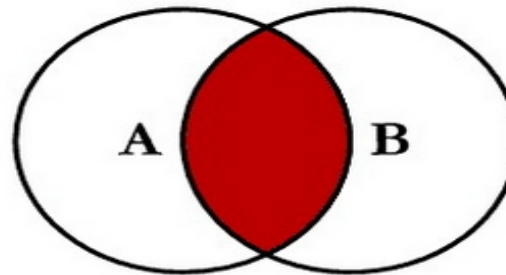
# SQL JOINS



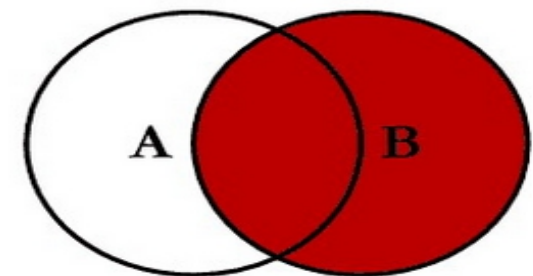
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



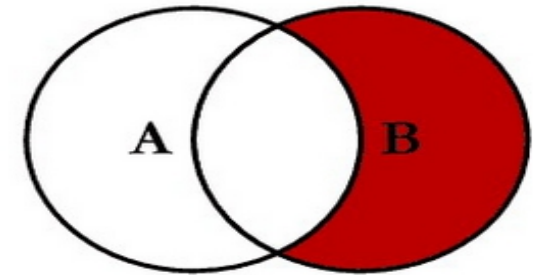
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



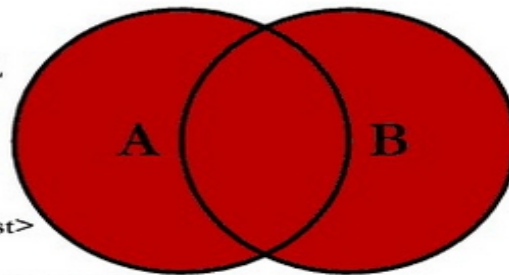
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



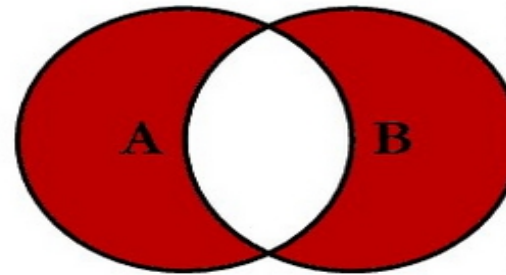
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

© C.L. Moffatt, 2008

and here [Visual-Representation-of-SQL-Joins](https://stackoverflow.com/questions/448023/what-is-the-difference-between-left-right-outer-and-inner-joins) explained in detail by [C.L. Moffatt](#)

<https://stackoverflow.com/questions/448023/what-is-the-difference-between-left-right-outer-and-inner-joins>



Verarbeitung der SQL-Anweisungen, sowie deren Optimierung, ist eine zentrale Aufgabe des DBMS.

Ablauf der Ausführung einer SQL-Anweisung:

- Eine SQL-Anweisung wird nach der Eingabe zuerst vom Parser auf Syntax überprüft.
- Danach erstellt Optimizer den optimalen Zugriffsplan auf die Daten der involvierten Tabellen.
- Der Optimizer übergibt den Zugriffsplan an den Row Source Generator, der einen Ausführungsplan auf die physikalischen Ressourcen generiert.
- Die Execution Engine erzeugt die Ergebnisse auf Basis dieses Ausführungsplans.



*Der Optimizer kann nach einem von folgenden Algorithmen arbeiten:*

- RBO (rule-based optimizer) berechnet den Zugriffspfad unter Verwendung der intern festgelegten Regeln.*
- CBO (cost-based optimizer) verwendet die internen Statistiken über einzelne Tabelle und Indizes, um den Zugriffspfad zu berechnen.*

*Das regelbasierte Verfahren fand den Einsatz in alten Versionen der Datenbanken (Oracle).*

*Das kostenbasierte Verfahren ist allgemein empfehlenswert. Es ist effizienter, da ihm die neuesten Informationen über die Leistung zugrunde liegen. Dies betrifft insbesondere die umfangreichen Abfragen mit mehreren JOINS oder Indizes. Entsprechend muss man die Statistiken regelmäßigen aktualisieren (Befehl ANALYZE).*

*Der Benutzer kann Hinweise an den Optimizer geben (Oracle), z.B.*

*`/*+CHOOSE */`      `/*+ORDERED */`*



*Die Informationen in der Datenbank werden nach folgenden Verfahren gesucht:*

- *Vollständiger Table-Scan (Full tablescan) wird verwendet, wenn eine Tabelle keine geeigneten Indizes hat. Das ist die langsamste Methode.*
- *Index-Scan wird verwendet, wenn geeigneter Index für eine Tabelle existiert. Das ist die schnellste Methode. RowID (Zeilennummer) ist Rückgabewert der Suche in Index.*
- *Hash-Scan wird verwendet, wenn keine geeigneten Indizes existieren. Für Inhalt der Tabelle werden Hash-Werte generiert.*



*Für Verknüpfung der Tabellen können mehrere Verfahren eingesetzt werden, abhängig von Bedingungen.*

- *Verschachtelte Schleifen. Es wird eine Schleife organisiert, in der die mit der JOIN-Bedingung übereinstimmenden Werte aus einer Tabelle ausgewählt werden. In diese Schleife wird eine verschachtelte Schleife eingebaut. In der verschachtelten Schleife wird die zweite Tabelle nach diesen Werten durchsucht.*
- *Sort-Merge-Join (Sortierung und Zusammenfassung). Beide Tabellen werden sortiert und in beiden Tabellen werden die Zeilen gesucht, die mit der JOIN-Bedingung übereinstimmen.*
- *Hash-Join. Für eine Tabelle wird eine Hash-Tabelle auf Basis des JOIN-Kriteriums erzeugt. Diese Hash-Werte werden dann verwendet, um nach Werten aus der anderen Tabelle zu suchen.*



- *Kartesisches Produkt. Fehlen in der SQL-Anweisung einige Bedingungen für Tabellen-Verbindungen, wird entsprechendes Kreuzprodukt erstellt.*
- *Index-JOIN. Existieren passende Indizes für die zu verknüpfenden Tabellen, dann werden Indizes statt Tabellen verknüpft. Dieses Verfahren funktioniert, nur wenn Indizes einspaltig sind.*



*Indizes beschleunigen die Verarbeitung der Abfragen beachtlich. Deren Verwaltung ist teilweise dem Benutzer überlassen und teilweise wird automatisch von Oracle gemacht.*

*Indizierungskonzepte bei Oracle bieten unterschiedliche Arten von Indizes. Hier werden nur folgende betrachtet:*

- Konventionelle Indizes als Binärbäume. Sie eignen sich besser für die Spalten, die viele unterschiedliche Werte enthalten, z.B. "PersID", "Matrikelnummer".*
- Bitmap-Indizes. Sie eignen sich besser für die Spalten, die viele gleiche Werte enthalten, z.B. "Geschlecht", "Kategorie".*

