Physische Optimierung

Physische Optimierung

- Physische Algebraoperatoren stellen die Realisierung der logischen Operatoren dar
- Für einen logischen Operator kann es mehrere physische Operatoren geben
- Nutzen den physischen Aufbau der Datenbank wie Indices, Sortierung von Relationen, … aus

Iterator

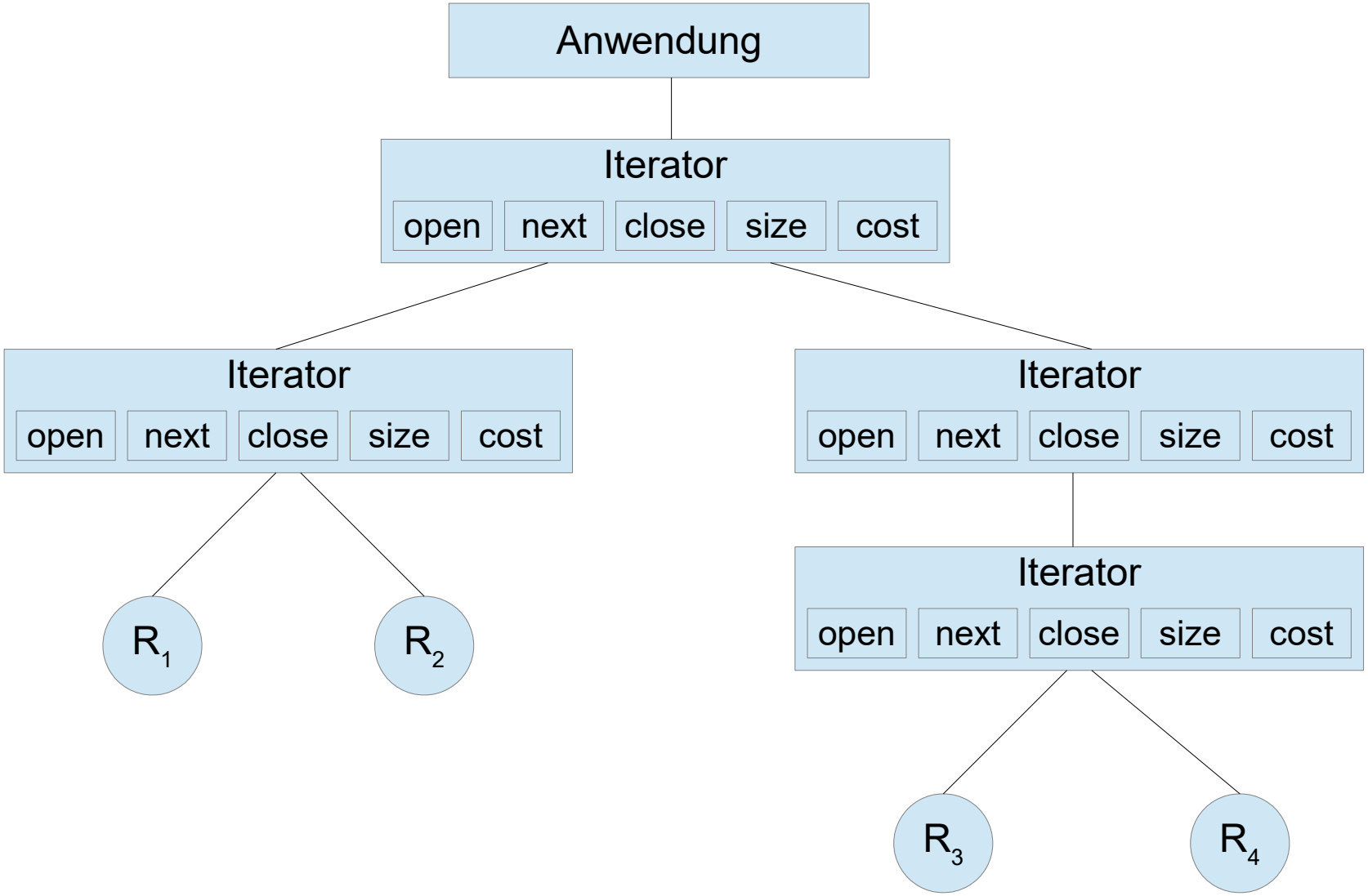
- Ermöglicht das baukastenartige Zusammensetzen der Auswertungspläne
- Ist ein abstrakter Datentyp/Schnittstelle mit folgenden Methoden

~ Cursor
  - **open** Eingabe öffnen, Initialisierungen
  - **next** liefert jeweils das nächste Tupel des Ergebnisses
  - **close** Schließen der Eingabe, Ressourcen freigeben
  - **cost** Infos zu geschätzten Kosten
  - **size** Größe des Ergebnisses

