

Architektur und Implementierung von Datenbanksystemen WS 05/06

Dr. Jens-Peter Dittrich

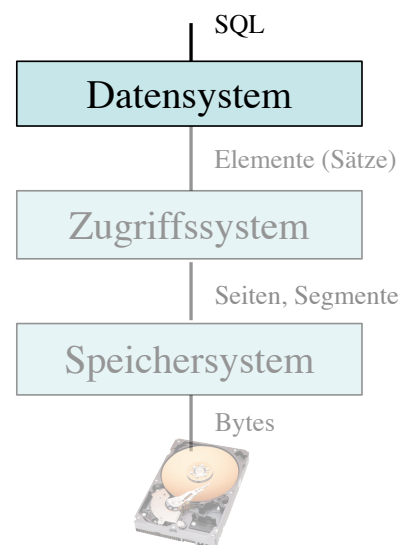
jens.dittrich@inf

www.inf.ethz.ch/~jensdi

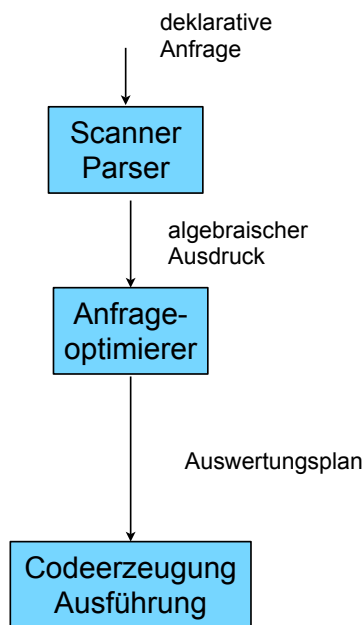
Institut für Informationssysteme

Datensystem

3. Implementierung relationaler Operatoren



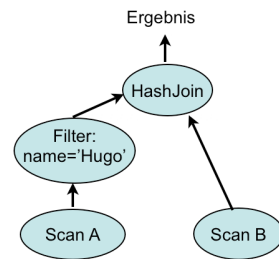
Motivation & Übersicht



```

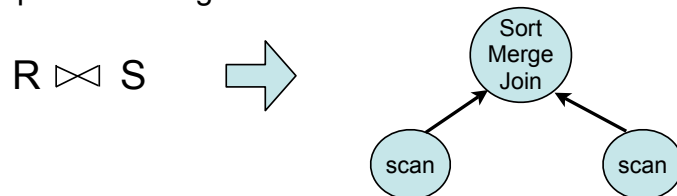
SELECT title
FROM A,C
WHERE A.name = 'Hugo' AND A.id = C.dz
  
```

$$\Pi_{\text{title}} (\sigma_{\text{A.name}='Hugo' \text{ and } \text{A.id}=\text{B.dz}} (\text{A} \times \text{B}))$$



Physische Operatoren

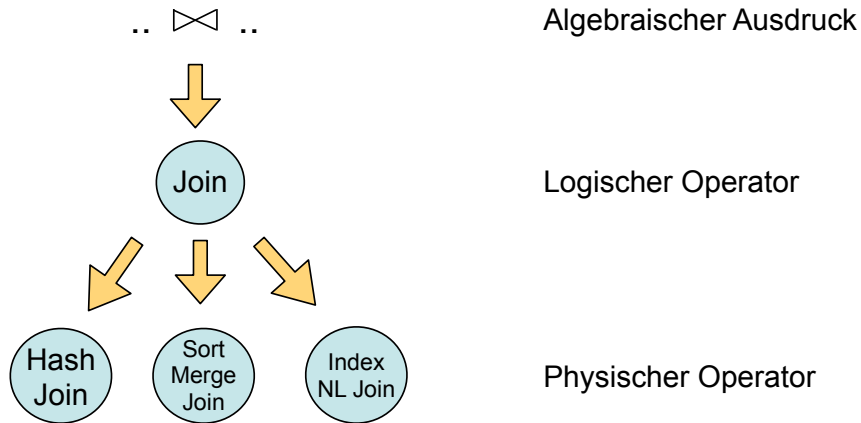
- Physische Operatoren sind die Atome eines Ausführungsplans
- Jeder Operator berechnet eine Funktion $f()$
- Jeder Ausdruck wird während der Optimierung auf einen oder mehrere physische Operatoren abgebildet