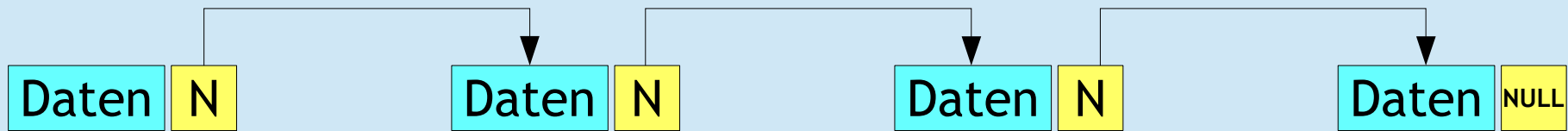
*Im Allgemeinen kann man sich einen Index zu einer Tabelle so vorstellen:*

$$\{ [ \text{Suchfeld}, \underline{\text{RowID}} ] \}$$

*Dementsprechend können die Indizes eindeutig und mehrdeutig sein.*



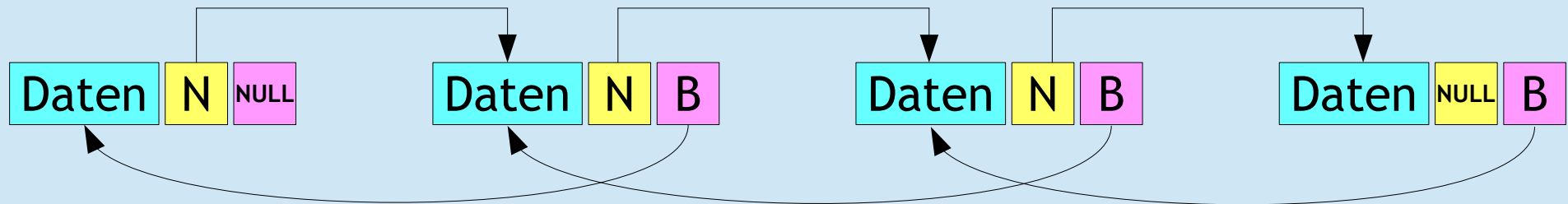
## Einfach verkettete Liste



```
typedef struct
{
    int Daten;
    Element *next;
}
Element;
```



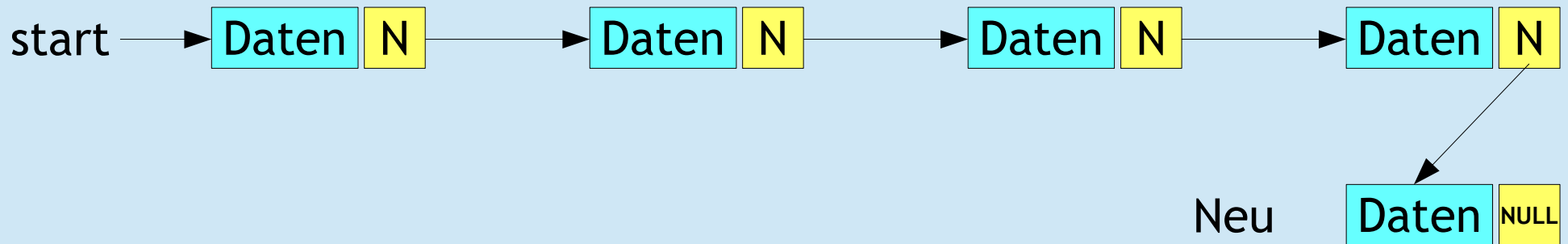
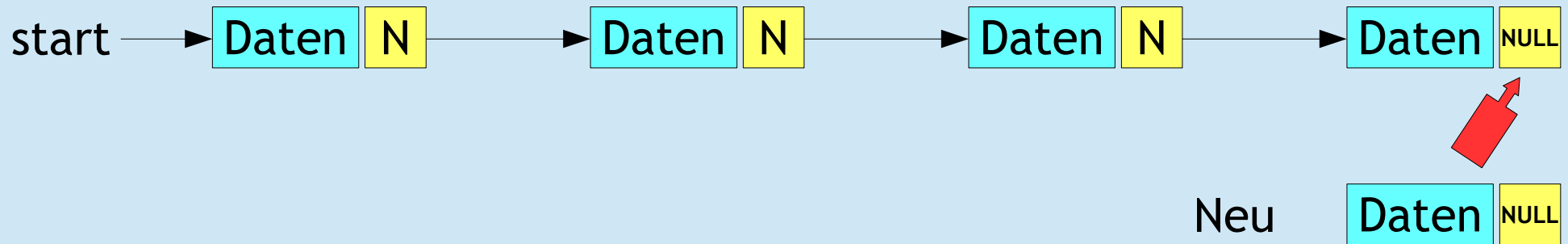
## Doppelt verkettete Liste



```
typedef struct
{
    int Daten;
    Element *next;
    Element *back;
}
Element;
```

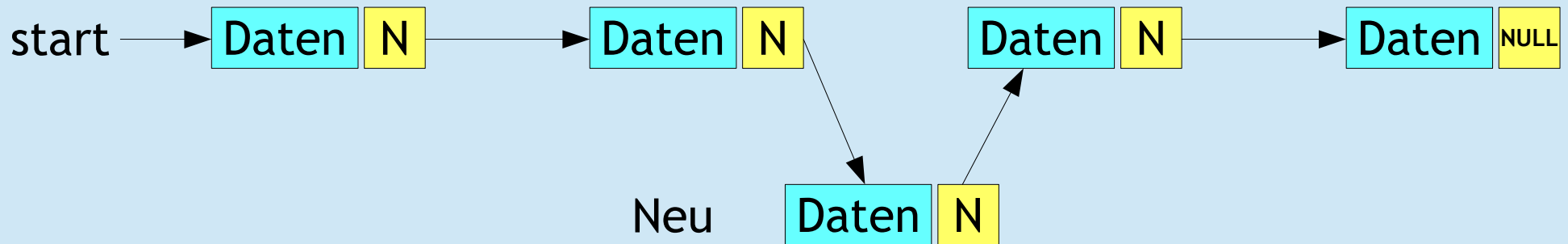
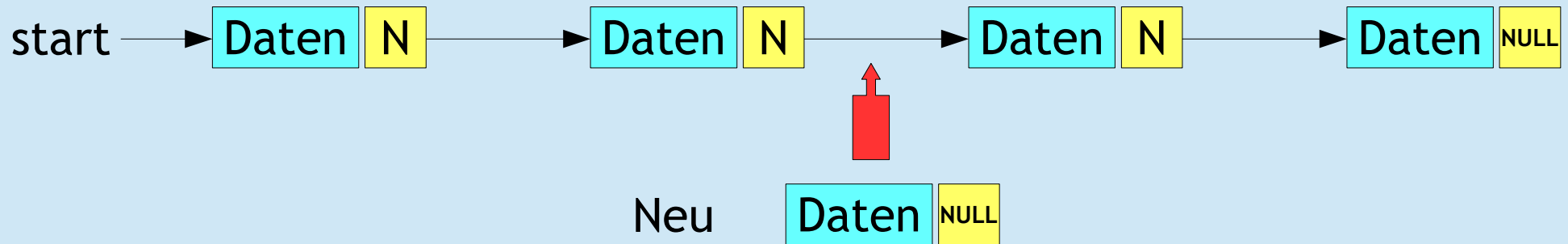


*Neues Element am Ende der Liste anhängen (keine Sortierung).*





Liste sortieren durch Einfügen eines neuen Elementes.

