Vorlesung Datenbanksysteme vom 3.11.2008

Anfragebearbeitung 2

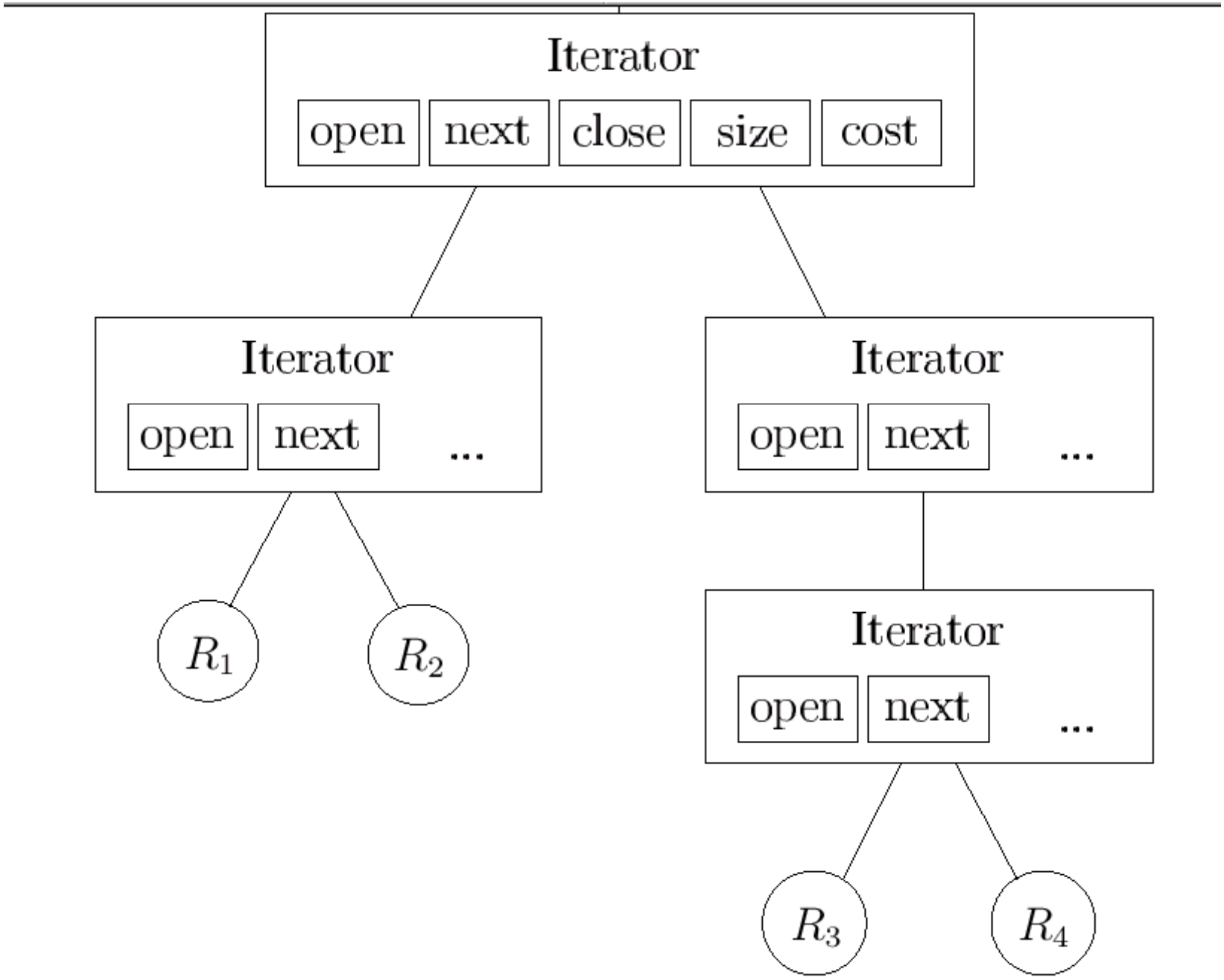
- Architektur eines DBMS
- Logische Optimierung
- Physische Optimierung
- Kostenmodelle + „Tuning“

Physische Optimierung

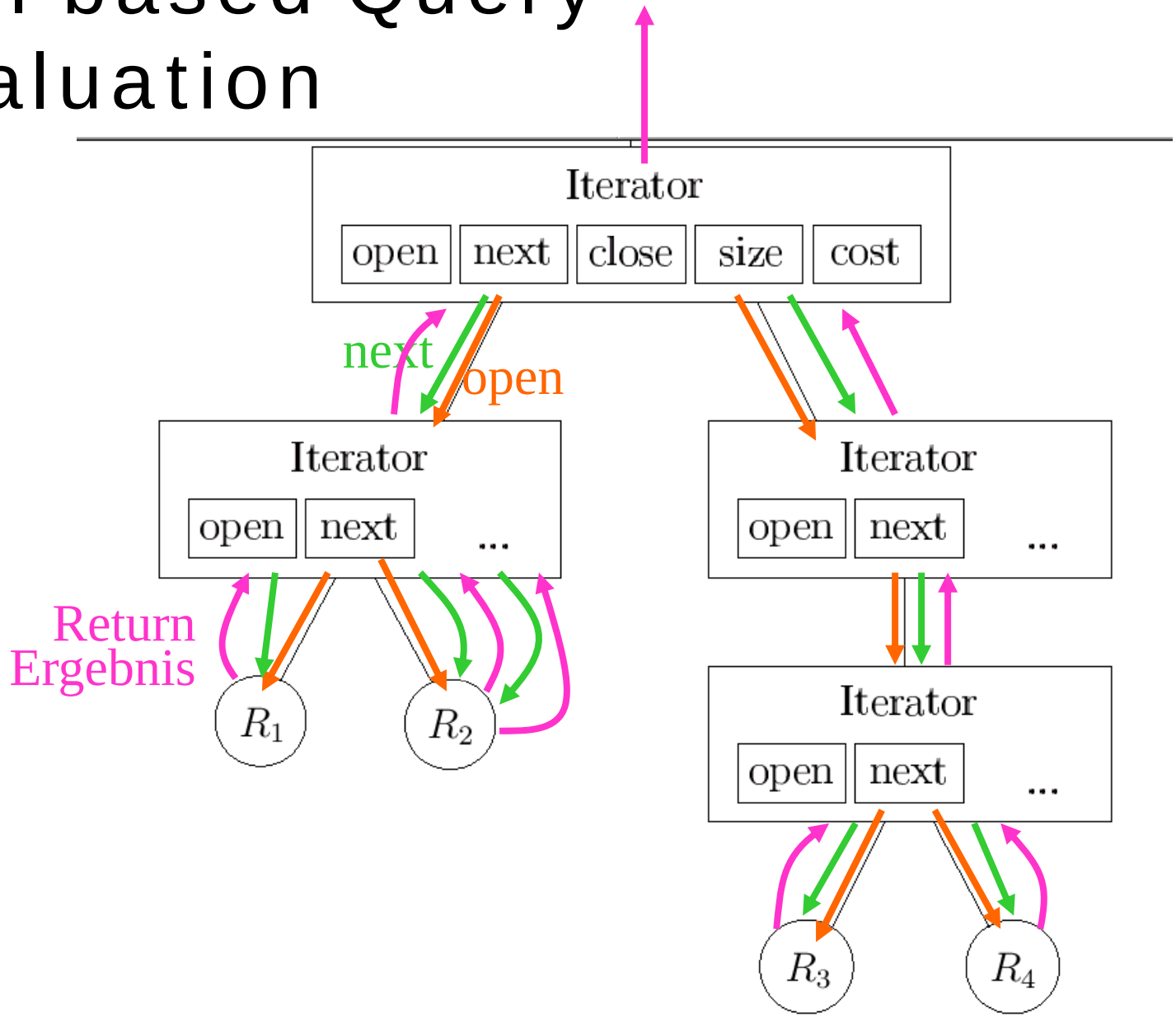
- Iterator: einheitliche Schnittstelle für die Operator-Auswertung
- Weiterreichen der Zwischenergebnisse
- Externes Sortieren
- Join-Implementierungen
- Weitere Operationen

Iterator

- Idee: Definition einer einheitlichen Schnittstelle für die Operator-Auswertung. $\Rightarrow$ Wesentliche Vereinfachung für die Steuerung der Anfrage-Auswertung.
- Die Methoden dieser Schnittstelle:
  - Open: Eingabe öffnen, Initialisierungen (falls nötig)
  - Next: liefert jeweils das nächste Tupel des (Zwischen-) Ergebnisses
  - Close: Eingabe schließen, Ressourcen frei geben (falls nötig)
  - Eventuell zusätzliche Methoden, z.B.: Cost, Size
- Bemerkung: Die Iterator-Beschreibungen im Kemper-Buch sind zum Teil fehlerhaft (insbesondere Abb. 8.14 "Merge-Join").


Iterator
open
next
close
size
cost
Iterator
open
next
...
Iterator
open
next
...
R_1
R_2
Iterator
open
next
...
R_3
R_4