- für einen algebraischen Ausdruck kann es verschiedene Implementierungen geben

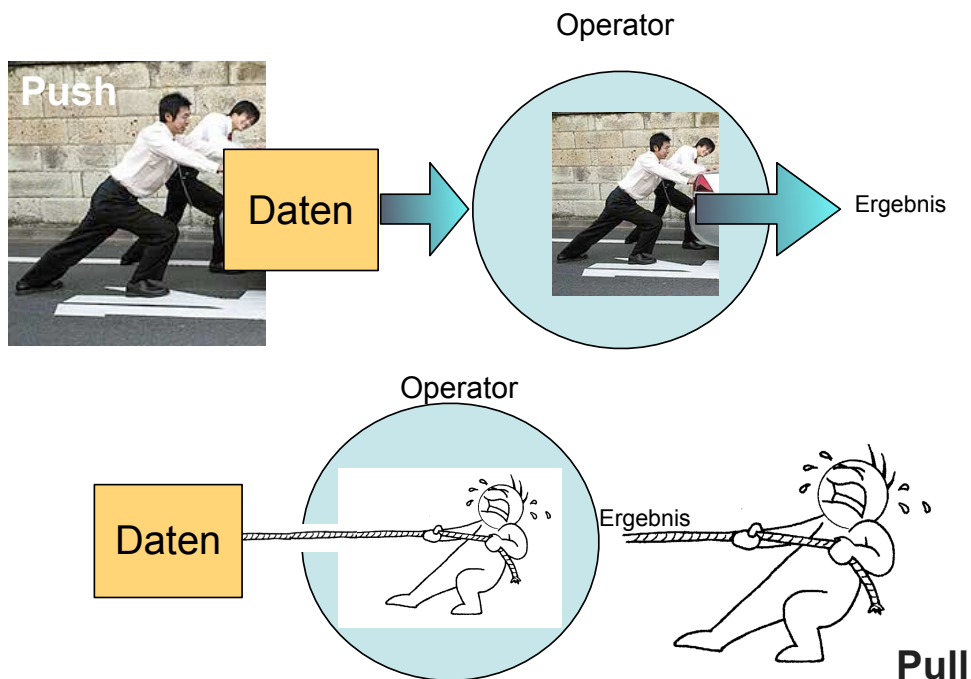


Operatoren: logisch vs. physisch

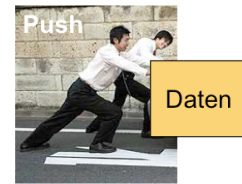


Heutige Vorlesung: Physische Operatoren

Push vs. Pull

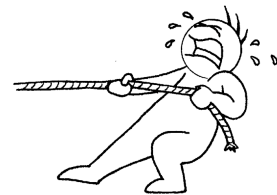


Push (Stream-Modell)



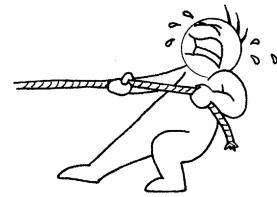
- Datenquellen produzieren Daten und reichen diese an den nächsten Operator weiter.
- Die Verarbeitung der Daten wird von den Datenquellen ausgelöst.
- Die Datenverarbeitung erfolgt **datengetrieben**.
- Beispiel
 - Temperatursensoren schicken Messwerte an lokale Zentren. Diese aggregieren die Messwerte und schicken sie weiter an eine zentrale Station für einen grösseren Bereich, usw.
- Vorteil:
 - Daten werden nach ihrer Entstehung sofort verarbeitet.
- Nachteil:
 - Benutzer kann Verarbeitung nicht anhalten oder abbrechen.

Pull (Iteratormodell)



- Der Benutzer fordert das nächste Ergebnistupel der Berechnung an.
- Die Verarbeitung der Daten wird vom Benutzer ausgelöst.
- Die Datenverarbeitung erfolgt **benutzergetrieben**.
- Beispiel:
 - "Computer, Zeig mir die nächstgelegene Pizzeria."
 - Pause.
 - "Computer, Zeig mir die zweitnächstgelegene Pizzeria."
 - Pause.
- Vorteil:
 - der Computer verrichtet keine überflüssige Arbeit (zu viele Ergebnisse)
- Nachteil:
 - Aufwand für Durchreichen der Benutzercalls.

Pull: Implementierung

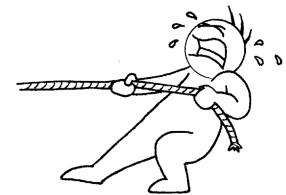


- Implementierung erfolgt über Iterator-Interface
- void **open()**:
Initialisiert den Operator
- Object **next()**:
gibt das nächste Element zurück und entfernt es aus dem Iterator
- void **close()**
schliesst den Operator

Man nennt dieses Interface auch **ONC**-Interface.

- Literatur: Goetz Graefe: Volcano - An Extensible and Parallel Query Evaluation System. IEEE Trans. Knowl. Data Eng. 6(1): 120-135 (1994)

java.util.Enumeration (Java 1.0)



java.util

Interface Enumeration

All Known Subinterfaces:

[NamingEnumeration](#)

All Known Implementing Classes:

[StringTokenizer](#)

public interface Enumeration

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the `nextElement` method return successive elements of the series.

For example, to print all elements of a vector `v`:

```
for (Enumeration e = v.elements(); e.hasMoreElements(); ) {
    System.out.println(e.nextElement());
}
```

Methods are provided to enumerate through the elements of a vector, the keys of a hashtable, and the values in a hashtable. Enumerations are also used to specify the input streams to a `SequenceInputStream`.

NOTE: The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional `remove` operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration.

Since:

JDK 1.0

See Also:

[Iterator](#), [SequenceInputStream](#), [nextElement\(\)](#), [Hashtable](#), [Hashtable.elements\(\)](#), [Hashtable.keys\(\)](#), [Vector](#), [Vector.elements\(\)](#)

Method Summary

boolean	hasMoreElements()	Tests if this enumeration contains more elements.
Object	nextElement()	Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

java.util.Iterator (Java 1.2)



java.util

Interface `Iterator`

All Known Subinterfaces:

[ListIterator](#)

All Known Implementing Classes:

[BeanContextSupport.BCSIterator](#)

public interface `Iterator`

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the [Java Collections Framework](#).