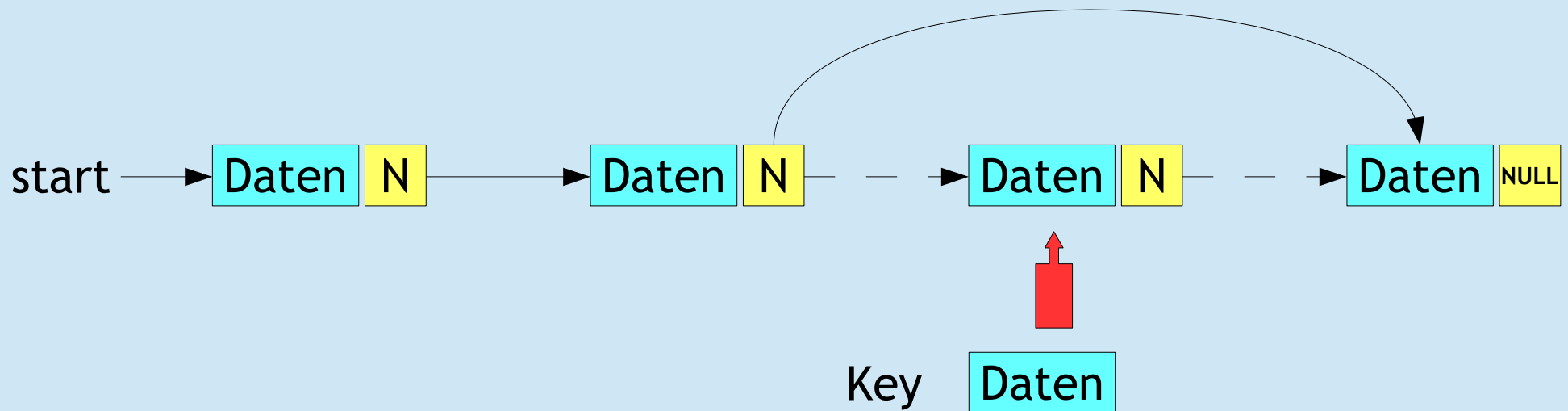




*Löschen von Elementen in linearen Listen ist recht einfach.*

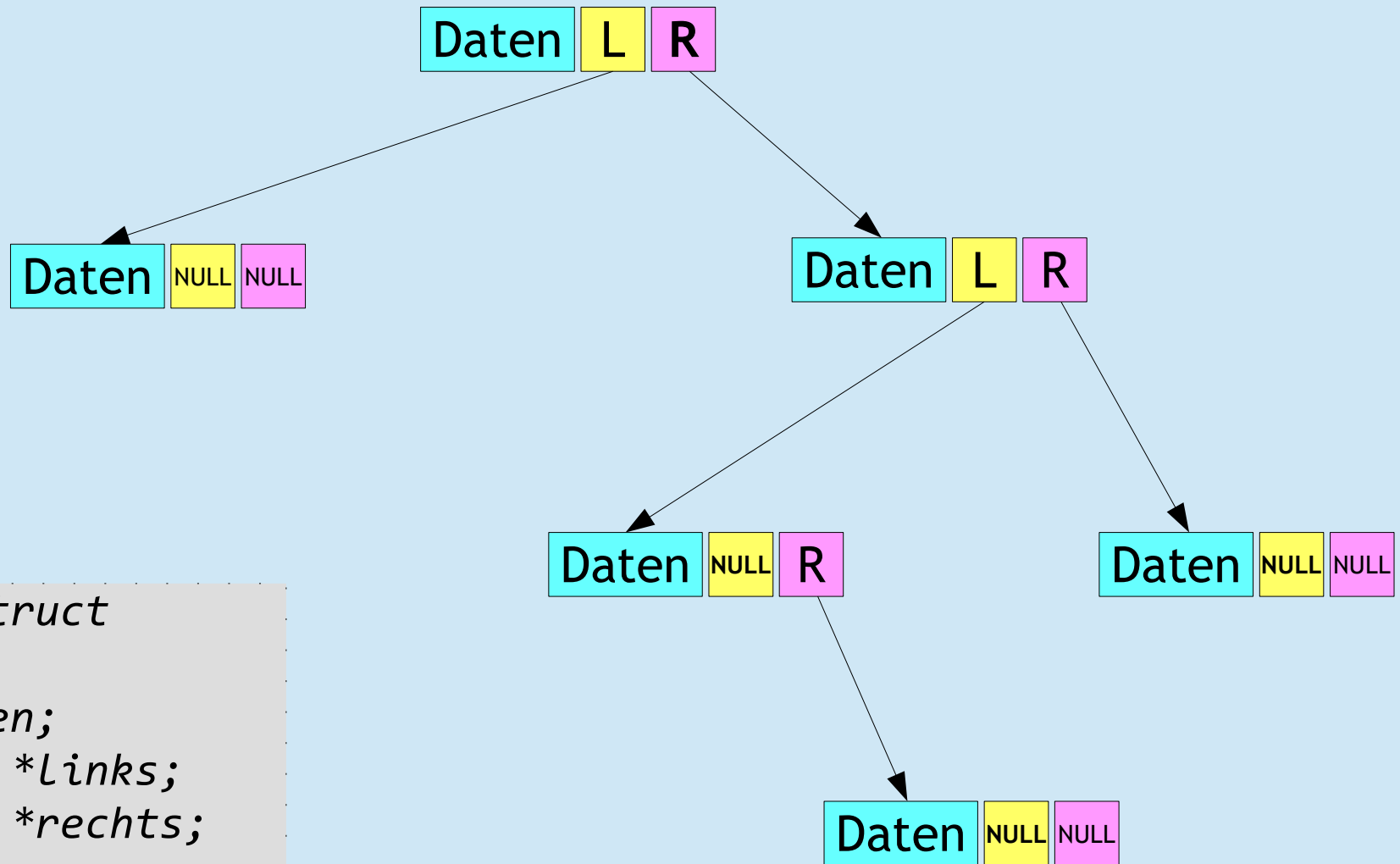
*Ein Schlüssel (Key) muss eingegeben werden, er wird in der Liste gesucht. Falls entsprechendes Element gefunden wird, wird es gelöscht. Dabei lediglich der Zeiger des Vorgängers auf den Nachfolger gesetzt werden muss.*