Pull-based Query Evaluation

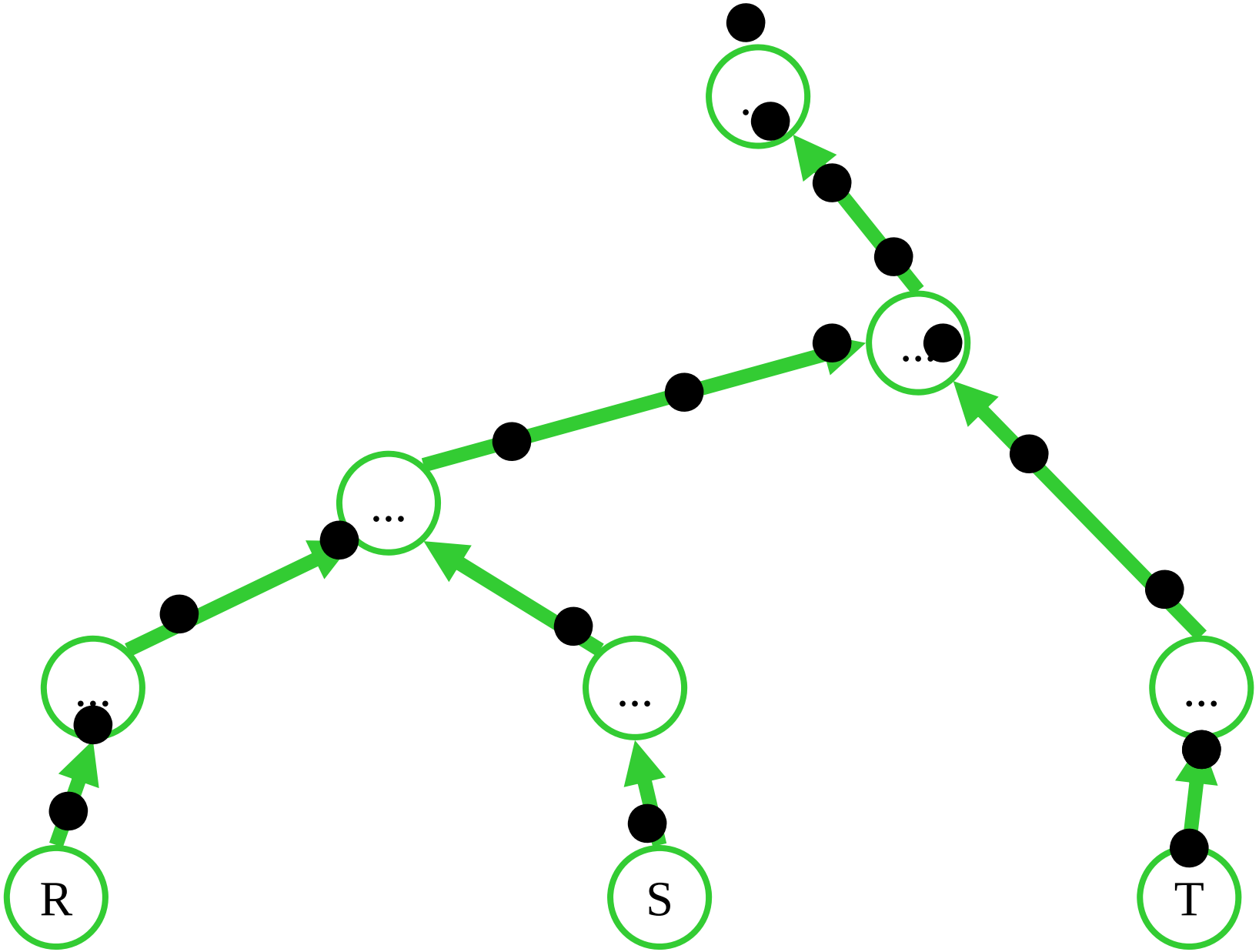


Weiterreichen der Zwischenergebnisse: „Pipelining“ vs. „Materializing“

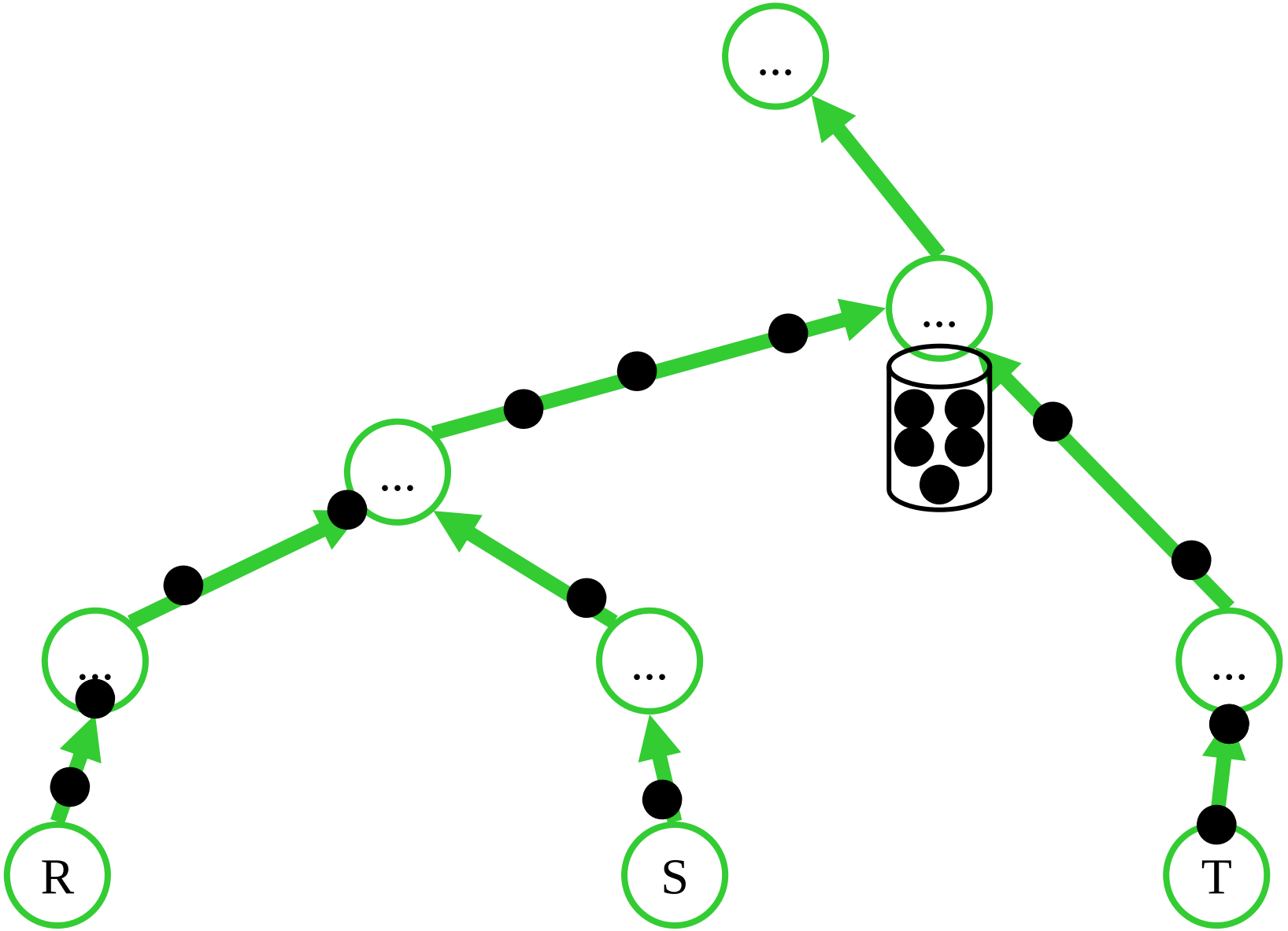
Idee:

- Ergebnis der Auswertung jeder relationalen Operation ist selbst wiederum eine Relation (d.h.: Menge von Tupeln).
- Diese Menge von Tupeln stellt gleichzeitig den Input für die nächste relationale Operation im Auswertungsplan dar.
- Zwei Alternativen für das Weiterreichen dieser Tupeln:
  1. "Materializing": Das gesamte Zwischenergebnis wird in einer Hilfs-Tabelle zwischengespeichert.
  2. "Pipelining": Direktes Weiterreichen der einzelnen Tupeln ohne Zwischenspeichern => erspart I/O für das Schreiben und Wiederauslesen.

Pipelining vs. Materializing



Pipelining vs. Materializing



Typische „Pipeline-Breaker“

- Wichtige Routinen, die Ergebnisse zwischenspeichern:
  - Sortieren
  - Hashing
- Operationen, die (je nach Implementierung) Zwischenspeichern erfordern:
  - Join
  - Duplikatelimination
  - Gruppieren
  - Mengendifferenz, Durchschnitt

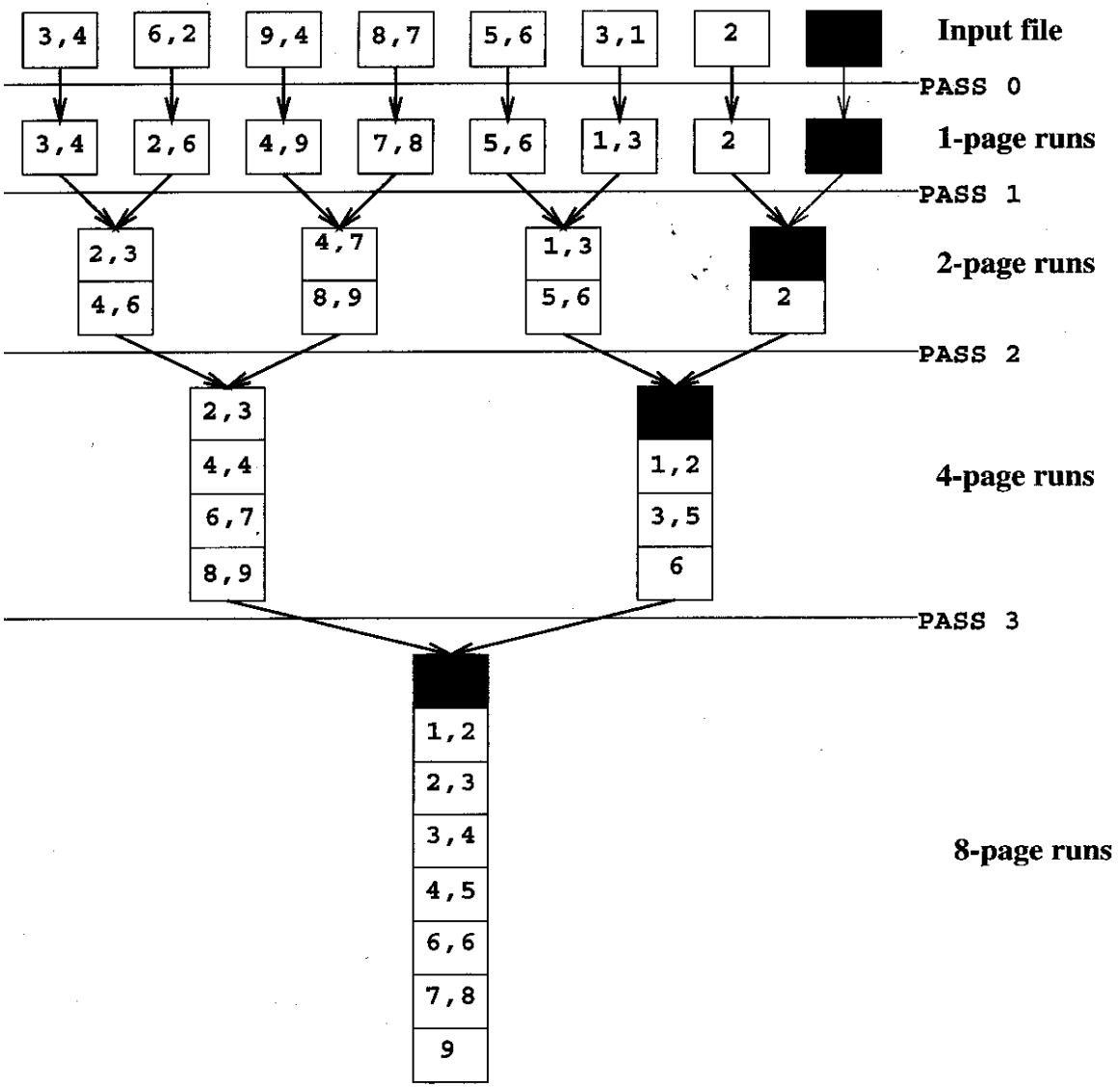
Externes Sortieren

- Anwendungsfälle:
  - Sortieren ist keine relationale Operation, wird aber für manche Implementierungen benötigt, z.B.: Sort-Merge Join.
  - Außerdem ist Sortieren erforderlich bei "order by" Klausel und eine Möglichkeit der Duplikat-Elimination.
- "Externes" Sortieren:
  - Sortieren von großen Datenmengen (die größten Teils auf dem Hintergrundspeicher abgelegt sind) erfordert andere Methoden als "in-memory" Sortieren.
  - Idee: Eigentliche Sortierung erfolgt im Hauptspeicher; "Merge Sort": Sortierte Teile werden zusammengefügt (unter Aufrechterhaltung der Sortierung).

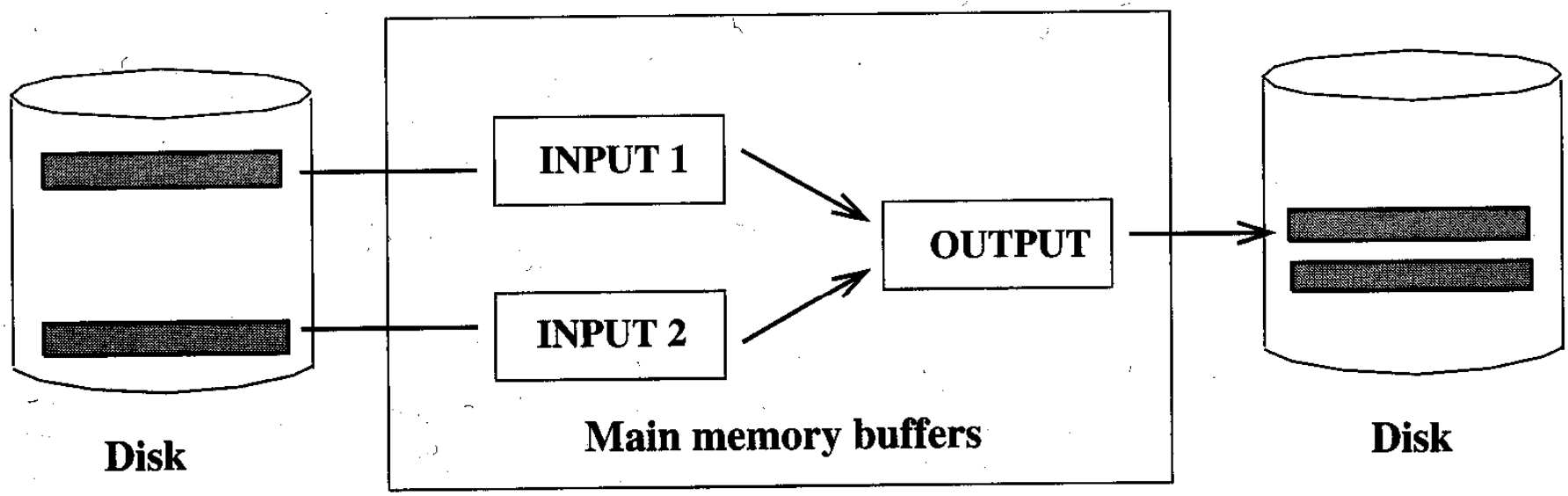
Algorithmen für Externes Sortieren

- Grundidee: Einfacher Two-Way Merge Sort
  - 3 Puffer-Seiten im Hauptspeicher genügen
  - Initialisierung (Pass 0): Jede einzelne Seite des Files wird eingelesen, sortiert und wieder ausgeschrieben.
  - Pass 1: je 2 sortierte Teilstücke (= "Runs") à 1 Seite werden zu einem Run der Länge 2 gemerged.
  - Pass 2: je 2 sortierte Teilstücke (= "Runs") à 2 Seiten werden zu einem Run der Länge 4 gemerged, etc.
  - Weitere Passes bis 1 Run das ganze File umfasst.
- Externes Sortieren mit realistischer Hauptspeichergröße:
  - Initialisierung: Größe der sortierten Teilstücke (= "Runs") entspricht der Anzahl der verfügbaren Puffer-Seiten
  - Multi-Way Merge Sort: Kombiniere m-1 Runs

Einfacher Two-Way Merge Sort



Einfacher Two-Way Merge Sort



Einfacher Two-Way Merge Sort

Algorithmus:

```
// Pass 0: Erzeuge Runs, die je 1 Seite lang sind.
   for each page of the file do {
         read page; sort it (in RAM); write it out; }

// Merge 2 Runs zu größerem Run, bis gesamtes File sortiert ist.
   while number of runs after previous pass is > 1 {
   //  Pass i = 1, 2, ...:
         while ∃ runs to be merged from previous pass {
            choose next two runs;
            read (1 page at a time) each run into an input buffer;
            merge the runs and write to output buffer;
            write output buffer to disk (1 page at a time); }}
```

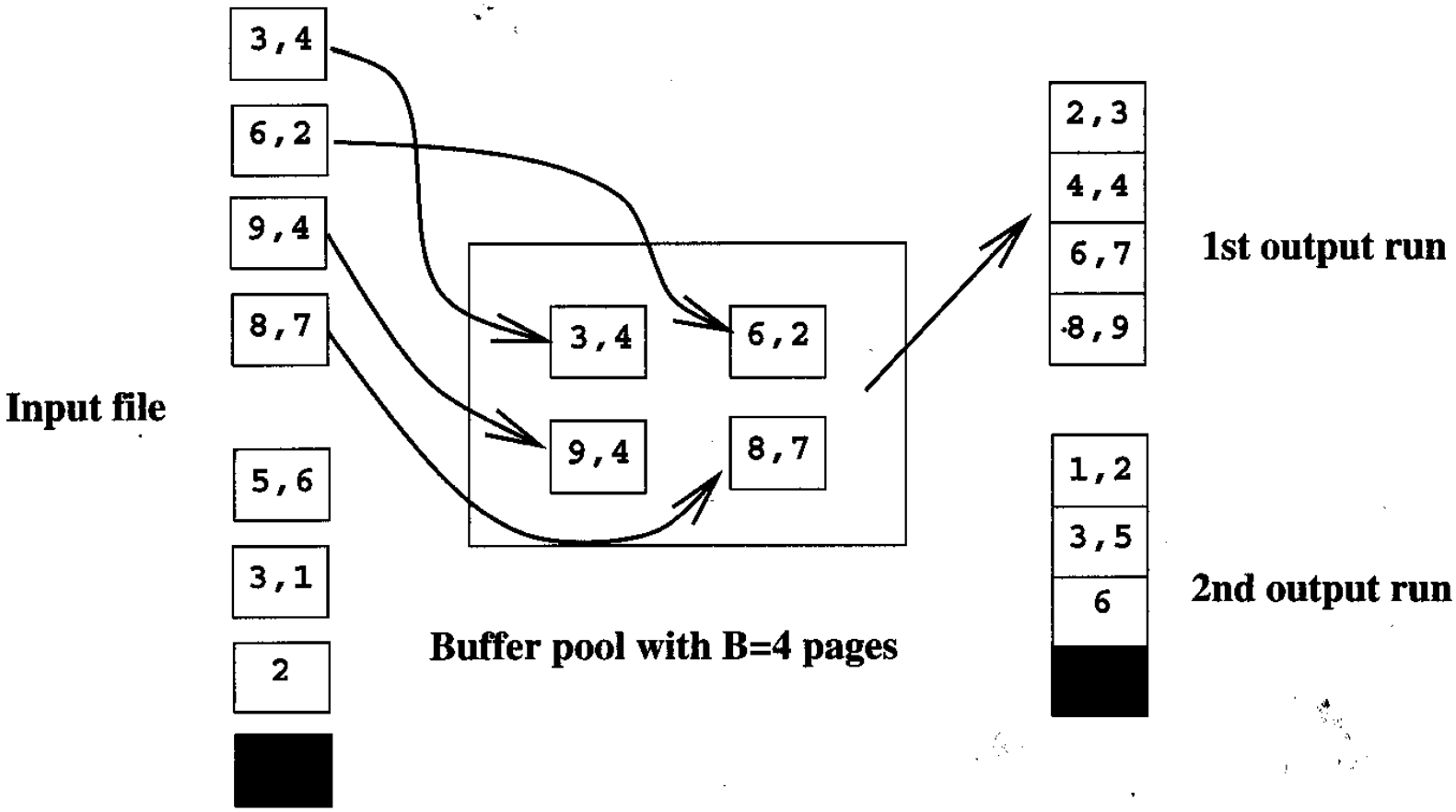
Externes Sortieren mit m Puffer-Seiten

Aufgabe: Externes Sortieren eines Files bestehend aus N Seiten mit Hilfe eines Puffers von m Seiten (im allgemeinen m << N).