Since:

1.2

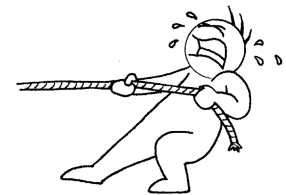
See Also:

[Collection](#), [ListIterator](#), [Enumeration](#)

Method Summary

boolean	hasNext()	Returns <code>true</code> if the iteration has more elements.
Object	next()	Returns the next element in the iteration.
void	remove()	Removes from the underlying collection the last element returned by the iterator (optional operation).

java.util.Iterator<E> (Java 1.5)



java.util

Interface `Iterator<E>`

All Known Subinterfaces:

[ListIterator<E>](#)

All Known Implementing Classes:

[BeanContextSupport.BCSIterator](#), [Scanner](#)

public interface `Iterator<E>`

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the [Java Collections Framework](#).

Since:

1.2

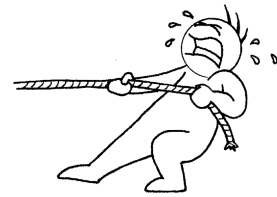
See Also:

[Collection](#), [ListIterator](#), [Enumeration](#)

Method Summary

boolean	hasNext()	Returns <code>true</code> if the iteration has more elements.
E	next()	Returns the next element in the iteration.
void	remove()	Removes from the underlying collection the last element returned by the iterator (optional operation).

java.util.Iterator

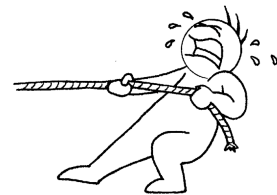


- void **open()**:
nicht vorhanden, passiert implizit im Konstruktor
- boolean **hasNext()**:
liefert true zurück, falls der Iterator noch ein Element liefert
- Object **next()**:
gibt das nächste Element zurück und entfernt es aus dem Iterator
- void **close()**
finalize()-Methode?

```
public static void main(String argv[] throws Exception {  
    Object object = new Object() {  
        protected void finalize() {  
            System.err.println("test");  
        }  
    };  
    System.err.println(object);  
}
```

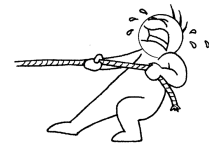
```
Console Search Tasks Problems  
<terminated> FolderList [Java Application] /System/Library//  
Test$1@6a55fa
```

Iteratoren: Einfache Beispiele



- **Projektion** {
Iterator input;
public Object next() {
 return project(input.next());
}}
- **Selektion** {
Iterator input;
public Object next() {
 Object result = null;
 do {
 result = input.next();
 } while (!P(result))
 return result;
}}

Iteratoren: Schleifen



- **Schleife in next() packen? NEIN!**

```
public Integer next() {  
    for(int i=0; i<42; i++){  
        return i;  
    }  
    throw new NoSuchElementException();  
}
```

Wie unterbreche ich hier?
...und mache später an
derselben Stelle weiter?

- **Lösung**

```
int i=0; // Schleifenzähler wird Attribut der Klasse  
public Integer next() {  
    while(i<42){  
        Integer result = i;  
        i++;  
        return result;  
    }  
    throw new NoSuchElementException();  
}
```

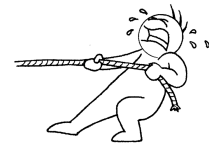
AbstractIterator: computeNext()



```
public abstract class AbstractIterator implements Iterator {  
    protected Object next = null;  
    protected Iterator results = null;  
    protected boolean hasNext = false;  
    protected boolean setNextIterator(Iterator results) {}  
    protected boolean setNext(Object result) {}  
    protected boolean getNext() {}  
    abstract protected boolean computeNext();  
    public boolean hasNext() {}  
    public Object next() {}  
    public void remove() {}  
}
```

computeNext() wird gerufen
wann immer neue
Ergebnisse berechnet
werden müssen.
Nur diese Methode muss
vom Entwickler
implementiert werden!

AbstractIterator: setNext(Object)

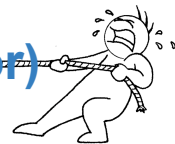


```
public abstract class AbstractIterator implements Iterator {  
    protected Object next = null;  
    protected Iterator results = null;  
    protected boolean hasNext = false;  
    protected boolean setNextIterator(Iterator results) {}  
    protected boolean setNext(Object result) {}  
    protected boolean getNext() {}  
    abstract protected boolean computeNext();  
    public boolean hasNext() {}  
    public Object next() {}  
    public void remove() {}  
}
```

immer wenn der Entwickler
ein neues Ergebnis
<instance> zurückgeben
kann, ruft er im Code

return setNext(instance);

AbstractIterator: setNextIterator(iterator)



```
public abstract class AbstractIterator implements Iterator {  
    protected Object next = null;  
    protected Iterator results = null;  
    protected boolean hasNext = false;  
    protected boolean setNextIterator(Iterator results) {}  
    protected boolean setNext(Object result) {}  
    protected boolean getNext() {}  
    abstract protected boolean computeNext();  
    public boolean hasNext() {}  
    public Object next() {}  
    public void remove() {}  
}
```

immer wenn der Entwickler
einen Iterator <iterator> mit
Ergebnissen zurückgeben
kann, ruft er im Code

return setNextIterator(iterator);

AbstractIterator: Beispiel



```
import java.util.Iterator;

public class SortBasedDistinct extends AbstractIterator {

    protected Iterator input = null;

    protected Object peek = null;

    public SortBasedDistinct(Iterator input) {}

    protected boolean computeNext() {
        if (this.input.hasNext()) {
            //compute next element <_next> here
            return setNext(_next);
        } else
            return false;
    }
}
```

Es muss nur sehr wenig Code in der Methode computeNext() implementiert werden. Alles andere macht der AbstractIterator!