*Der Speicherplatz des Elements ist freizugeben.*





## Binärer Baum



```
typedef struct
{
    int Daten;
    Element *links;
    Element *rechts;
}
Element;
```



*Als Baum bezeichnet man eine dynamische Datenstruktur, in der jedes Element exakt einen Vorgänger und möglicherweise mehrere Nachfolger hat. Ausnahme bildet nur das höchste Element (sog. Root), es hat keine Vorgänger.*

*Beispiele:*

- *Suchsysteme;*
- *Dokumenten mit Gliederung;*
- *Aufbau einer Internet-Domäne;*
- *Nummerierung der Räume in einer Hochschule;*
- *Bauteile eines Mechanismus.*

*Hat ein Baum per Definition höchstens zwei direkte Nachfolger, dann heißt er binärer Baum.*



*Ein Baum kann so charakterisiert werden:*

- Baumstrukturen bestehen aus Baumstrukturen mit kleineren Größen.*
- Die Elemente des Baums heißen Knoten, wobei das Anfangselement Wurzel (Root) heißt.*
- Die Endpunkte heißen Blätter. Sie haben keine Nachfolger.*
- Die Wurzel ist der einzige Knoten der keinen Vorgänger hat.*
- Knoten, die weder Wurzel noch Blatt sind, heißen innere Knoten. Solche Knoten haben sowohl Vorgänger als auch Nachfolger.*
- Die Verbindungslinie zwischen zwei Knoten heißt Kante. Sie ist immer gerichtet und zeigt von dem Vorgänger zum Nachfolger.*
- Die Weglänge zwischen zwei Knoten wird durch die Anzahl der Kanten bestimmt.*



- *Knoten, die gleiche Länge der Pfade von der Wurzel haben, bilden jeweils eine Ebene.*
- *Die Gesamtzahl der Ebenen gibt die Tiefe des Baumes an. Oder, anders ausgedrückt, gibt die Anzahl der Knoten an dem längsten Pfad die Tiefe des Baumes an.*
- *Die maximale Anzahl der direkten Nachfolger, die ein Knoten hat, heißt Grad des Baumes. Also, der binäre Baum ist der Baum des zweiten Grades. Lineare Liste ist der Baum des ersten Grades.*



*Die Knoten in einem Baum können untereinander keine Ordnung haben, dann heißt der Baum ungeordnet. Das ist eine relativ seltene Situation.*

*Meisten Bäume sind geordnet, d.h. die Nutzinhalt der Knoten beinhalten Schlüssel, und diese Schlüssel geben die Ordnung im Baum vor.*

*Alle (oder fast alle) Operationen in dem Baum basieren auf den Verhältnissen zwischen den Schlüsseln:*

- neuen Knoten hinzufügen;*
- einen Knoten finden;*
- einen Knoten löschen.*

*Weiterhin werden die geordneten binären Bäume betrachtet.*



Weil alle Operationen in einem geordneten binären Baum auf einem Suchvorgang basieren, und die Suche hier den logarithmischen Aufwand hat (vgl. mit der binären Suche in einer geordneten linearen Liste), spielen die Bäume eine sehr wichtige Rolle in Informationssystemen.

## Problem.

Im Laufe der Zeit, wenn neue Elemente in den Baum hinzugefügt oder alte daraus gelöscht werden, ändert sich der Baum, und damit ändern sich seine Eigenschaften, z.B. der Suchaufwand kann vom logarithmischen bis zum linearen wechseln:  $\log(N) \rightarrow N$ .

## Lösung.

Man muss den Baum regelmäßig pflegen, oder spezielle Input/Output-Operationen implementieren, die seine Eigenschaften beibehalten.



Definition.

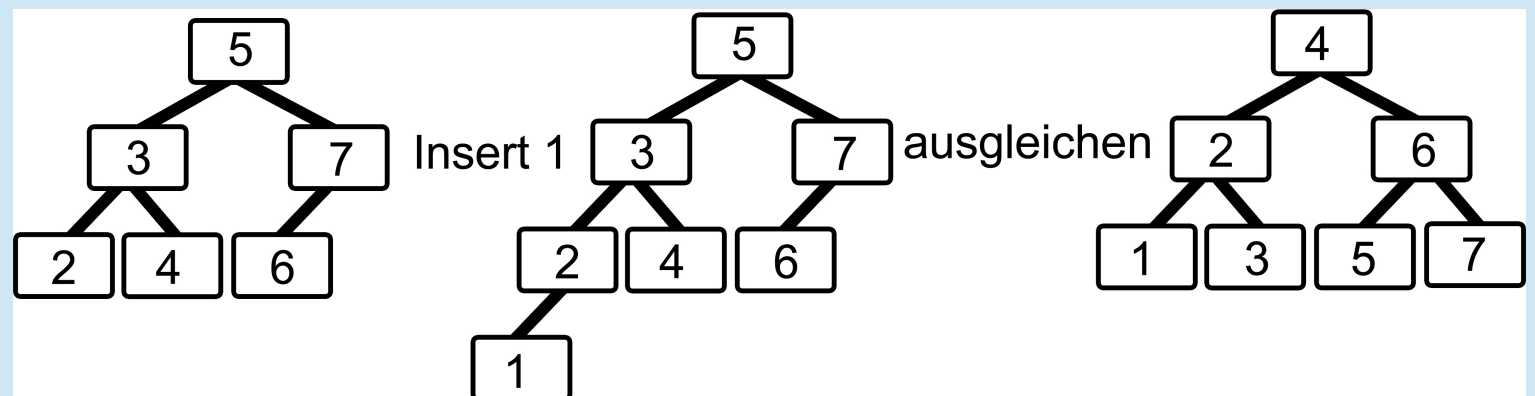
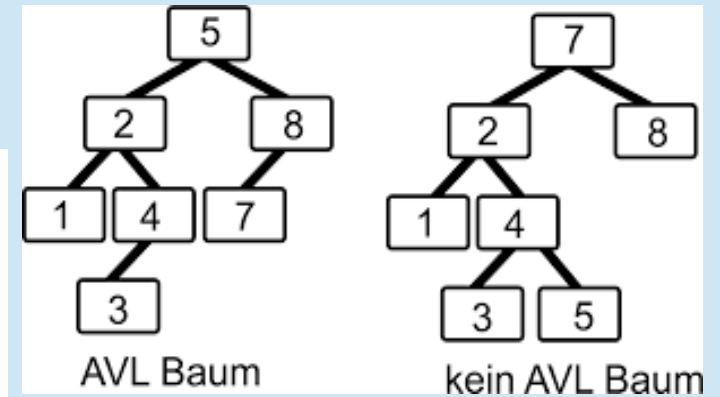
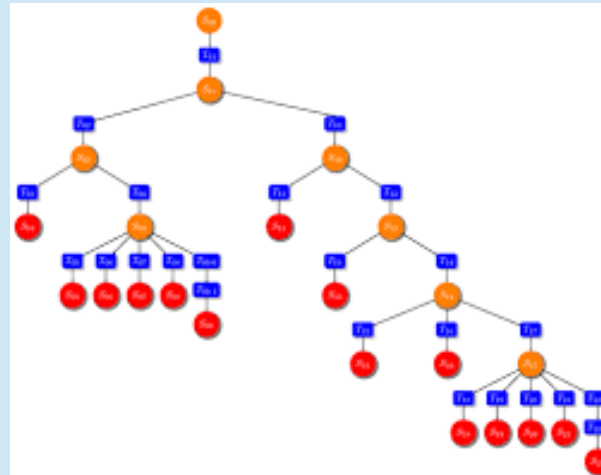
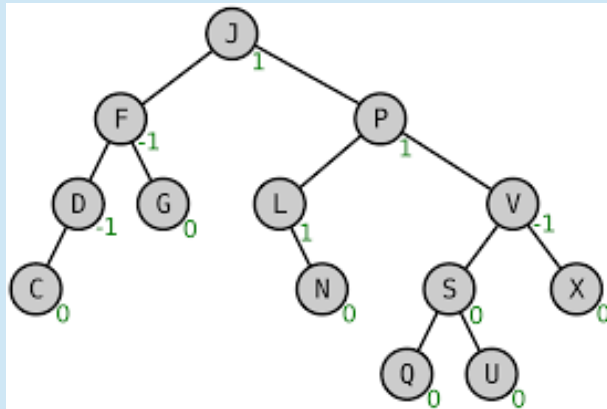
Ein binärer geordneter Baum heißt AVL-Baum, wenn sich die Höhe der beiden Teilbäume an jedem Knoten höchstens um eins unterscheidet.