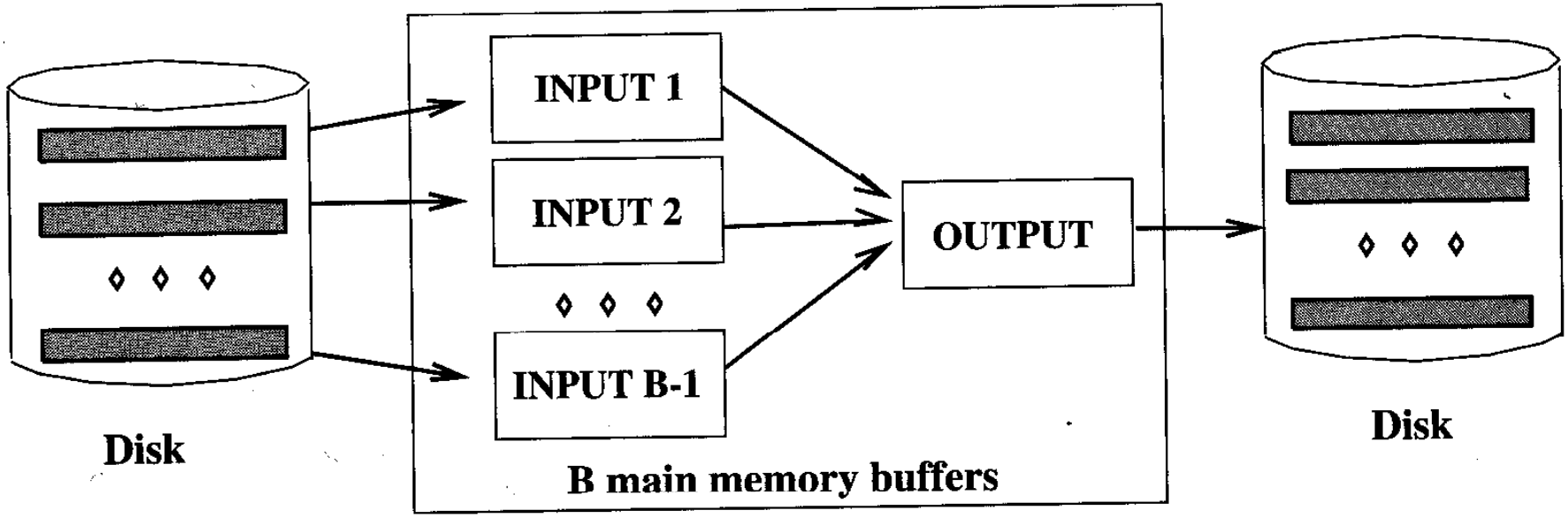
Modifikationen des einfachen Two-Way Merge Sort:

- Initialisierung (Pass 0):
  - Es werden Blöcke von je m Seiten in den Puffer gelesen, sortiert und wieder ausgeschrieben.
    => $\lceil N / m \rceil$ Runs zu je m Seiten (außer ev. der letzte Run)
- Eigentlicher Merge-Sort (Pass i = 1, 2, ...):
  - Es werden jeweils m-1 Runs gemerged.
  - Je eine Seite dieser m-1 Runs steht im Puffer (= Input)
  - Für den Output wird nach wie vor nur 1 Seite benötigt.

Externes Sortieren mit B Puffer-Seiten



Externes Sortieren mit B Puffer-Seiten



Join-Implementierungen

- Implementierungs-Varianten für $R \bowtie_{A=B} S$:
  1. Nested Loop Join
  2. Verfeinerungen: Page-oriented + Block Nested Loop Join
  3. Index Nested Loop Join
  4. Sort Merge Join
  5. Hash Join
  6. Verfeinerung: Hybrid Hash Join

- Join mit allgemeinen Join-Bedingungen:
  - Konjunktionen $A_1 = B_1 \wedge A_2 = B_2 \wedge \ldots \wedge A_n = B_n$
  - Ungleichungen $A < B$, $A \leq B$, $A \neq B$, etc.

Equi-Join: Beispiel

R = Vorlesungen, A = gelesenVon

S = Professoren, B = PersNr

Vorlesungen $\bowtie_{\text{gelesenVon = PersNr}}$ Professoren							
Vorlesungen				Professoren			
VorlNr	Titel	SWS	Gelesen Von	PersNr	Name	Rang	Raum
5001	Grundzüge	4	2137	2137	Kant	C4	7
5041	Ethik	4	2125	2125	Sokrates	C4	226
5043	Erkenntnistheorie	3	2126	2126	Russel	C4	232
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
...	...	...	...	...	...	...	...

Nested Loop Join

"brute force"-Algorithmus:

```
foreach tuple r ∈ R
   foreach tuple s ∈ S
     if s.B = r.A  then Res := Res ∪ {(r,s)}
```

iterator NestedLoop$_p$

open

- Öffne die linke Eingabe

next

- Rechte Eingabe geschlossen?
  - Öffne sie
- Fordere rechts solange Tupel an, bis Bedingung p erfüllt ist
- Sollte zwischendurch rechte Eingabe erschöpft sein
  - Schließe rechte Eingabe
  - Fordere nächstes Tupel der linken Eingabe an
  - Starte **next** neu
- Gib den Verbund von aktuellem linken und aktuellem rechte Tupel zurück

close

- Schließe beide Eingabequellen

Page-oriented Nested Loop Join

Idee:

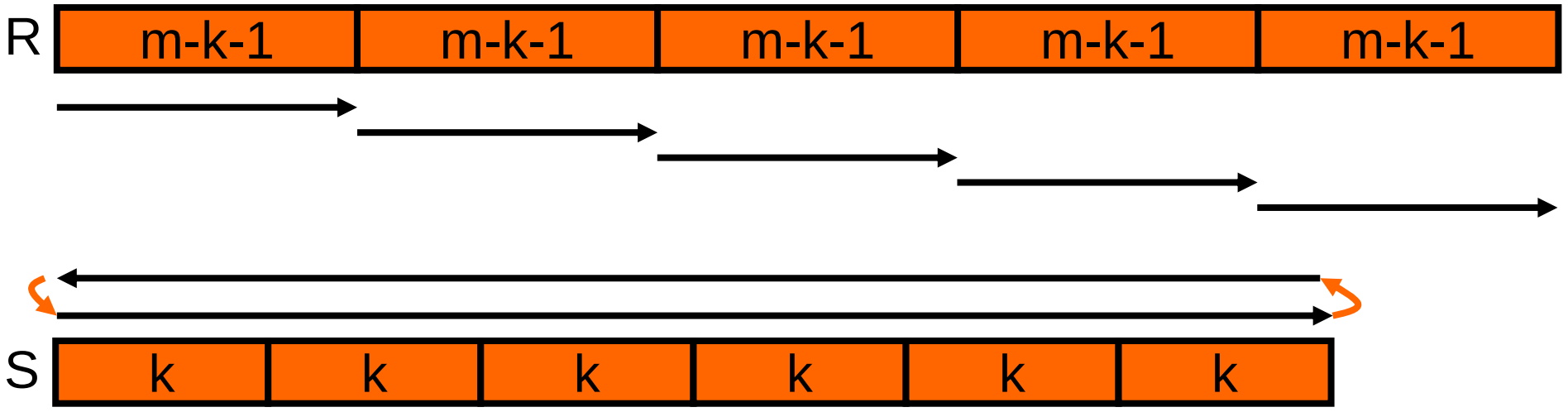
Für jede Seite p_R von R und jede Seite p_S im Puffer werden alle möglichen Kombinationen von Tupeln $r \in p_R$ und $s \in p_S$ getestet.

```
foreach page p_R of R
  foreach page p_S of S
    foreach tuple r ∈ p_R and s ∈ p_S
      if s.B = r.A then Res := Res ∪ {(r,s)}
```

Block Nested Loop Join

Idee:

- m Seiten im Puffer: k Seiten für innere Relation (einfachster Fall: k=1), m-k-1 Seiten für äußere Relation, 1 Seite für Output.
- (wie page-oriented NL): Teste jede Kombination von Tupeln $r \in R$ $s \in S$, die sich gerade im Puffer befinden. (In der Praxis wird dies mittels in-memory Hash Table für den Block von R realisiert).
- (weitere Verbesserung): Durch "Zick-Zack" Abarbeitung von S erspart man sich (ab dem 2. Durchlauf) 1 I/O pro Durchlauf von S.



Index Nested Loop Join

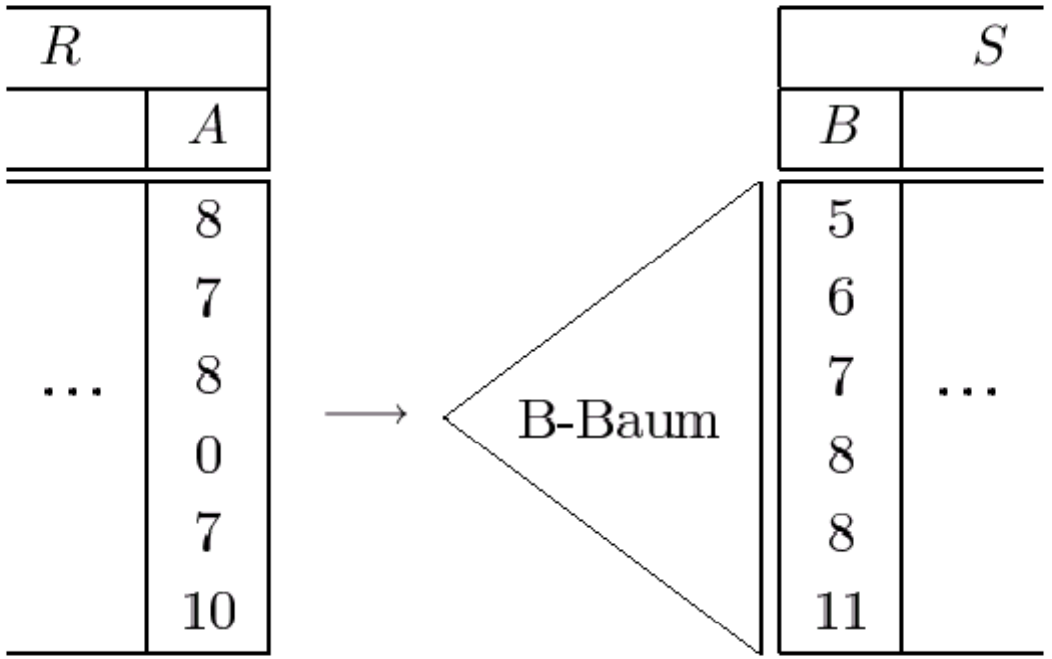
Idee:

- Beim Durchlauf von *R* werden nur die in *S* qualifizierenden Tupel gelesen.
- Dazu ist ein Index auf *B* erforderlich.

```
foreach  r ∈ R
  foreach s ∈ S[B=r.A]
       Res := Res ∪ {(r,s)}
```

Index-Join

Beispiel:



Sort-Merge Join

Idee:

- erfordert zwei Sortierungen
  1. *R* muss nach *A* und
  2. *S* muss nach *B* sortiert sein.
- Falls *A* oder *B* Schlüsselattribut ist, wird jedes Tupel in *R* und *S* nur genau einmal gelesen.