Mehr hierzu in der 2. Programmieraufgabe

AbstractIterator als Ergebnis einer Methode



```
import java.util.Iterator;

public class QueryProcessor {

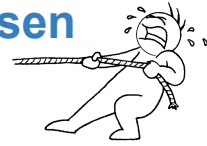
    public QueryProcessor() {
    }

    public Iterator query(String queryExpression) throws Exception {
        return new AbstractIterator() {
            protected boolean computeNext() {
                // put your code here
                return false;
            }
        };
    }
}
```

Ergebnis einer Anfrage soll als Iterator zurückgegeben werden

Lösung: anonyme Klasse (implizit nicht-statisch!)

Statische vs. nicht-statische innere Klassen



```
public class QueryProcessor {
    protected List myList = null;

    public static class MyIteratorStatic extends AbstractIterator {
        protected boolean computeNext() {
            myList.get(42);
            return false;
        }
    }

    public class MyIteratorNonStatic extends AbstractIterator {
        protected boolean computeNext() {
            myList.get(42);
            return false;
        }
    }

    public QueryProcessor() {}

    public Iterator query(String queryExpression) throws Exception {
        return new AbstractIterator() {
            protected boolean computeNext() {
                if (myList.size() > 42) {
                    return setNext(myList.get(42));
                }
                return false;
            }
        };
    }
}
```

Zugriff **nicht** erlaubt!
Diese **statische** innere Klasse hat keine Referenz auf die Instanz von QueryProcessor!

Diese **nicht-statische** innere Klasse hat eine Referenz auf die Instanz von QueryProcessor!

Zugriff erlaubt!
Diese anonyme Klasse ist implizit **nicht-statisch**.

AbstractIterator und Konstruktoren



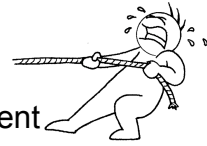
```
public Iterator query(String queryExpression) throws Exception {
    return new AbstractIterator() {
        public AbstractIterator() {
            //code to initialize this instance
        }
        protected boolean computeNext() {
            if (myList.size() > 42) {
                return setNext(myList.get(42));
            }
            return false;
        }
    };
}
```

Überschreiben des Konstruktors nicht erlaubt!

```
public Iterator query(String queryExpression) throws Exception {
    return new AbstractIterator() {
        {
            //code to initialize this instance
        }
        protected boolean computeNext() {
            if (myList.size() > 42) {
                return setNext(myList.get(42));
            }
            return false;
        }
    };
}
```

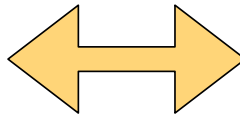
Lösung:
Instanz-Initialisierer wird vor dem Konstruktor gerufen

Grobgranulare Iteratoren



- Bisher wurde davon ausgegangen, dass immer nur ein Element als Ergebnis von next() produziert wird.
- Dies hat zur Folge, dass für jeden Aufruf von next() u.U. sehr viele kaskadierte Methodenaufrufe erfolgen.
- Dies kann zu einem Overhead führen, der in gewissen Anwendungen nicht mehr tolerierbar ist.
- Lösung
 - auf Objektmengen iterieren
 - auf Seiten iterieren
- Trade-off zwischen

Feingranular iterieren
Viele Methodenaufrufe
Sauberes Design
Schlechtere Performanz



Grobgranular iterieren
Wenige Methodenaufrufe
Aufwendigeres Design
Höhere Performanz

Operatoren: Übersicht



- Heutige DBMS implementieren (leider) nur das Pull-Modell.
- Wichtige Operatoren:
 - Sortieren (externen)
 - Joins
 - Gruppierung und Aggregation
 - Selektion
 - Division
 - Schnitt
- Desweiteren
 - Scan
 - IndexScan

Blockierende Operatoren

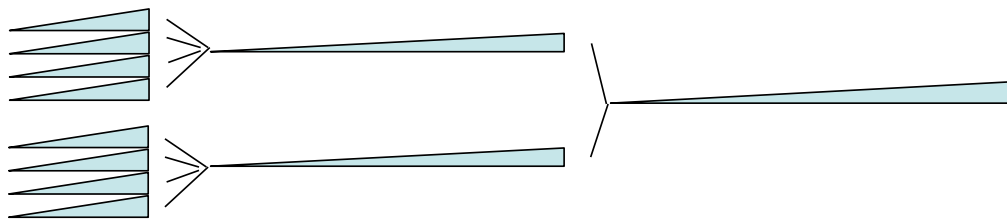


- Wie kann ein Sortieroperator sofort das nächste Ergebnis weiterreichen?
- Die Entscheidung welches das nächste Element ist kann nur getroffen werden, wenn der Sortieroperator alle Eingabeelemente gesehen hat!
- D.h. der Sortieroperator sammelt erst alle Objekte der Eingabe ein: er **blockiert** den Datenfluss.