Definition.

Ein binärer geordneter Baum ist balanciert (ausgeglichen), wenn höchstens die letzte Ebene nicht voll mit Knoten besetzt ist.

Bei einem AVL-Baum können höchstens zwei letzte Ebenen nicht voll mit Knoten besetzt werden, deswegen ist ein balancierter Baum immer auch ein AVL-Baum, umgekehrt ist es aber nicht – nicht jeder AVL-Baum ist ein balancierter Baum. Umgekehrte Aussage "Sind höchstens zwei letzte Ebenen nicht voll mit Knoten besetzt werden, dann ist es ein AVL-Baum" ist aber falsch (s. Beispiel mit Tieren).

Diese beiden Arten von Bäumen unterscheiden sich im Aufwand, der nötig ist, um die beiden Eigenschaften zu unterhalten. Bei einer sehr großen Anzahl der Knoten weisen die AVL-Bäume und die balancierten Bäume sehr ähnliche Charakteristiken auf.





*Bearbeitung (Suchen, Einfügen, Löschen) der Knoten und der entsprechenden Teilbäume ist mit Durchlauf des Baums eng verbunden.*

*Meist eingesetzte Richtungen, um einen Baum durchzulaufen:*

- *Preorder (WLR):  
zuerst wird der aktuelle Knoten verarbeitet, dann der linke Teilbaum, dann der rechte Teilbaum.*
- *Inorder (LWR):  
zuerst wird der linke Teilbaum verarbeitet, dann der aktuelle Knoten, dann der rechte Teilbaum.*
- *Postorder (LRW):  
zuerst wird der linke Teilbaum verarbeitet, dann der rechte Teilbaum, dann der aktuelle Knoten.*

*Die anderen Möglichkeiten (WRL, RLW, u.s.w.) werden selten verwendet.*



*Die drei Richtungen für Verarbeitung des Baums kann man für verschiedene Zwecke verwenden:*

- *Preorder (WLR) kann verwendet werden, um den Baum linear auf einem Datenträger zu speichern. Beim Einlesen der Daten entsteht genau so ein Baum.*
- *Inorder (LWR) kann verwendet werden, um eine sortierte einfach verkettete lineare Liste zu erstellen. Diese Liste kann man auf einem Datenträger speichern. Aus dieser Liste kann man einen balancierten Baum erstellen (wichtig!).*
- *Postorder (LRW) kann verwendet werden, um die Geräte zu programmieren, wo zuerst die Parameter in die Register geladen werden müssen, und nur dann die dazu gehörige Operation.*



*Der wichtigste Vorgang ist Balancieren des Baums. Ist ein Baum ausgeglichen, die höchste Zugriffsgeschwindigkeit auf seine Knoten und Blätter ist gewährleistet. Es gibt folgende Balancier-Methoden:*

- Offline-Methode. Man erstellt eine Kopie des Baums und alle Zugriffe auf diese Kopie werden gesperrt. Man erstellt aus dem Kopie-Baum eine sortierte einfach verkettete lineare Liste (LWR-Ausgabe). Diese Liste wird nach dem binären Verfahren in einen neuen Baum eingefügt. Als Ergebnis bekommt man einen ausgeglichenen Baum, der den momentan aktiven Baum ersetzt. Es ist programm-technisch das einfachste Verfahren.*
- Online-Methode. Zur Laufzeit werden die Ebenen des Baums rekursiv ausgeglichen, das kann kurzfristig zu Fehlern führen, da die Verweise geändert werden. Für ein Element auf der untersten Ebene können die Änderungen bis zur Wurzel ziehen. Die Einfüge- und Lösch-Operationen sollten auch gesperrt werden.*



*Hashing-Verfahren kommt zum Einsatz, wenn sehr viele Daten in kurzer Zeit durchgesucht oder verarbeitet werden müssen.*

*Beispiele der Einsatzbereiche:*

- Speicherung von großen Datenmengen in Datenbanken;*
- Berechnung der Prüfsummen zur Überprüfung der Integrität von Daten;*
- Ver- und Entschlüsselung der elektronischen Kommunikationswege, um die Geheimhaltung, Integrität und Authentizität der Nachrichten zu gewährleisten;*
- assoziative Arrays (dynamische Referenzierung von Programmbefehlen im Compiler).*



## Funktionsprinzip von Hashing:

- *Es gibt viele sehr große Datensätze, die gespeichert werden müssen mit dem Zweck, sie später finden und verarbeiten zu können.*
- *Für jeden Datensatz kann ein eindeutiger Schlüssel gebildet werden, der allerdings auch sehr große sein kann – im schlimmsten Fall ist es der ganze Datensatz.*
- *Es gibt eine Funktion (Hash-Funktion), die jedem Schlüssel einen relativ kurzen Wert (Hash-Wert) zuordnet.*
- *Die Hash-Funktion soll eigentlich eindeutig (injektiv) sein, d.h. zu zwei unterschiedlichen Schlüsseln soll sie unterschiedliche Hash-Werte bilden. Leider ist es sehr selten möglich, weil die Daten und Schlüssel groß sind. Liefert die Hash-Funktion zu zwei Schlüsseln einen Hash-Wert, so spricht man über eine Kollision.*
- *Kollisionen müssen extra verarbeitet werden.*