R

…	A
…	5
…	5
…	5
…	6
…	6
…	6
…	7
…	7
…	7

S

B	…
4	…
4	…
4	…
5	…
5	…
6	…
7	…
7	…
7	…
8	…

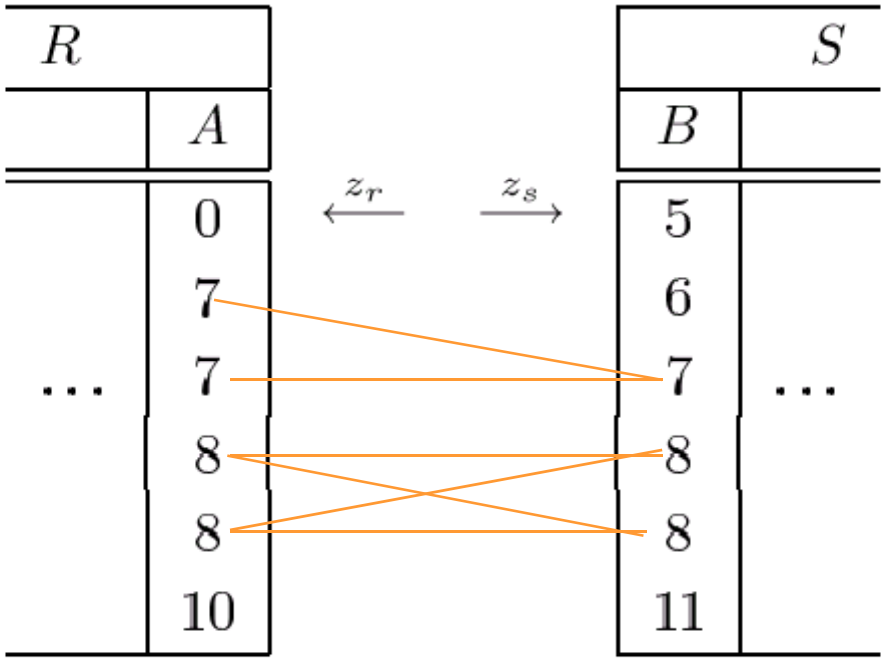
Ergebnis:

…A	B…
…5	5…
…5	5…
…5	5…
…5	5…
…5	5…
…5	5…
…6	6…
…6	6…
…6	6…
…7	7…

Der Merge-Join

- Voraussetzung: R und S sind sortiert (notfalls vorher sortieren)

Beispiel:



Hash Join

Idee: (für m Seiten Puffergröße)

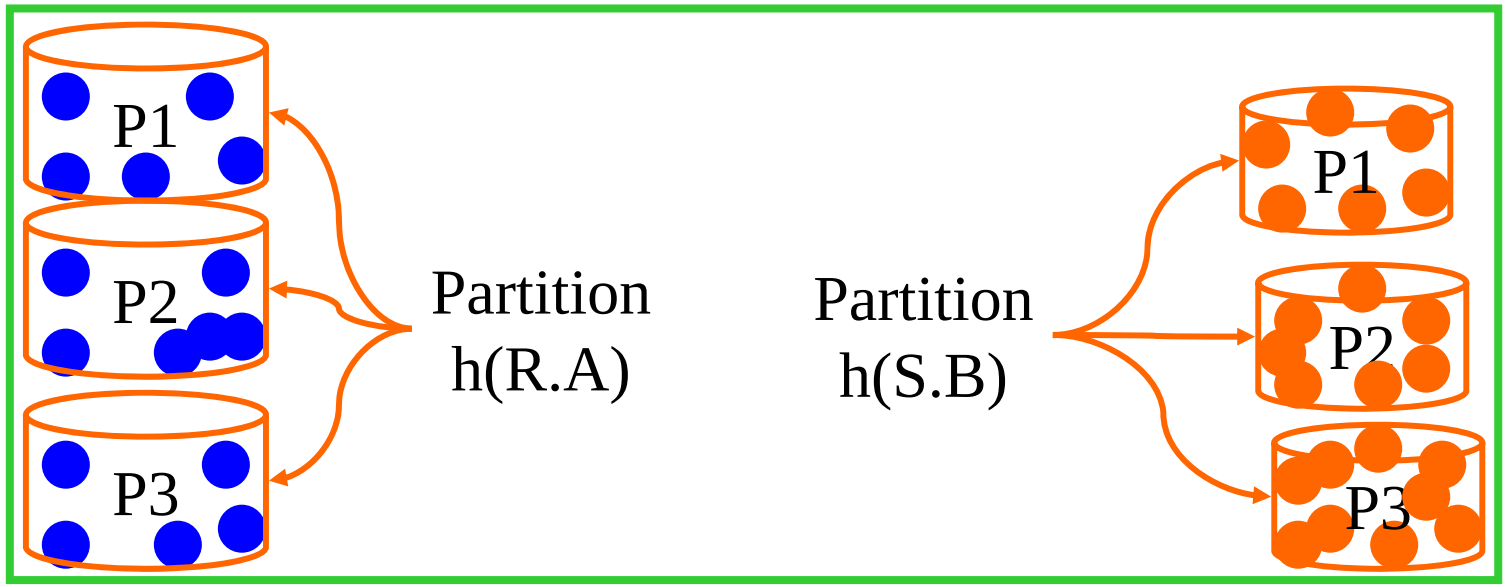
Phase 1: "Build" (oder "Partition")-Phase:

- *R* und *S* werden mit Hilfe der gleichen Hashfunktion *h1* (angewendet auf *R.A* bzw. *S.B*) in m-1 Buckets partitioniert.
- (Angenommen S benötigt weniger Seiten als R):
  Falls eines der Buckets von S mehr als m-2 Seiten braucht, dann muss dieses Bucket (sowohl von R als auch von S) mit einer anderen Hashfunktion h2 weiter unterteilt werden.

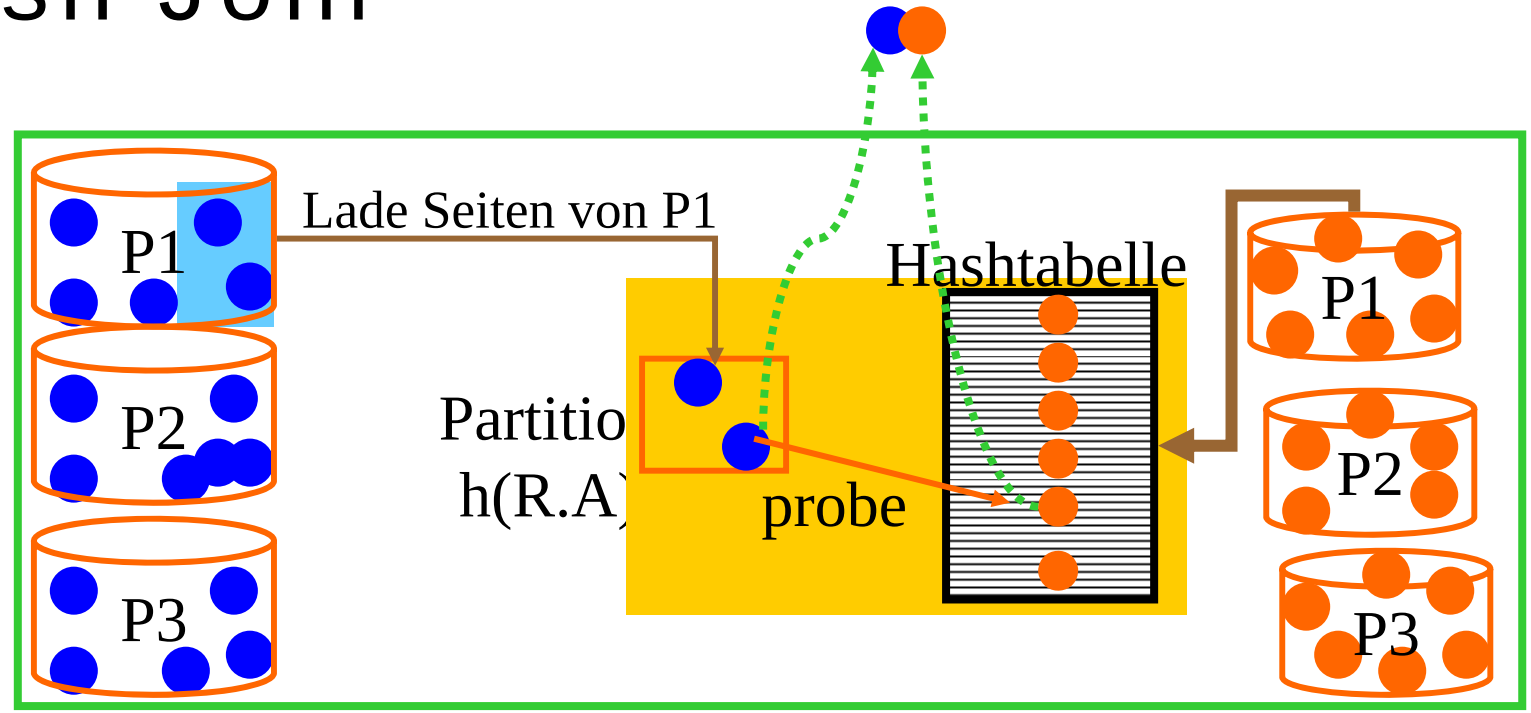
Phase 2: "Probe" (oder "Matching")-Phase:

- Lade jeweils 1 Bucket von S in den Puffer
- Lade vom entsprechenden Bucket von R eine Seite nach der anderen in den Puffer und teste jedes Tupel von dieser Seite von R mit den Tupeln von S (In der Praxis: Realisierung mittels in-memory Hash Table für das Bucket von S).