Externes Sortieren

- Grundlage für viele andere Operatoren im DBMS
- Parameter:
 - B: Anzahl Sätze pro Datenseite
 - N: Anzahl der Daten (Sätze)
 - $n = \lceil N/B \rceil$, Anzahl der Daten (Seiten)
 - M: verfügbarer Hauptspeicher (Sätze)
 - $m = \lceil M/B \rceil$, verfügbarer Hauptspeicher (Seiten)
- Problem: N Datensätze passen nicht gleichzeitig in den Hauptspeicher
- Algorithmus:
 1. Phase: Run-Erzeugung
Solange R nicht leer:
 - Lade M Datensätze von R in den Hauptspeicher
 - Sortiere M im Hauptspeicher und schreibe sortierte Sequenz in temporäre Datei
 2. Phase: Merge-Phase
Solange (Anzahl Runs > 1):
 - Merge jeweils F Runs zu grösseren Runs zusammen

Externes Sortieren



Run-Erzeugung

Merge: Ebene 1

Merge: Ebene 2

- F: Verzweigungsgrad des Merge: $F = m - 1 = \Theta(m)$
- Anzahl der Verschmelzungsebenen: $\lceil \log_F n \rceil$
- E/A-Aufwand fürs Verschmelzen: $\Theta(n \log_m n)$
- Damit entspricht der asymptotische Aufwand für das externe Sortieren

$$\Theta(n \log_m n)$$

Replacement Selection

- Beobachtung: Länge der initialen Runs entspricht Grösse des HS: M
- R = Eingabe, R' = Ausgabe, h = Heapgrösse
- **Algorithmus: Replacement Selection**
 - Fülle HS mit M nächsten Tupeln von R
 - Erzeuge Heap auf den M Datensätzen, $h := M$
 - Solange Heap nicht leer:
 - Schreibe Top-Element r des Heaps nach R'
 - Sei t das nächste Element der Eingabe R
 - Falls $t \geq r$:
 - Füge t in heap ein
 - Sonst: ($t < r$)
 - $h := h - 1$
 - Speichere t an frei gewordenem Platz des HS

Replacement Selection Beispiel

	Ausgabe R'	Speicher HS				Eingabe R						
		10	20	30	40	25	73	16	26	33	50	31
	10	20	25	30	40	73	16	26	33	50	31	
	10 20	25	30	40	73	16	26	33	50	31		
	10 20 25	(16)	30	40	73	26	33	50	31			
	10 20 25 30	(16)	(26)	40	73	33	50	31				
	10 20 25 30 40	(16)	(26)	(33)	73	50	31					
Zeit ↓	10 20 25 30 40 73	(16)	(26)	(33)	(50)	31						
		16	26	31	33	50						

- Anfangszustand: HS enthält 10, 20, 30, 40
- Es wird jeweils das nächste Element t von der Eingabe R gezogen
- Ist t grösser (gleich) als das letzte top-Element: Einfügen in Heap
- Ist t kleiner als das letzte top-Element: Einfügen in HS

Bewertung

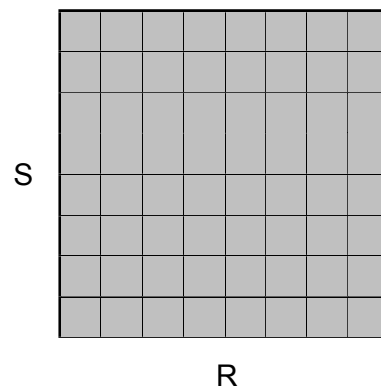
- Replacement Selection erzeugt im Schnitt Runs der Länge 2M (!)
- Bei teilsortierten Daten können die Runs noch länger werden
- Nachteil: Heap nicht cache-sensitiv
- Für Ein-&Ausgabe muss extra Speicher reserviert werden

Join-Algorithmen

- **Bedeutung**
 - wichtigste Operation im DBMS
 - grossen Einfluss auf Gesamtperformanz des DBMS
 - grossen Einfluss auf Blockierverhalten von Anfragen, d.h. Zeit bis das erste Anfragetupel vom DBMS produziert wird
 - kann zur Implementierung anderer Operatoren genutzt werden, Beispiel: Intersect, Minus, Komplement
- 3 wichtige Klassen von Verfahren
 - Nested-Loops Joins
 - Hash Joins
 - Sort-Merge Joins

Einfacher Nested-Loops Join

- Eingaben:
 - Relation R
 - Relation S
 - Join-Prädikat $P(r,s)$
 - Ergebnismenge $RES = \{ \}$
- Algorithmus:
 - Für alle r in R:
 - Für alle s in S:
 - Falls $P(r,s)$:
 - $RES = RES \cup \{ (r,s) \}$
- Bewertung:
 - $|R| \times |S|$ Vergleiche: $\Theta(n^2)$
 - Kann für sehr kleine Eingaberelationen Sinn machen
 - Für jedes Join-Prädikat einsetzbar



Seitenorientierter Nested-Loops Join

- Eingaben:
 - Relation R
 - Relation S
 - Join-Prädikat $P(r,s)$
 - Ergebnismenge $RES = \{ \}$

- Algorithmus:

Für alle **Seiten P** von R:

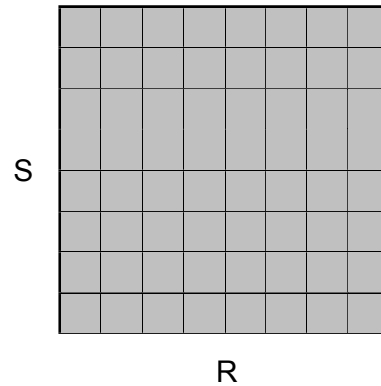
Für alle s in S:

Für alle r in P:

Falls $P(r,s)$:

$RES = RES \cup \{ (r,s) \}$

- CPU-Aufwand derselbe wie bei einfachem Nested-Loops Join
- Aber: wesentlich besseres E/A-Verhalten



Index Nested-Loops Join

- Eingaben:
 - Relation R
 - Relation S
 - Ergebnismenge $RES = \{ \}$

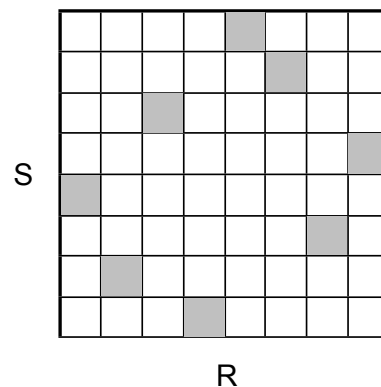
- Idee: nutze Indexstruktur auf einer der Eingaben aus

- Algorithmus:

Für alle r in R:

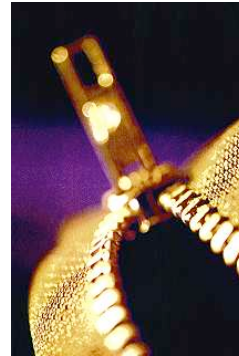
$RES = RES \cup (r, S.index.query(r))$

- Aufwand: $r \times$ Kosten für Index-Zugriff: $\Theta(n \log n)$
- Es kann sich u.U. lohnen extra für die Anfrage einen Index zu erzeugen



Sort-Merge Join

- Grundidee:
 - Sortiere beide Eingaben R und S (ggf. extern) nach demselben Kriterium
 - Durchlaufe sortierte Eingaben gleichzeitig und ermittle dabei Tupel für die das Join-Prädikat gilt
- Veranschaulichung: Reissverschluss



Beispiel: Sort-Merge

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32



Beispiel: Sort-Merge

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

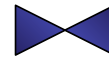


Beispiel: Sort-Merge

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

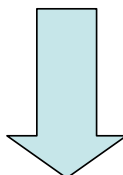


Beispiel: Sort-Merge

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



!



Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Hardware		Employee			
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43



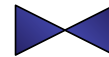
Beispiel: Sort-Merge

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Hardware		Employee			
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43

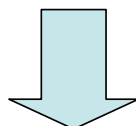


Beispiel: Sort-Merge

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32



Hardware		Employee			
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43
1	Slow PC	8	Müller	Peter	55



Beispiel: Sort-Merge

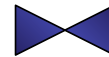
Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32



Hardware		Employee			
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43
1	Slow PC	8	Müller	Peter	55
12	Fast PC	42	Dittrich	Jens	32



Beispiel: Sort-Merge

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32



Hardware		Employee			
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43
1	Slow PC	8	Müller	Peter	55
12	Fast PC	42	Dittrich	Jens	32
36	Stapler	42	Dittrich	Jens	32



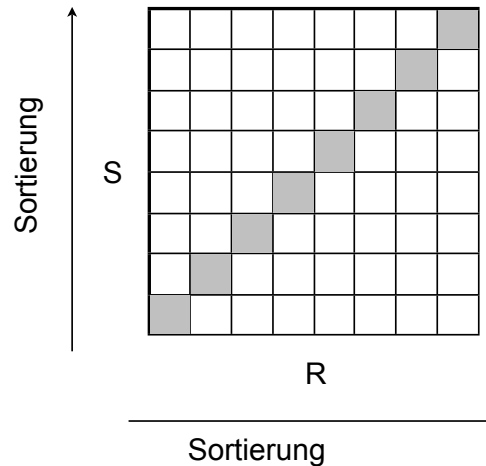
Beispiel: Sort-Merge

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32

Hardware		Employee			
ID	Description	PersNo.	Surname	First Name	Age
8	Fire Blade	7	Dittrich	Klaus	43
1	Slow PC	8	Müller	Peter	55
12	Fast PC	42	Dittrich	Jens	32
36	Stapler	42	Dittrich	Jens	32

Vergleichsstrategie: Sort-Merge Join



Bewertung

- Aufwand: $\Theta(n \log n + |RES|)$
- Sortierung kann auch durch einen geclusterten Index hergestellt werden
- Sortieren entfällt, falls ein vorhergehender Operator bereits die Sortierung hergestellt hat!
- Kombinationen sind möglich:
 - eine Relation bereits implizit sortiert auf dem Join-Attribut durch Index-Clusterung
 - die andere Relation wird explizit sortiert
- Dieses Verfahren kann verallgemeinert werden für andere Join-Prädikate (dazu später mehr).
- Ist keine der Joinattribute ein Primärschlüssel muss man u.U Zeiger zurücksetzen. Das kann man aber durch eine elegante Implementierung beheben! (Mehr dazu nächste Stunde)

Einfacher Hash-Join

- Annahme: R ist kleiner als S, d.h. $|R| < |S|$
- Algorithmus:
 - Lese R in den HS
 - Baue Hash-Tabelle auf R auf
 - Lese sequentiell die Tabelle S
 - Für jedes Tupel s aus S:
 $RES = RES \cup \{ s, R.hash_table.probe(s) \}$

Was passiert, wenn R grösser ist als der Hauptspeicher?