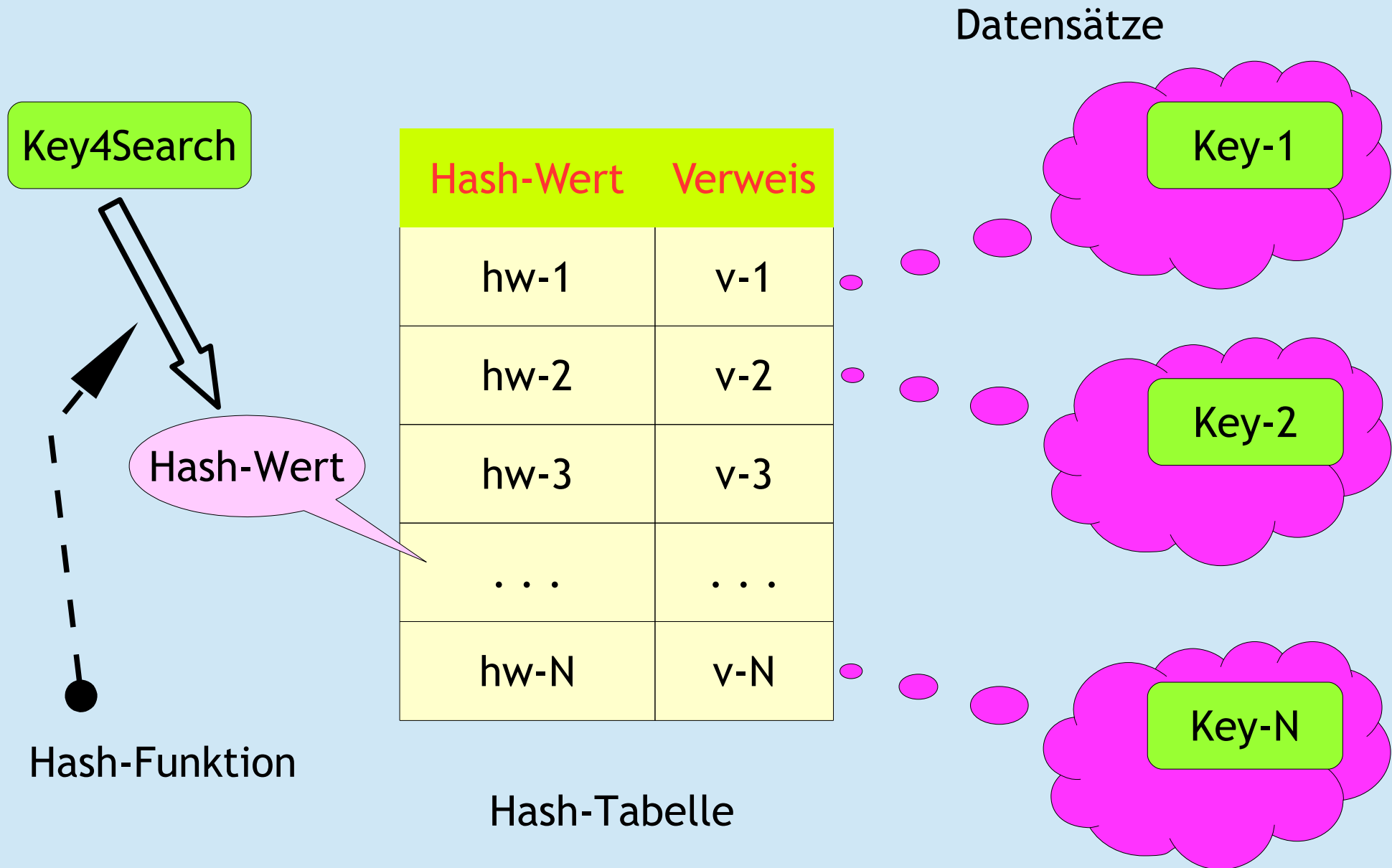


- *Um einen Datensatz zu finden, reicht es, den Schlüssel zu besitzen.*
- *Zu dem Schlüssel wird entsprechender Hash-Wert gebildet.*
- *Dieser Hash-Wert wird als Index in der Hashing-Tabelle (zweispaltige Tabelle) verwendet, die Hash-Werte und Verweise auf entsprechende Datensätze enthält.*
- *Ist die Übereinstimmung vorhanden, werden der Such-Schlüssel und der Schlüssel des gespeicherten Datensatzes verglichen. Stimmen auch sie überein, ist der gesuchte Datensatz gefunden. Sonst liegt ein Kollisionen vor, die nach besonderen Methoden verarbeitet werden soll.*



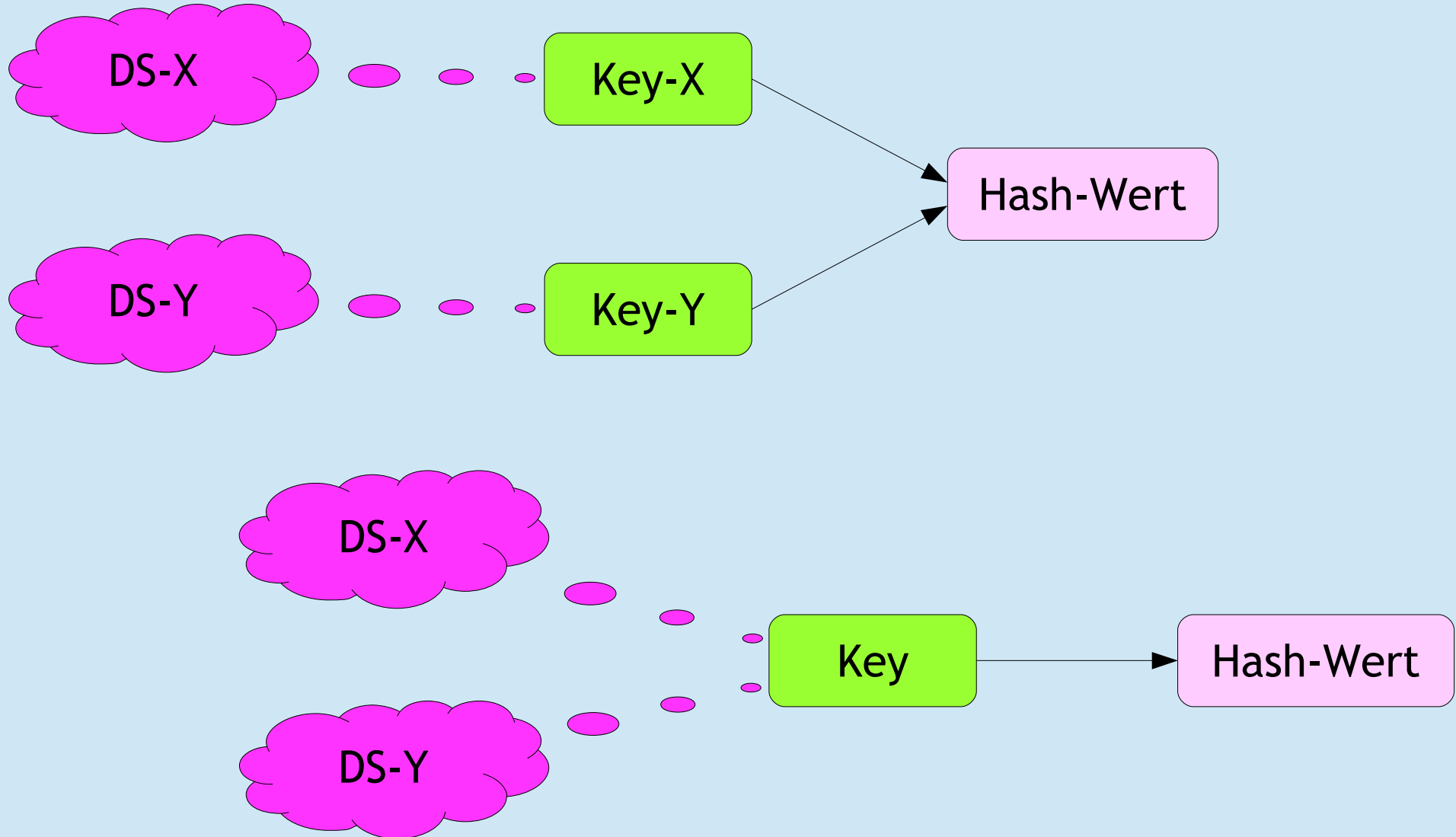
*Mehrere Hashing-Verfahren existieren, die sich durch Behandlung der Kollisionen und durch Verwaltung der gespeicherten Datensätzen unterscheiden:*