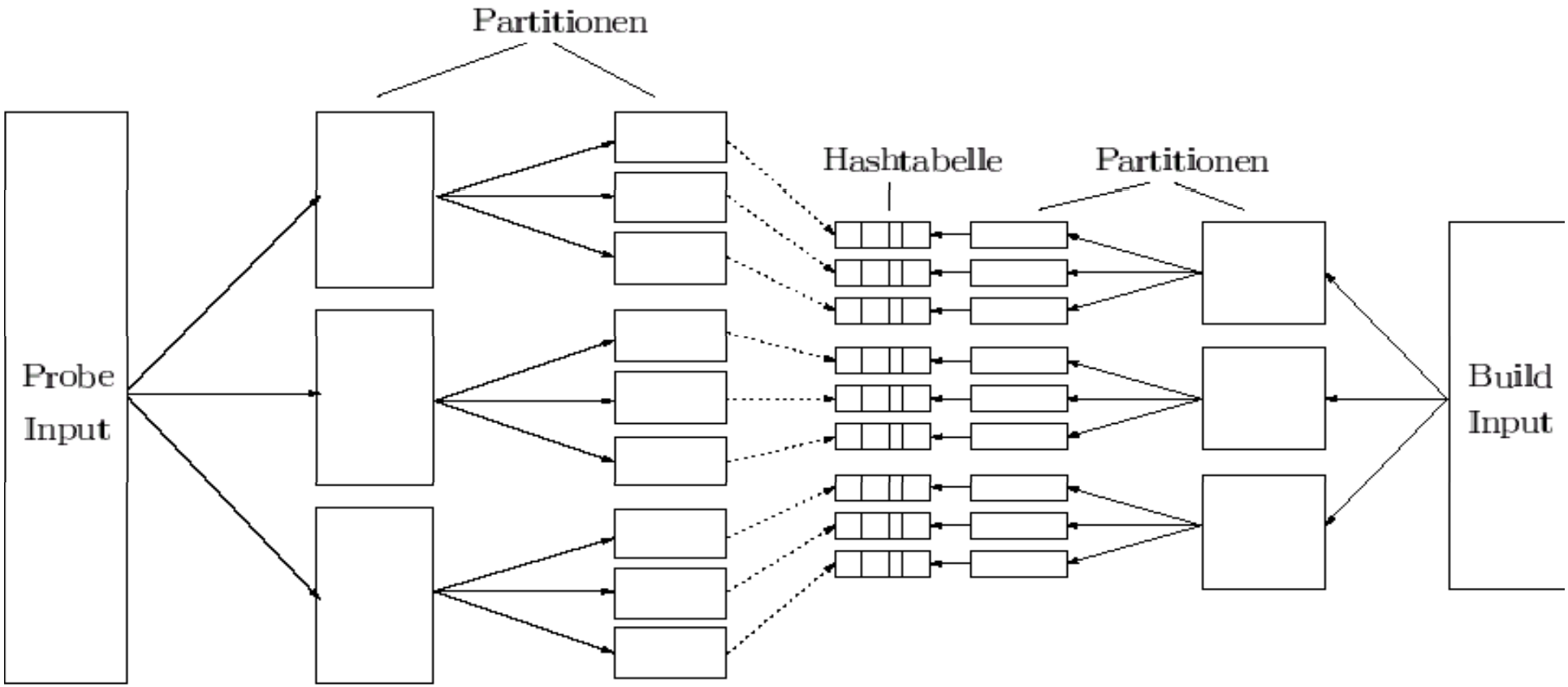
Hash Join



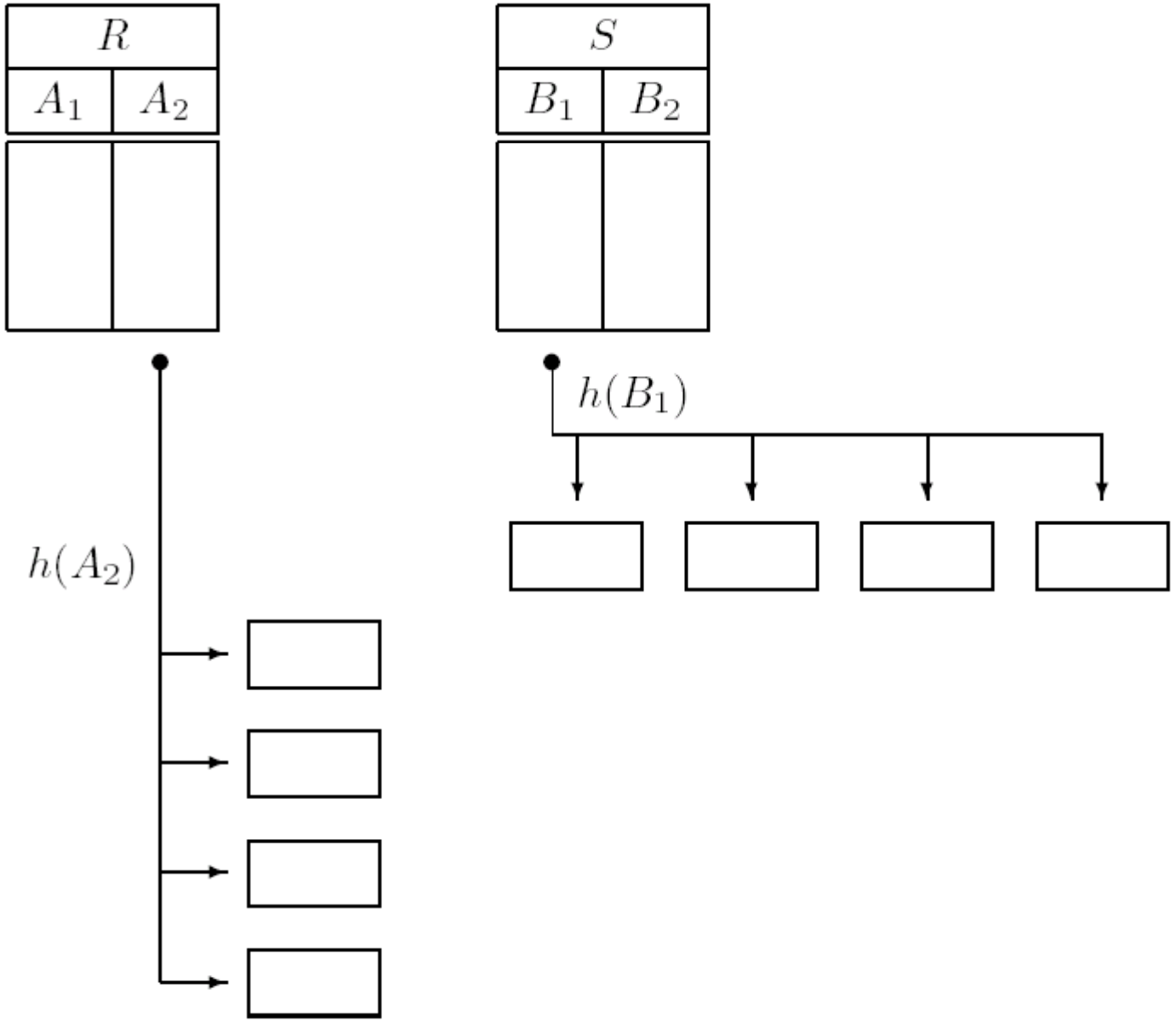
Hash Join

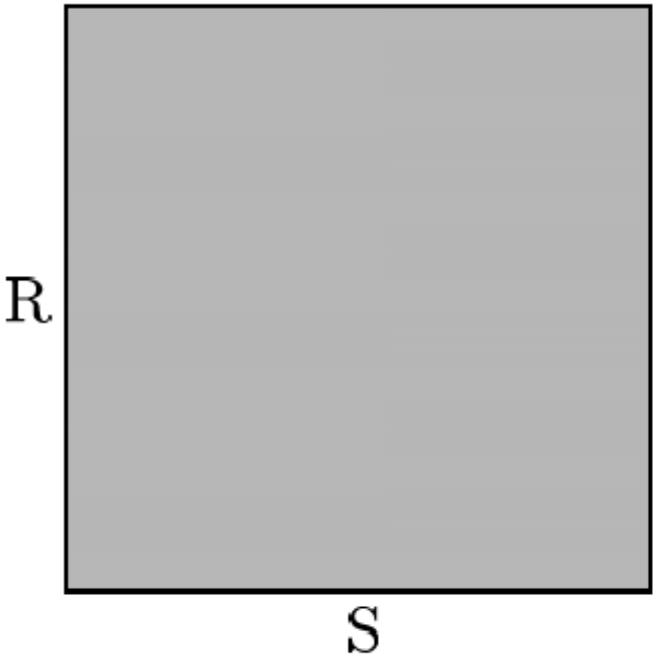

P1
P2
P3
Lade Seiten von P1
Partitio
h(R.A)
Hashtabelle
probe
P1
P2
P3

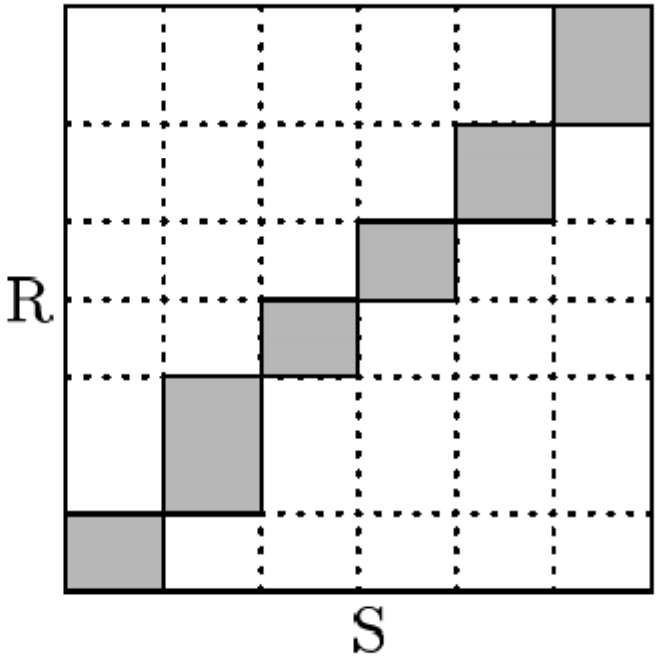
Partitionierung von Relationen



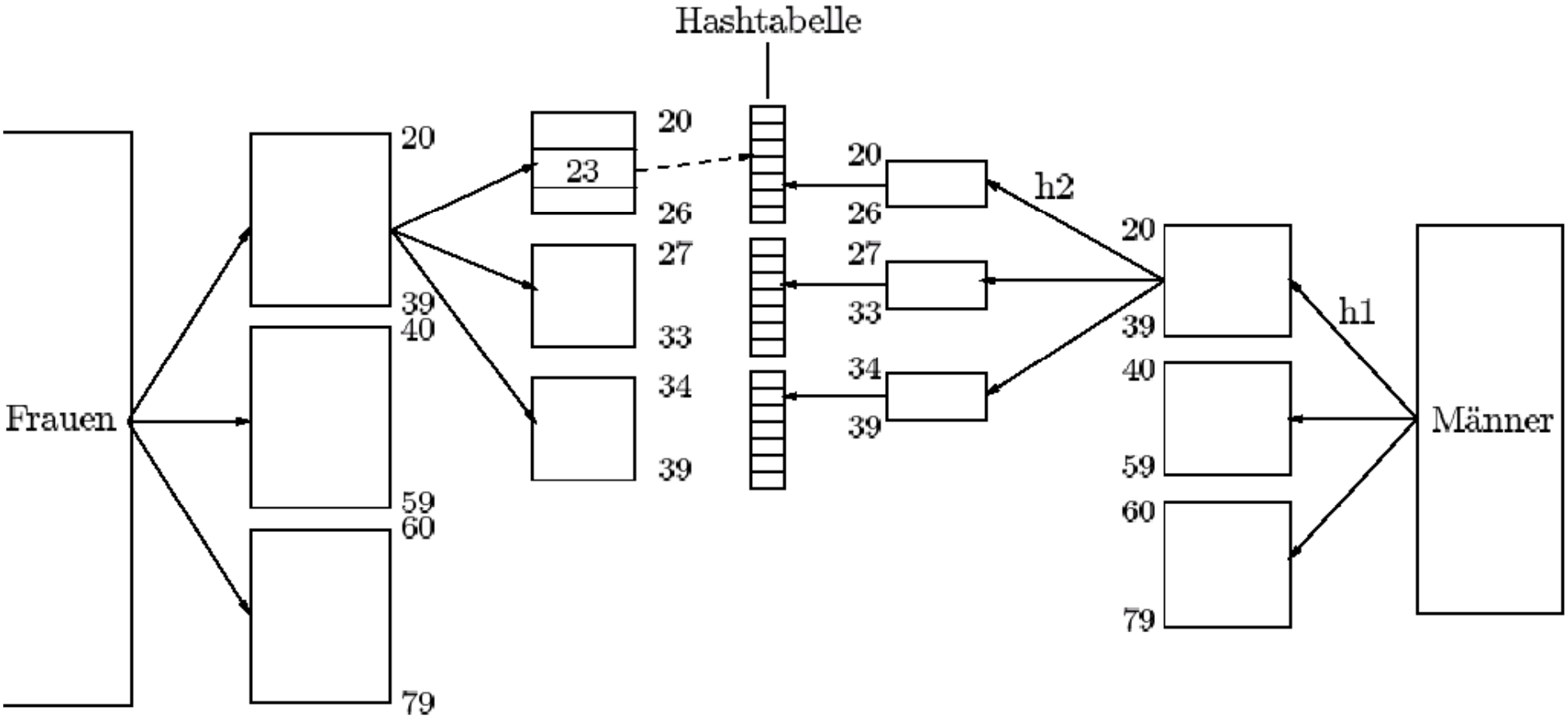
Vergleich der Tupel in der "Diagonalen"



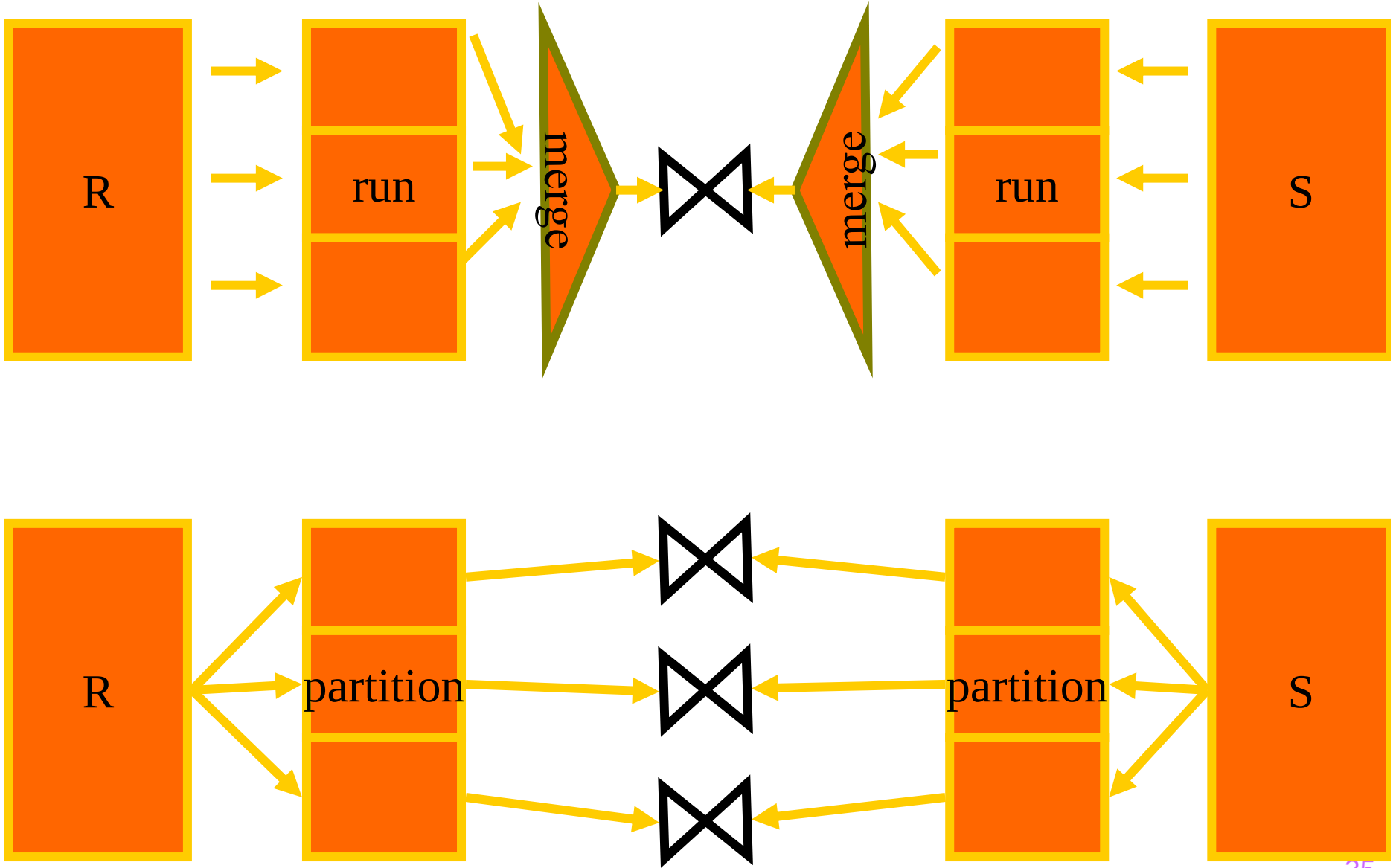

R
S


R
S

Demonstration der Partitionierung



Vergleich: Sort/Merge-Join versus Hash-Join



Hybrid Hash Join

Idee:

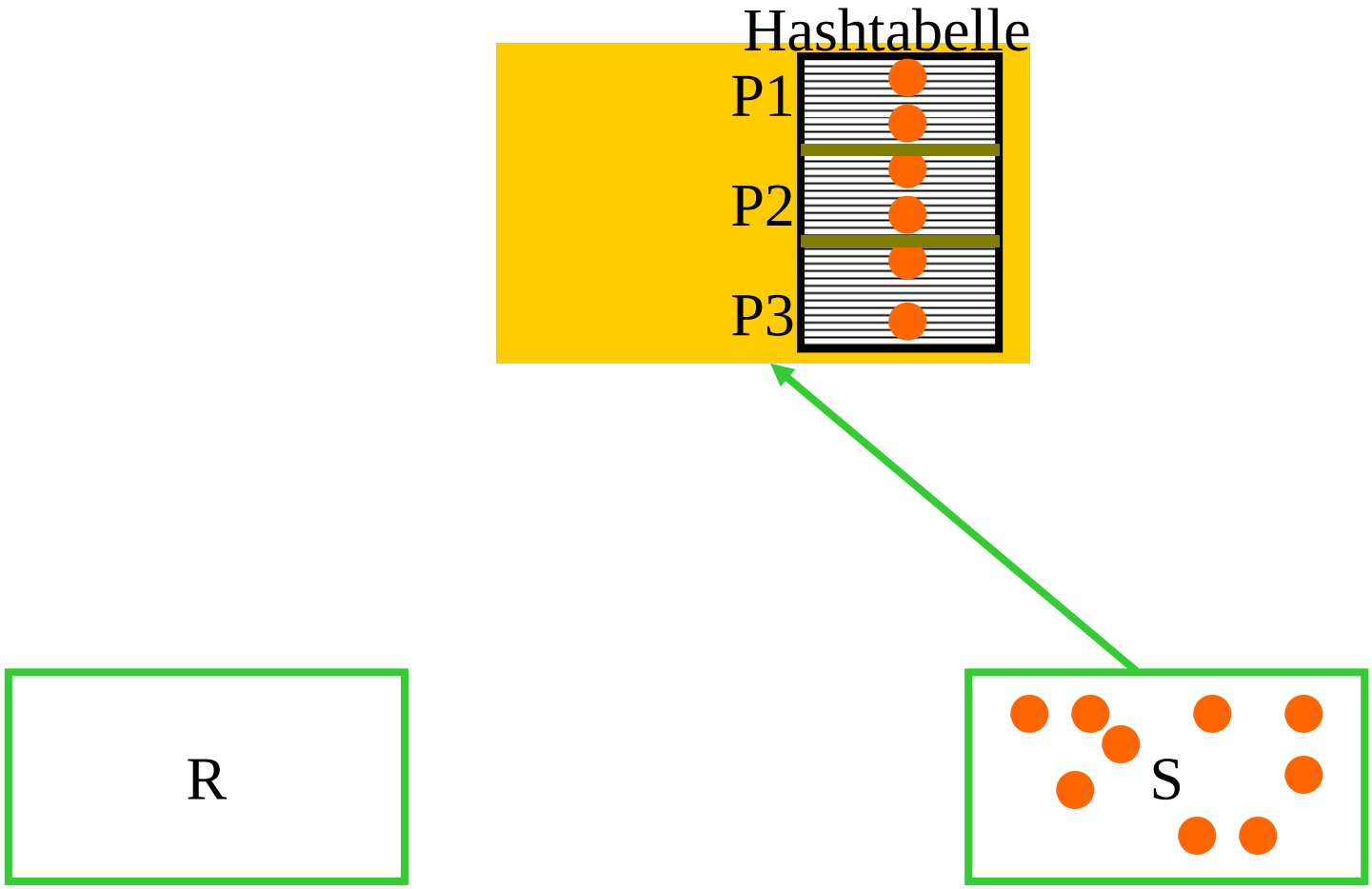
- Wenn im Hauptspeicher ausreichend Platz ist, kann man eventuell während der Build-Phase von S einige Partionen von S (als in-memory Hash-Table) im Puffer lassen.
- Während der Build-Phase von R kann man dann alle Tupel, deren potentielle Join-Partner im Hauptspeicher sind, auf der Stelle verarbeiten.
- Ersparnis: Wenn k Partitionen von S im Puffer Platz haben, erspart man sich für diese k Partitionen – sowohl von R als auch von S – das Ausschreiben (während der Build-Phase) und das Einlesen (während der Probe-Phase).

Hybrid Hash Join

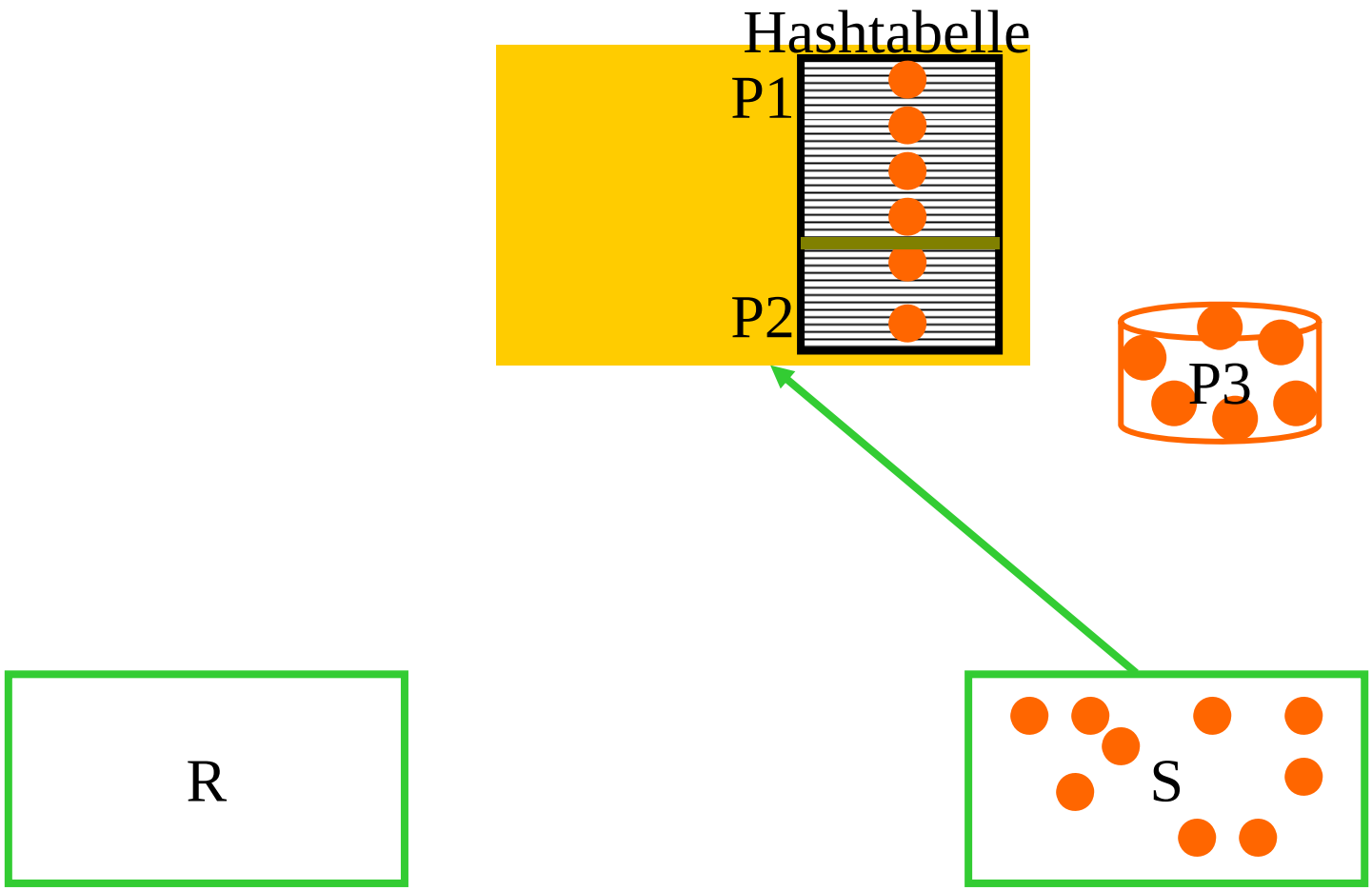
Algorithmus:

- Fange so an, als ob der Build-Input S vollständig in den Hauptspeicher passen würde.
- Sollte sich dies als zu optimistisch herausstellen, verdränge eine Partition nach der anderen aus dem Hauptspeicher.
- Mindestens eine Partition wird aber im Hauptspeicher verbleiben (Ansonsten haben wir einen "normalen" Hash Join).
- Während der Build-Phase von R werden alle Tupel aus R, deren potentielle Join-Partner im Hauptspeicher sind, sofort verarbeitet.
- Danach beginnt die "normale" Probe-Phase mit den restlichen Partitionen.

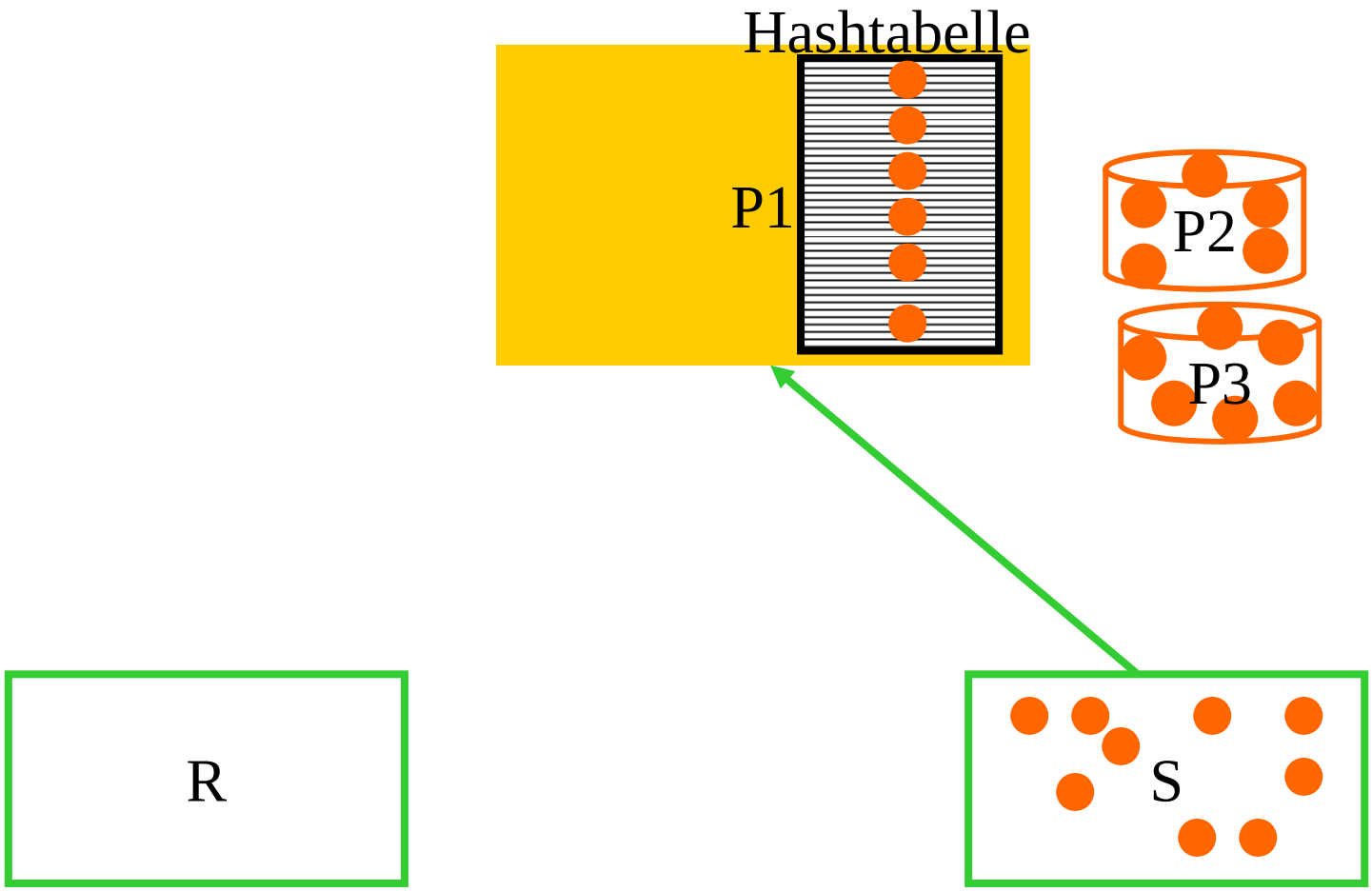
Hybrid Hash Join



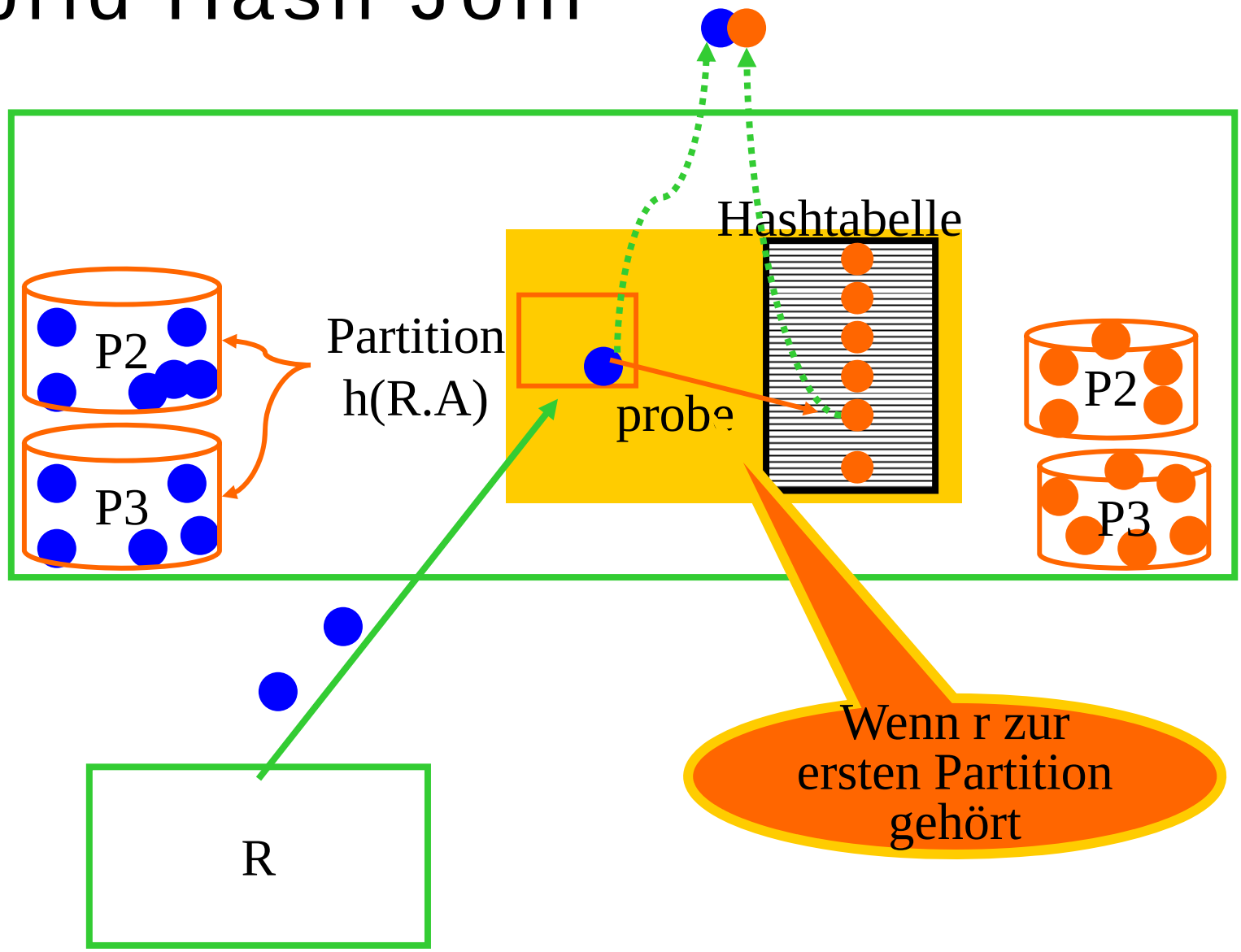
Hybrid Hash Join



Hybrid Hash Join



Hybrid Hash Join



Allgemeine Join-Bedingungen

Konjunktionen $A_1 = B_1 \wedge A_2 = B_2 \wedge \ldots \wedge A_n = B_n$

- Nested Loop Join (+ Verfeinerungen): unverändert.
- Index Nested Loop Join:
  Angenommen S ist die innere Relation: Dann benötigt man einen Index für die Kombination der Attribute $(B_1, \ldots, B_n)$.
- Sort Merge Join:
  Sortiere R nach der Kombination der Attribute $(A_1, \ldots, A_n)$ und S nach der Kombination der Attribute $(B_1, \ldots, B_n)$.
- (Hybrid) Hash Join:
  Hashing von R mittels Kombination der Attribute $(A_1, \ldots, A_n)$ und von S mittels Kombination der Attribute $(B_1, \ldots, B_n)$.