Grace Hash Join

- Algorithmus:
 - Partitioniere kleinere Tabelle $T \in \{ R, S \}$, so dass jede Partition T_i von T in den HS passt
 - T heisst **Build Input** oder **innere Tabelle**
 - Partitioniere $Q \in \{ R, S \}$, $Q \neq T$ mit derselben Partitionierungsfunktion in Partitionen Q_i (ein Paar (T_i, Q_i) , $i=j$ bezeichne korrespond. Partitionen)
 - Q heisst **Probe Input** oder **äussere Tabelle**
 - Für alle Partitionen T_i von T:
 - Lese T_i in den Hauptspeicher
 - Lese sequentiell alle Seiten von Q_i und joine diese mit T_i

Grundidee des Grace Hash Join

- Problem: R passt nicht in den Hauptspeicher
- Lösung, 1. Phase:
 - Divide & Conquer
 - Teile R in Partitionen R_i auf, so dass jede Partition R_i in den Hauptspeicher passt
 - Wende **dieselbe** Partitionierungsfunktion nun auf S an: es entstehen Partitionen S_j
 - D.h. R und S werden so partitioniert, dass gilt:
 - $\text{join}(R_i, S_j) = \{\}$, wenn $i \neq j$
(Joinergebnisse können nur in Paaren von Partitionen (R_i, S_i) vorkommen!)
- Lösung, 2. Phase:
 - Joine Paare von Partitionen mit dem einfachen Hash Join

Details zur Partitionierung

- Gegeben:
 - Eingabe R
- Gesucht:
 - Funktion, die für ein gegebenes Tupel aus r die zugehörige Partition bestimmt
- Wichtig:
 - Die Partitionierung muss mit Hilfe des Joinschlüssels erfolgen (Sonst wird die Bedingung $\text{join}(R_i, S_j) = \{\}$, wenn $i \neq j$ verletzt)
 - Round Robin oder ähnlich über TID funktioniert nicht!

Partitionierung mit Intervallen

M=3

- Idee: Teile Daten in feste Intervalle auf

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



		PersNo						
R ₁ =	<table border="1"><tr><td>4</td><td>Cray</td><td>3</td></tr></table>	4	Cray	3	1-6			
4	Cray	3						
R ₂ =	<table border="1"><tr><td>8</td><td>Fire Blade</td><td>7</td></tr><tr><td>1</td><td>Slow PC</td><td>8</td></tr></table>	8	Fire Blade	7	1	Slow PC	8	7-12
8	Fire Blade	7						
1	Slow PC	8						
R ₃ =	{ }	13-18						
R ₄ =	{ }	19-24						
R ₅ =	{ }	25-30						
R ₆ =	{ }	31-36						
R ₇ =	<table border="1"><tr><td>12</td><td>Fast PC</td><td>42</td></tr><tr><td>36</td><td>Stapler</td><td>42</td></tr></table>	12	Fast PC	42	36	Stapler	42	37-42
12	Fast PC	42						
36	Stapler	42						

Partitionierung mit Hash-Funktion

M=3

- Idee: Benutze Hash-Funktion Modulo Anzahl Partitionen
 - Ziel: Hash-Funktion soll die Daten möglichst gleichmässig auf die Partitionen verteilen
 - Beispiel für gute Hash-Funktion
 - Idee:
 - erzeuge eine **Pseudo**zufallszahl mit Hilfe eines Generators
 - benutze den Schlüssel PersNo als den **seed** des Generators
 - $h(\text{persNo}) := \text{random}(\text{perNo})$
 - Damit berechnet sich die Partition eines Elements als
 - $P = \text{random}(\text{perNo}) \% \langle \text{Anzahl Partitionen} \rangle$
 - W bezeichne im folgenden die Anzahl der Partitionen
 - Setze $W := \lceil R/(M-1) \times F \rceil$, $F \approx 1.2$
- (Falls dies nicht möglich oder ungünstig verteilte Daten: Rekursive Partitionierung)

Partitionierung mit Hash-Funktion

M=3
W=3

- Idee: Benutze HashFunktion Modulo Anzahl Partitionen

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



			rand(PersNo)	%3	
R ₁ =	12	Fast PC	42	19488	0
	36	Stapler	42	19488	0
R ₂ =	1	Slow PC	8	14218	1
	4	Cray	3	17893	1
R ₃ =	8	Fire Blade	7	15467	2

Kompletter Join: Phase 1

M=3
W=3

- Partitioniere R

Hardware (R)		
ID	Description	PersNo.
4	Cray	3
8	Fire Blade	7
1	Slow PC	8
12	Fast PC	42
36	Stapler	42



			rand(PersNo)	%3	
R ₁ =	12	Fast PC	42	19488	0
	36	Stapler	42	19488	0
R ₂ =	1	Slow PC	8	14218	1
	4	Cray	3	17893	1
R ₃ =	8	Fire Blade	7	15467	2