- *Hashing mit Verkettung (wird unten betrachtet).*
- *Lineares Hashing (dynamische Erweiterung der Tabelle, wenn Belegungsfaktor der verketteten Liste zu groß wird).*
- *Hashing mit linearer/quadratischer/zufälliger Sondierung der Tabelle (Datensätze werden direkt in der Tabelle gespeichert).*





## Kollisionen beim Hashing:

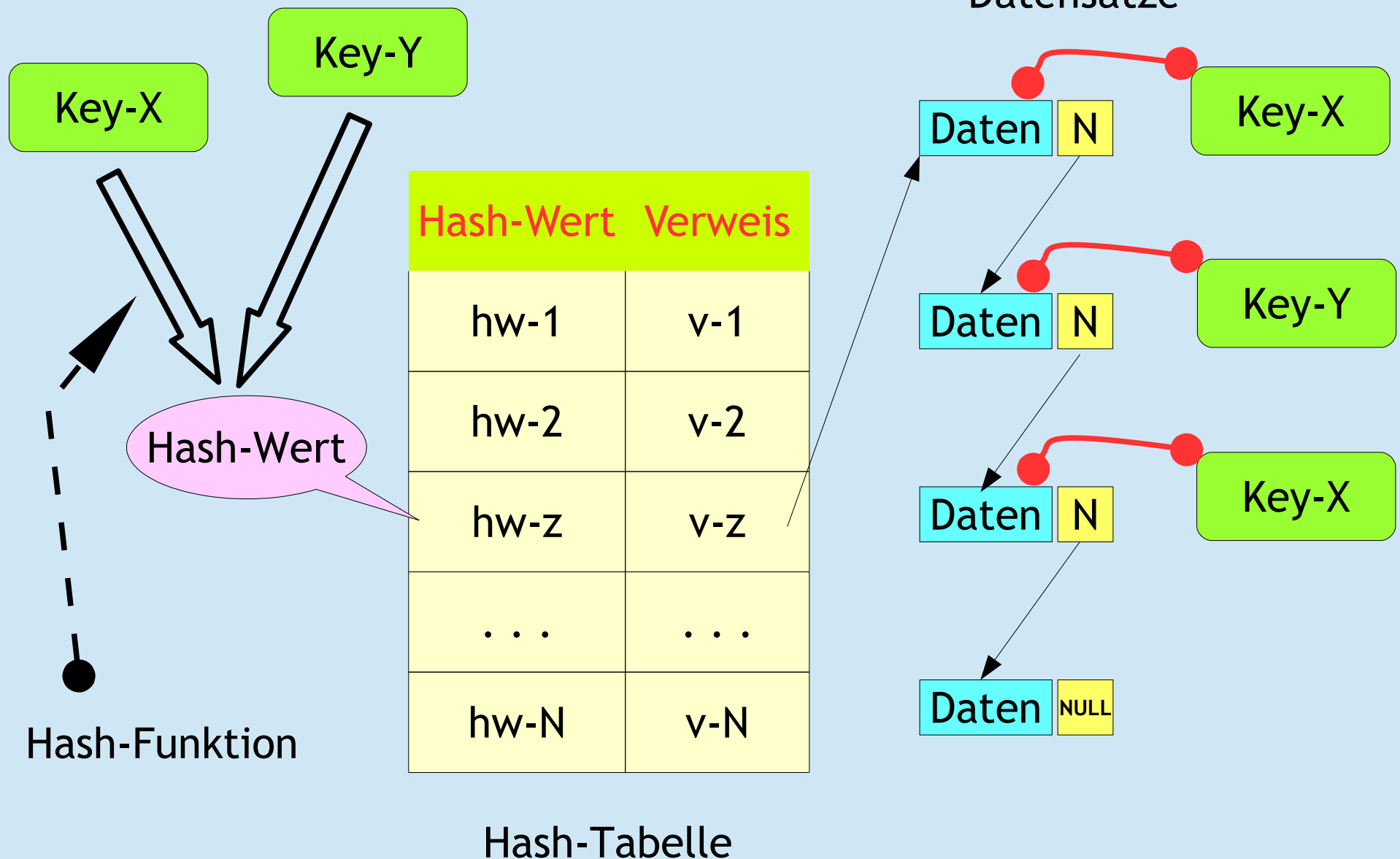




*Probleme, die bei großen Datensätzen entstehen und erfolgreich durch Hashing gelöst werden können:*

- Die Schlüssel von zwei Datensätzen sind zwar unterschiedlich, aber generieren den selben Hash-Wert. Das ist eine durchaus gängige Situation, da die Daten sehr große sind – Kollision.*
- Zwei unterschiedliche Datensätze haben den selben Schlüssel, was selbstverständlich zu einem einzigen Hash-Wert führt – Kollision.*

*Lösung gibt der Aufbau der Hash-Tabelle selbst – sie enthält keine eigentlichen Datensätze, sondern nur Verweise (Zeiger) auf diese Datensätze. Von hier aus kann man entweder eine lineare verkettete Liste aufbauen, oder einen Baum, oder beliebige Kombinationen von abstrakten Datentypen. Das löst die Kollisionen, kann aber zu schlechteren Zugriffszeiten führen.*





### Extrem-Beispiel:

Der Befehl `md5sum` erstellt einen Hash-Wert zu einem eingegebenen Text oder zu einer beliebigen Datei (d.h. als Schlüssel dient die ganze Datei). Obwohl sich die unten dargestellten Texte geringfügig unterscheiden, sind die entsprechenden Hash-Werte total unterschiedlich.

```
art@cent Do Mär 16 16:09:22 ~/work  
=> md5sum -t
```