Allgemeine Join-Bedingungen

Ungleichungen $A < B$, $A > B$, $A \leq B$, $A \geq B$, $A \neq B$

- Nested Loop Join (+ Verfeinerungen):
  Unverändert.
- Index Nested Loop Join:
  - Bei $A \neq B$ nicht anwendbar.
  - Bei $A < B$, $A > B$, $A \leq B$, $A \geq B$ benötigt man einen geballten Index für B.
- Sort Merge Join und (Hybrid) Hash Join:
  Sind in diesem Fall nicht anwendbar!

Weitere Operationen

- Selektion
- Projektion
- Duplikatelimination
- Mengenoperationen: R x S, R $\cap$ S, R $\cup$ S, R - S
- Gruppierung und Aggregat-Funktionen

Selektion

- "brute force":
  Sequentielles Durchlaufen (= Scan) des gesamten Files.
- Mit einem "passenden" Index:
  Beispiel: $\sigma_{\text{Name= 'Sokrates'} \wedge \text{Raum} > 300}$ (Professoren):
  -> Suche im Index für das Attribut Name nach 'Sokrates' und teste anschließend die Bedingung Raum > 300.
- Bei Sortierung (z.B.: Resultat eines Sort Merge Join):
  "logarithmisches" Suchen.
- Üblicherweise wird versucht, die Selektion mit einem anderen Schritt zu kombinieren, z.B.: im Rahmen eines Join, beim ersten Zugriff auf ein File, etc.

Projektion

- In der physischen Algebra werden bei Projektion keine Duplikate eliminiert.
- Falls eine Duplikatelimination gewünscht ist (select distinct), dann ist dieser Schritt (und die verwendete Methode) explizit anzugeben.
- Bei der Projektion werden daher einfach die Tupeln auf die gewünschten Attribute reduziert und weitergereicht.
- Üblicherweise wird versucht, die Projektion mit einem anderen Schritt zu kombinieren, z.B.: im Rahmen eines Join, beim ersten Zugriff auf ein File, etc.

Duplikatelimination

- Duplikatelimination mittels Sortierung:
  - Sortiere die Relation für die Kombination aller Attribute.
  - Vergleiche im sortierten File jeweils nur benachbarte Tupel.
- Duplikatelimination mittels Hashing:
  - Build-Phase: Hash-Funktion für Kombination aller Attribute
  - Für jedes Bucket wird anschließend eine in-memory Hash-Table (mit einer anderen Hash-Funktion) erzeugt.
    -> dabei entdeckte Duplikate werden sofort verworfen.

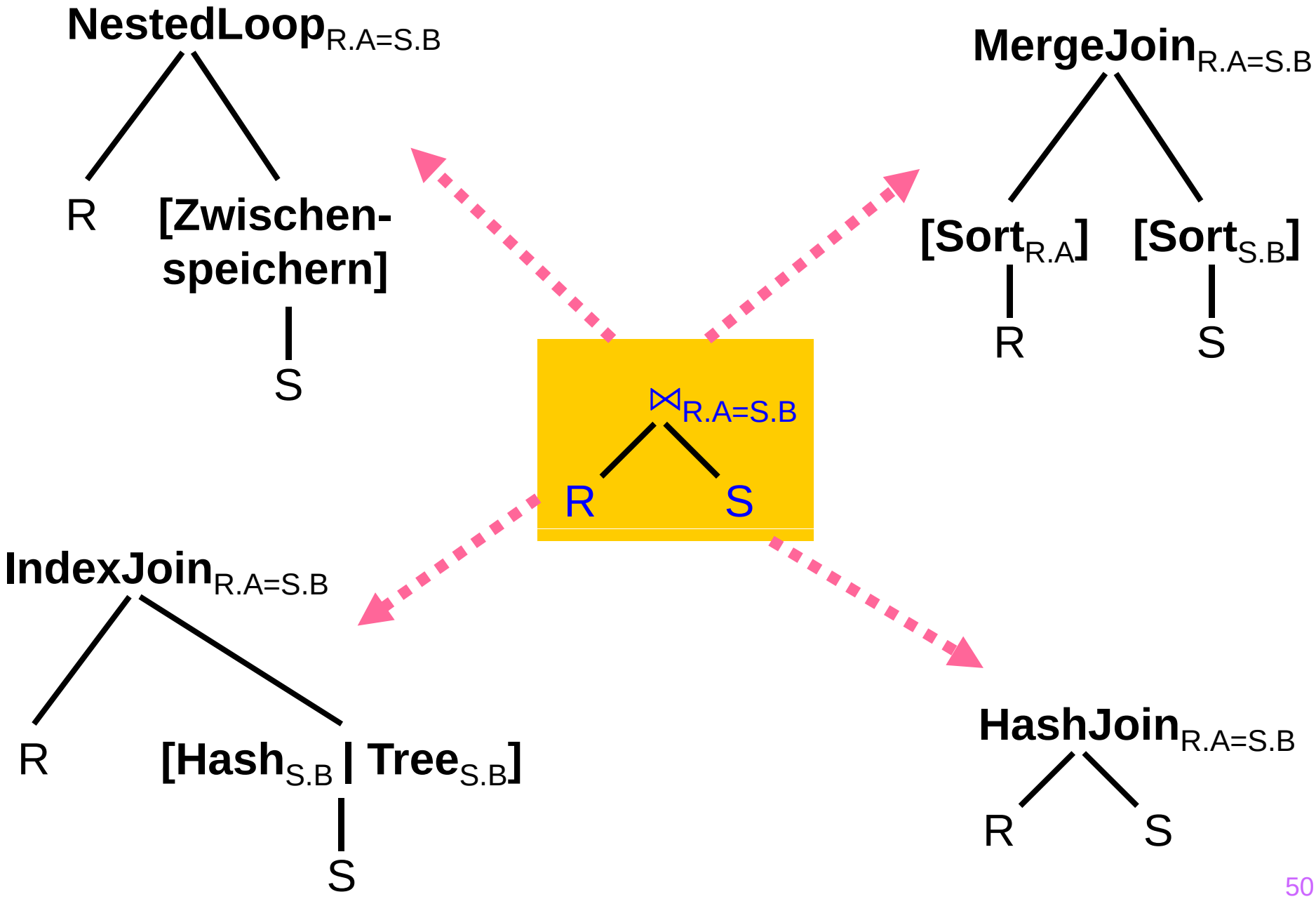
Mengenoperationen

- $R \times S$ und $R \cap S$ sind einfach Spezialfälle von Joins,
  d.h.: Join über keine Attribute bzw. über alle Attribute.
- In der physischen Algebra ist bei Vereinigung keine Duplikatelimination vorgesehen. => Vereinigungsoperator liest einfach beide Input-Relationen und gibt alle Tupeln weiter
- Mengendifferenz bzw. Vereinigung mit Duplikatelimination:
  - Mittels Sortierung für Kombination aller Attribute oder mittels Hashing
  - Die Kombination Vereinigung + Duplikatelimination ist effizienter als Vereinigung mit anschließender Duplikatelimination.

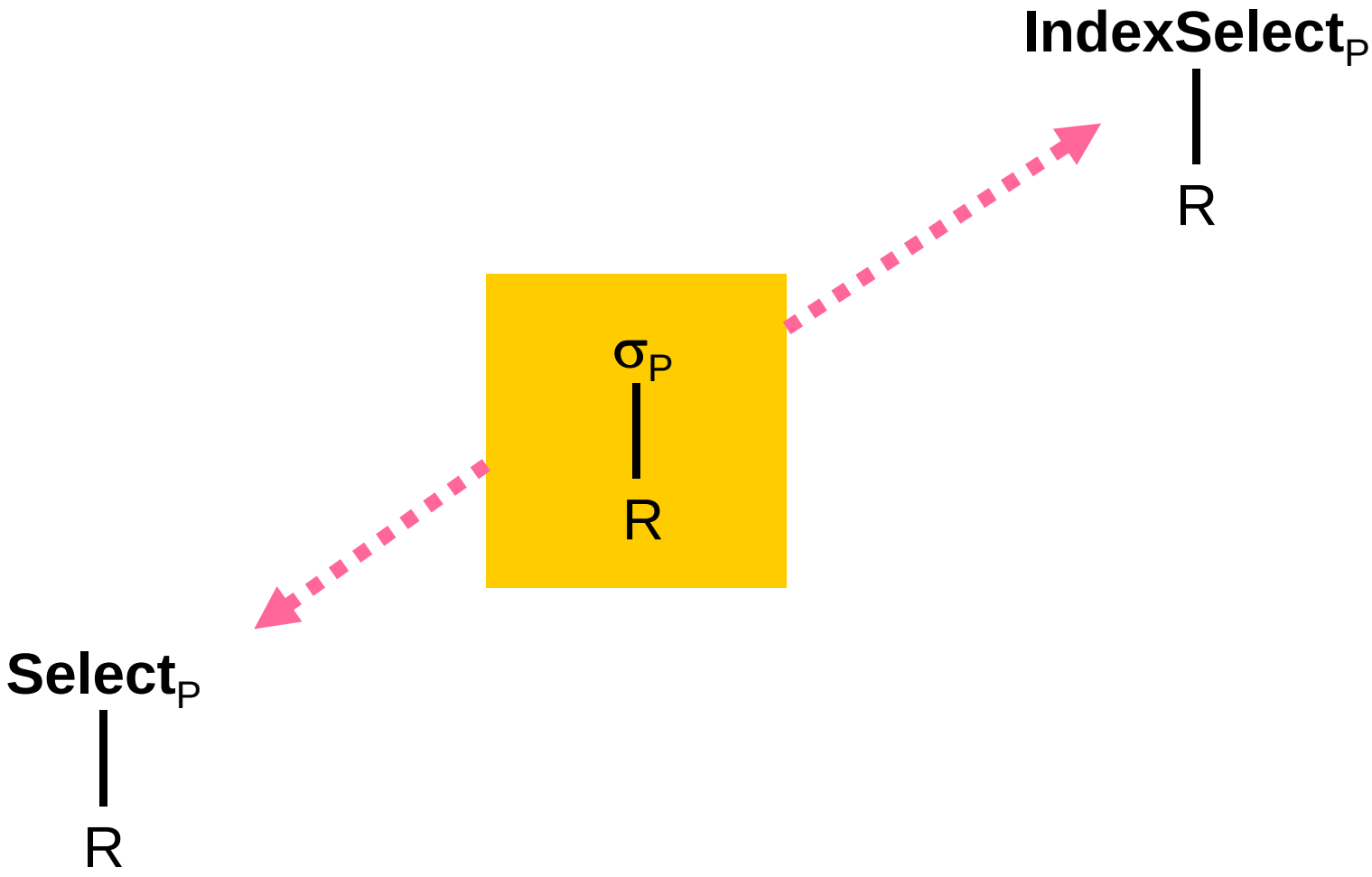
Gruppierung + Aggregat-Funktionen

- Gruppierung mittels Sortierung:
  - Sortiere die Relation für die "group by"- Attribute.
  - Berechne die Aggregatfunktionen mit Hilfe von einem Durchlauf des sortierten Files.
- Gruppierung mittels Hashing:
  - Hash-Funktion für die "group by"- Attribute
  - Die Einträge in der Hash-Tabelle bestehen aus den "group by"- Attributen und Variablen (die laufend aktualisiert werden) für die Aggregat-Funktionen.
  - Am Ende stehen die gesuchten Werte in der Hash-Tabelle.