Kompletter Join: Phase 1

M=3
W=3

- Partitioniere **S**

Employee (S)			
PersNo.	Surname	First Name	Age
6	Meier	Hans	37
7	Dittrich	Klaus	43
8	Müller	Peter	55
42	Dittrich	Jens	32



	PersNo				rand(PersNo)	%3
S ₁ =	42	Dittrich	Jens	32	19488	0
S ₂ =	8	Müller	Peter	55	14218	1
S ₃ =	7	Dittrich	Klaus	43	15467	2
	6	Meier	Hans	37	18488	2

Kompletter Join: Phase 2, Paar (R₁,S₁)

- Lade erste Partition R₁ von R in den HS und erzeuge Hashtabelle hierauf:

R ₁ =	12	Fast PC	42
	36	Stapler	42

HS

- Führe ein probe für jedes Tupel aus S₁ gegen die Hashtabelle aus

S ₁ =	42	Dittrich	Jens	32
------------------	----	----------	------	----

Probe von S₁ gegen Hashtabelle ergibt Ergebnisse:

12	Fast PC	42	Dittrich	Jens	32
36	Stapler	42	Dittrich	Jens	32

Kompletter Join: Phase 2, Paar (R₂,S₂)

1. Lade zweite Partition R₂ von R in den HS und erzeuge Hashtabelle hierauf:

R₂ =

1	Slow PC	8
4	Cray	3

HS

2. Führe ein probe für jedes Tupel aus S₂ gegen die Hashtabelle aus

S₂ =

8	Müller	Peter	55
---	--------	-------	----

Probe von S₂ gegen Hashtabelle ergibt Ergebnis:

1	Slow PC	8	Müller	Peter	55
---	---------	---	--------	-------	----

Kompletter Join: Phase 2, Paar (R₃,S₃)

1. Lade dritte Partition R₃ von R in den HS und erzeuge Hashtabelle hierauf:

R₃ =

8	Fire Blade	7
---	------------	---

HS

2. Führe ein probe für jedes Tupel aus S₃ gegen die Hashtabelle aus

S₃ =

7	Dittrich	Klaus	43
6	Meier	Hans	37

Probe von S₃ gegen Hashtabelle ergibt Ergebnis:

8	Fire Blade	7	Dittrich	Klaus	43
---	------------	---	----------	-------	----

Gesamtergebnis des Joins

(R₁,S₁)

12	Fast PC	42	Dittrich	Jens	32
36	Stapler	42	Dittrich	Jens	32

(R₂,S₂)

1	Slow PC	8	Müller	Peter	55
---	---------	---	--------	-------	----

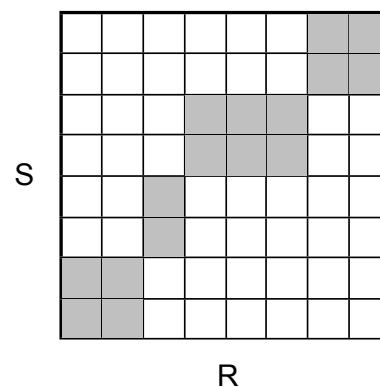
(R₃,S₃)

8	Fire Blade	7	Dittrich	Klaus	43
---	------------	---	----------	-------	----

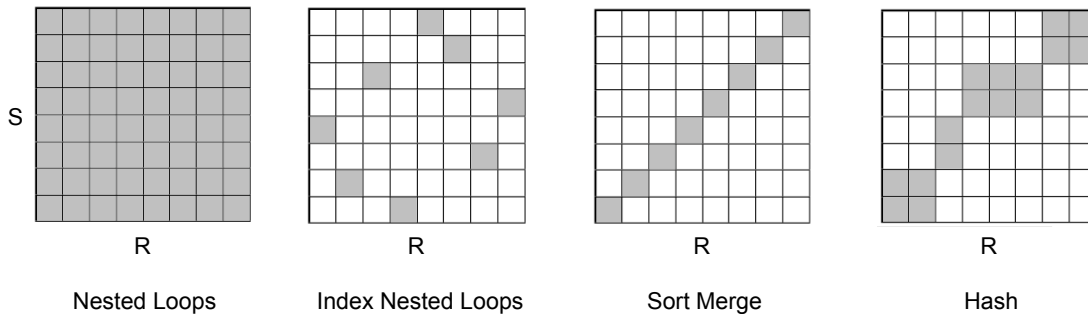
- Achtung: Tupel werden nicht sortiert nach Joinschlüssel produziert (wir haben ja auch die Daten nicht sortiert)

Bewertung

- Immer die kleinere Relation als Build Input verwenden
- Grace Hash Join ist oft das beste Verfahren für relationale Joins
- aber: nur bei Equi-Joins einsetzbar!
- Einige Tricks:
 - Bloom-Filter zum Vorfiltern der äusseren Tabelle verwenden
 - Rolle der inneren und äusseren Tabelle nach Bedarf tauschen
 - Falls möglich, Partitionen dynamisch zusammenfassen



Übersicht: Tupel-Vergleichstrategien



Siehe auch: Priti Mishra, Margaret H. Eich: Join Processing in Relational Databases. ACM Computing Surveys 24(1), 1992: 63-113

Nächste Woche

Fortsetzung

Desweiteren: Implementierung

Nicht-Relationaler Operatoren