Ich weiss nicht, was soll es

```
fb62d4e2d72ff279ce60e598d89d8347 -
```

```
art@cent Do Mär 16 16:09:44 ~/work  
=> md5sum -t
```

Ich weiss nicht, was soll es.

```
658491fdef953783e832d3b2a7cae13c -
```



## Anforderungen an Hash-Funktion:

- Die Funktion soll zu einem Schlüssel den entsprechenden Hash-Wert (Hash-Adresse) schnell berechnen und keinen großen Speicherbedarf ausweisen (Effizienz).
- Die Funktion soll wenig Kollisionen erzeugen (Kollisionsresistenz). Kleine Wahrscheinlichkeit von Kollisionen der Hashwerte bedeutet eine Gleichverteilung der Hashwerte auf die Schlüssel.
- Die Funktion soll eine Einwegfunktion sein. Aus dem Hash-Wert sollte der Schlüssel kaum oder nicht berechnet werden.
- Der Speicherbedarf der Hashwerte soll deutlich kleiner sein, als jener der Schlüssel.
- Kein Hash-Wert soll unmöglich sein (Surjektivität).



- *Schlüssel, die sich wenig unterscheiden, sollen zu völlig verschiedenen Hash-Werten führen. Unterschied der Schlüssel in einem Bit führt zu Unterschieden in mindestens Hälfte der Bits der Hash-Werte (Lawineneffekt).*

*Ansonsten ist die Gestaltung einer guten Hash-Funktion eine wahre Kunst, die auf guten Kenntnissen der Schlüsselstruktur (des Eingabebereiches) und vielen Test-Versuchen und Adaptionen basiert.*



## *Beispiele der Hash-Funktionen:*

- *Modulo-Berechnung;*
- *Abschneiden;*
- *Zerlegung und Addition.*

*In folgenden Beispielen werden die 4-stelligen dezimalen Zahlen als Schlüssel und die 2-stelligen dezimalen Zahlen als Hash-Werte betrachtet.*



## Beispiel 1 – Modulo-Berechnung.

*Hash-Wert = Schlüssel % Basis, Basis = 100*

*Key1 = 4159, HW =  $4159 \% 100 = 59$  – das ist Hash-Wert.*

*Key2 = 6973, HW =  $6973 \% 100 = 73$  – das ist Hash-Wert.*

*Als Basis sind die Primzahlen empfehlenswert, die so nahe wie möglich zur Basis sind. In dem Fall ist empfehlenswerte Basis 97.*

*Key1 = 4159, HW =  $4159 \% 97 = 85$  – das ist Hash-Wert.*

*Key2 = 6973, HW =  $6973 \% 97 = 86$  – das ist Hash-Wert.*



## Beispiel 2 – Abschneiden.

*Schlüssel wird potenziert (hoch 2) und dann von links und rechts abgeschnitten, bis das Ergebnis in den passenden Bereich kommt.*

*Key1 = 3425*

*$3425 * 3425 = 11730625 \Rightarrow 173062 \Rightarrow 7306 \Rightarrow 30$  – das ist Hash-Wert.*

*Key2 = 1781*

*$1781 * 1781 = 3171961 \Rightarrow 17196 \Rightarrow 719 \Rightarrow 1$  – das ist Hash-Wert.*



## Beispiel 3 – Zerlegung und Addition.

*Der Schlüssel wird in mehrere Teile zerlegt, die dieselbe Ziffernanzahl wie die gesuchte Hash-Adresse haben. Diese Teile werden dann addiert.*

*Key1 = 5723  $\Rightarrow$  57 + 23 = 80 – das ist Hash-Wert.*

*Key2 = 2815  $\Rightarrow$  28 + 15 = 43 – das ist Hash-Wert.*



## *Vorteile binärer Bäume im Vergleich zum Hashing:*

- Garantie für Leistungsfähigkeit auch im ungünstigsten Fall;*
- Unterstützung von vielen Operationen außer Suchen (das Sortieren);*
- Größe der Bäume ist dynamisch, d.h. Information über die Anzahl der Einfügungen bei binären Bäumen ist nicht nötig, im Gegensatz zum Hashing.*



## Aufbau, Eigenschaften und Empfehlungen für Bitmap-Indizes:

- *Attribut (Spalte) hat sehr wenige unterschiedliche (einzigartige) Werten, d.h. geringe Kardinalität. Geringe Kardinalität zählt von 0,1% bis 1%.*
- *Daten in Tabellen werden nie oder nur selten geändert (wenige INSERT/UPDATE/DELETE-Operationen) – Data Warehouse (OLAP).*
- *Für jeden einzigartigen Wert des Tabellenattributes wird eine Bit-Spate im Index gebildet.*
- *Ausnahmen bestätigen die Regeln – die Bitmap-Indizes, unter bestimmtem Bedingungen, können gute Leistungen auch in dem Fall zeigen, wenn die Daten nicht unbedingt eine niedrige Kardinalität aufweisen. Sie können manchmal erfolgreich auch für die Attribute mit vielen Tausenden von verschiedenen Werten angewendet werden.*



## Funktionsweise der Bitmap-Indizes:

PersNr	Filial
AB-0911	Marzahn
BC-0921	Köpenick
AB-0927	Mitte
AD-0953	Köpenick
XY-0913	Mitte
BC-0973	Köpenick
AB-0968	Marzahn
XY-0919	Marzahn

*Tabelle Mitarbeiter*

RowNum	Marzahn	Köpenick	Mitte
0001	1	0	0
0002	0	1	0
0003	0	0	1
0004	0	1	0
0005	0	0	1
0006	0	1	0
0007	1	0	0
0008	1	0	0

*Bitmap-Index*



## Funktionsweise der Bitmap-Indizes:

<u>MatrNr</u>	Geschl.	Alter
s0911	M	31
s0921	M	19
s0927	W	33
s0953	W	27
s0913	W	33
s0973	M	25
s0968	W	19
s0919	W	27

*Tabelle Studenten*

RowNum	Geschl.	18-21	22-28	29-35
0001	1	0	0	1
0002	1	1	0	0
0003	0	0	0	1
0004	0	0	1	0
0005	0	0	0	1
0006	1	0	1	0
0007	0	1	0	0
0008	0	0	1	0

*Bitmap-Index*



## Vorteile der Bitmap-Indizes:

- *Sie sind stark komprimiert und können deswegen schnell gelesen werden, sowohl im Arbeitsspeicher als auch von dem Datenträger.*
- *Sie ermöglichen mehrere Indizes für einen schnellen Zugriff auf die zugrunde liegende Tabelle zu kombinieren.*
- *Sie werden sehr schnell verarbeitet, weil logische Operationen relativ schnell im Prozessor implementiert sind.*

## Nachteile der Bitmap-Indizes:

- *Bitmap-Indizes weisen einen immensen Aufwand bei ihrer Wartung auf – im Vergleich zu binären Bäumen.*
- *Bandbreite der Kanäle im Prozessor ist sehr wichtig.*
- *Bitmap-Indizes können Deadlocks verursachen.*