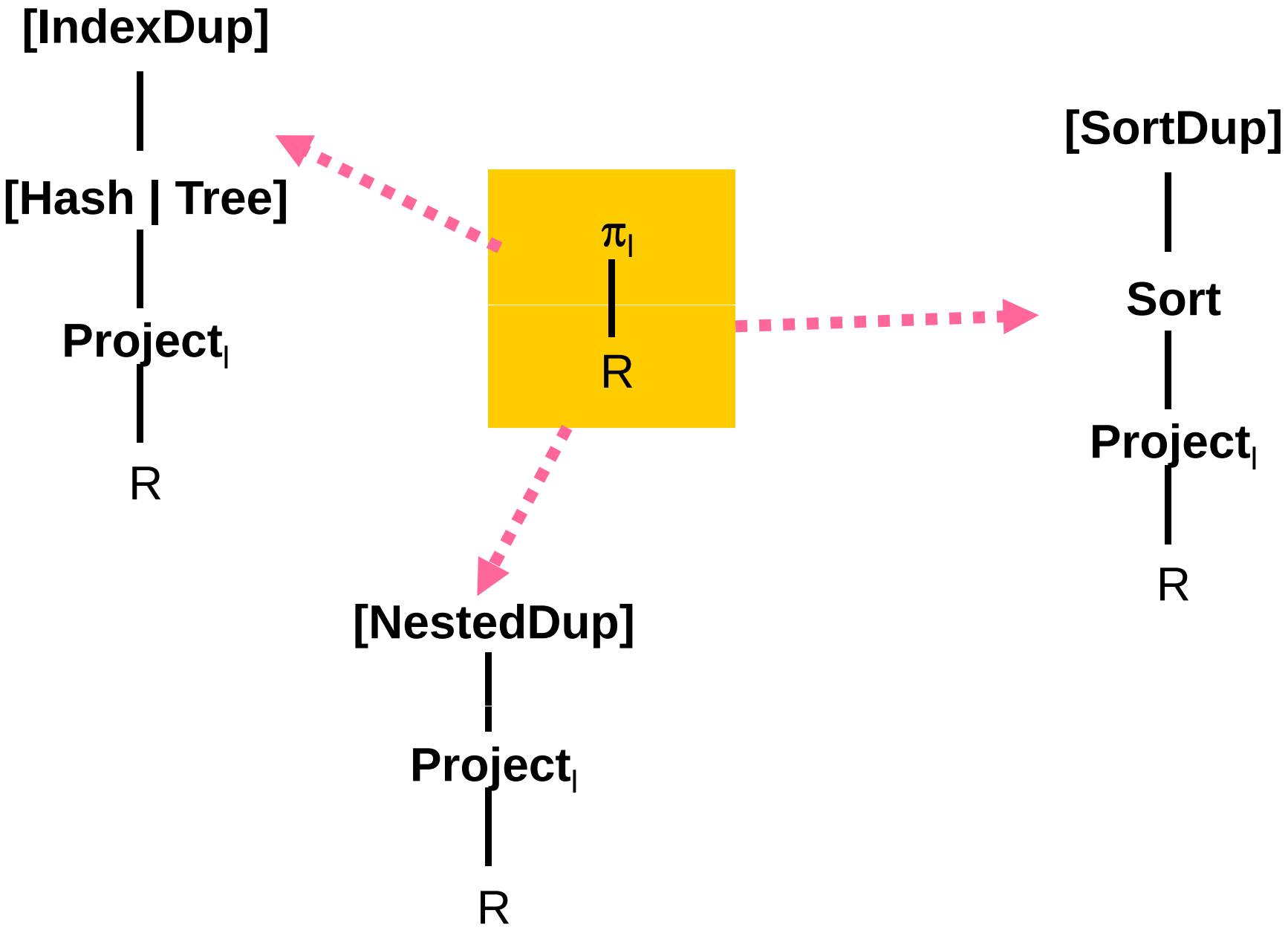
Übersetzung der logischen Algebra



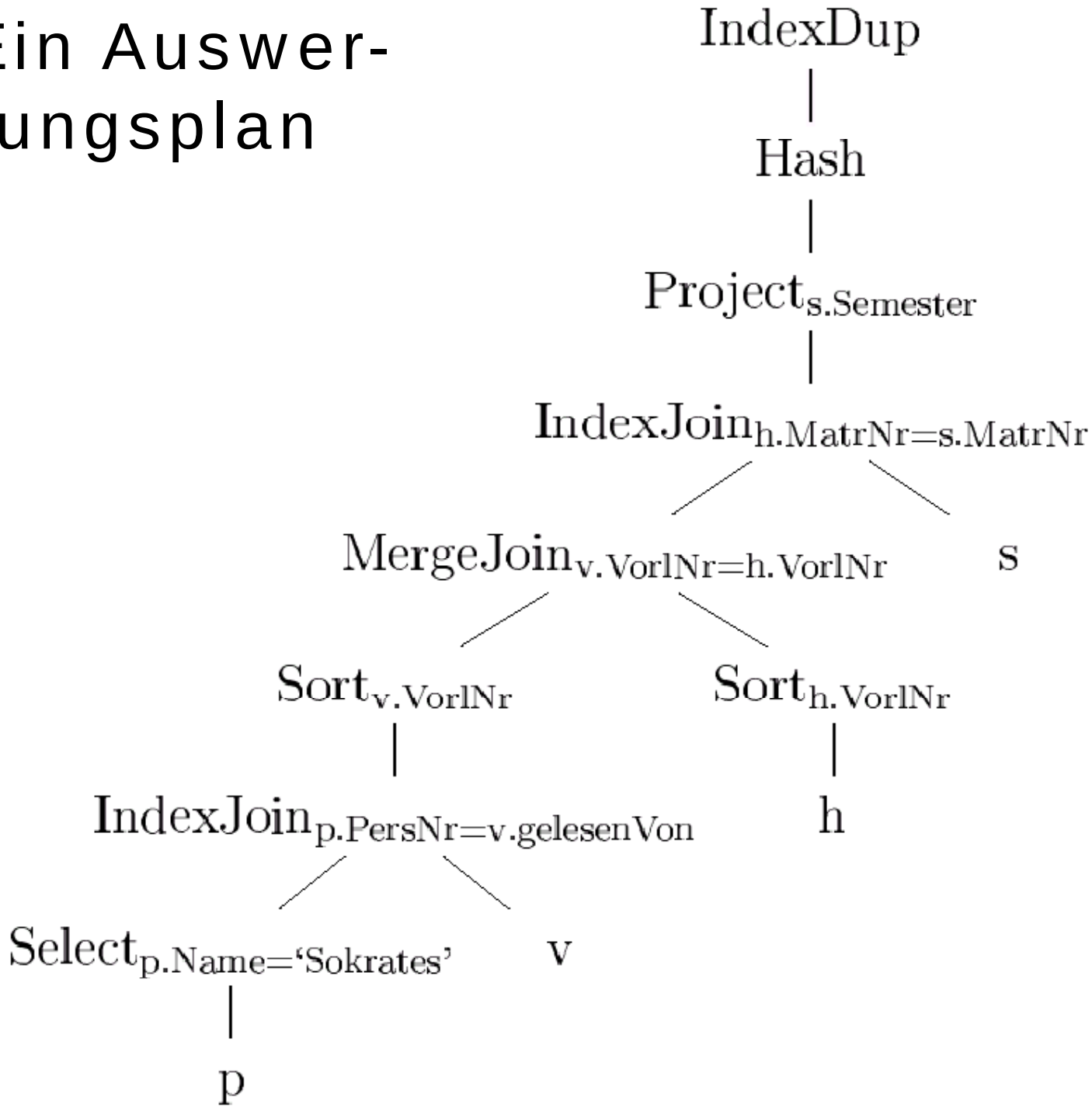
Übersetzung der logischen Algebra



Übersetzung der logischen Algebra

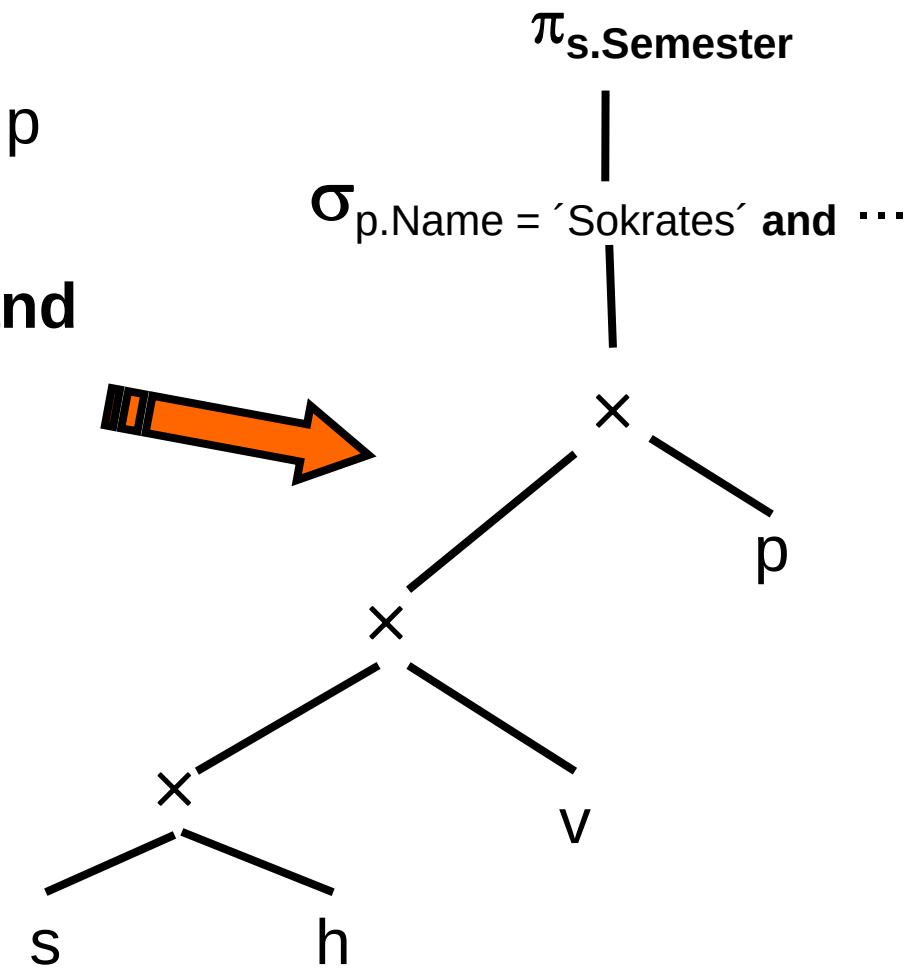


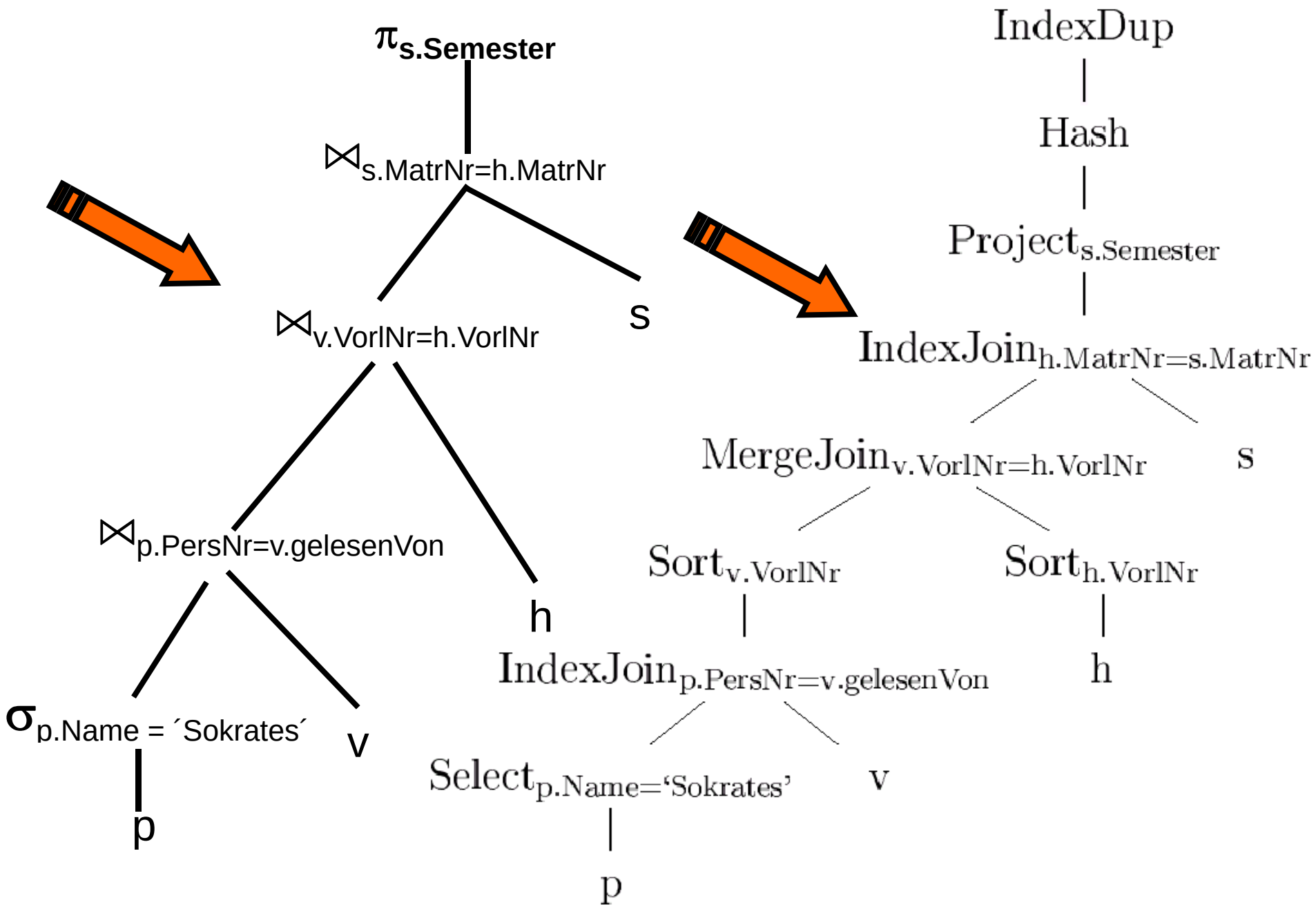
Ein Auswer-tungsplan



Zusammenfassung: Optimierungsphasen

select distinct s.Semester
from Studenten s, hören h
Vorlesungen v, Professoren p
where p.Name = ´Sokrates´ **and**
v.gelesenVon = p.PersNr **and**
v.VorlNr = h.VorlNr **and**
h.MatrNr = s.MatrNr




πs.Semester
⋈s.MatrNr=h.MatrNr
⋈v.VorlNr=h.VorlNr
s
⋈p.PersNr=v.gelesenVon
h
σp.Name = ´Sokrates´
v
p
IndexDup
Hash
Projects.Semester
IndexJoinh.MatrNr=s.MatrNr
MergeJoinv.VorlNr=h.VorlNr
s
Sortv.VorlNr
Sorth.VorlNr
IndexJoinp.PersNr=v.gelesenVon
h
Selectp.Name='Sokrates'
v
p