Iterator

- Iteratoren werden so wie auch die relationalen Operatoren baumartig dargestellt und zu einem Ausführungsplan kombiniert.
- Beginnend beim Wurzeliterator werden mittels **next**-Befehl so lange Daten abgerufen, bis keine mehr verfügbar sind. Für die Berechnung benötigt der Iterator die Datensätze der Kind-Iteratoren, wo wiederum mittels Iterator-Befehlen die Datensätze geholt werden.
  Dies setzt sich fort bis zu den Blättern (Basisrelation)
- Zwischenergebnisse müssen nicht zwingend zwischengespeichert werden (zB nicht bei reiner Selektion und Projektion) → ***Pipelining***

Iterator-Baum



Selektion

„Brute-Force“:

iterator Select_p

open
- Öffne Eingabe

next
- Hole solange Tupel, bis eines die Bedingung p erfüllt, ansonsten Ende
- Gib dieses Tupel zurück

close
- Schließe Eingabe

Zugriff über Indexstruktur:

iterator IndexSelect_p

open
- Suche im Index die erste Stelle, an der ein Tupel die Bedingung erfüllt

next
- Gib nächstes Tupel zurück, solange Bedingung erfüllt wird

close
- Schließe Eingabe

Nested Loop Join

Es werden 2 Schleifen ineinander geschachtelt und dabei jedes Tupel der einen Menge mit jedem der anderen verglichen.

Beispiel: $R \bowtie_{R.A=S.B} S$

```
for each r ∈ R
  for each s ∈ S
    if r.A = s.B then
      res := res ∪ (r × s)
```

Nested Loop Join mit Iterator

iterator NestedLoop_p

open
- Öffne die linke Eingabe

next
- Rechte Eingabe geschlossen? → Öffne sie
- Hole rechts so lange Tupel, bis Bedingung p erfüllt ist
- Ist rechte Eingabe erschöpft?
  - Schließe rechte Eingabe
  - Fordere nächstes Tupel der linken Eingabe
  - Starte **next** neu
- Gib den Verbund von aktuellem linken und aktuellem rechten Tupel zurück

close
- Schließe beide Eingaben

Seitenorientierter Nested Loop Join

Berücksichtigung der Seiten-Struktur beim Join.

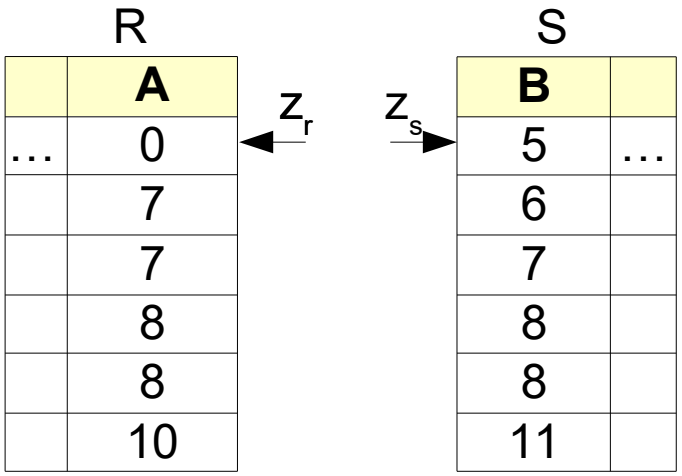
- Für den Join stehen ***m*** Seiten zur Verfügung, davon ***k*** für die inner Relation und somit ***m-k*** für die äußere Relation.
- Teste jede Kombination von Tupeln r $\in$ *R* und s $\in$ *S* die sich gerade im Puffer befinden

```
for each p_R of R
   for each p_s of S
      for each tuple r ∈ p_R and p_s ∈ S
      if r.A = s.B then res:=res ∪ (r ⨯ s)
```

- Weitere Optimierung durch Zick-Zack-Abarbeitung von S, dies spart 1 I/O pro Durchlauf von S.

(Sort-)Merge Join

Wenn beide Eingaben nach den jeweiligen Join-Attributen sortiert sind (werden), kann effizienter gearbeitet werden.



- Paralleles Abarbeiten von oben nach unten
- Zu Beginn jeweils ein Zeiger auf das erste Tupel
- Kleinster Wert ist 0, auf der anderen Seite 5 ->kein Treffer möglich, daher z_r weiterbewegen auf 7
- Nun ist 5 die kleinste Zahl, daher wird z_s weiterbewegt → bei 7 wird Joinpartner gefunden
- Z_r wird weiterbewegt → nochmals Join mit 7

- Sobald ein erster Joinpartner gefunden wurde, muss dieser markiert werden, um bei mehreren gleichen Attributwerten auf beiden Seiten wiederum eine Seite zurücksetzen zu können (siehe Join von 8)

Wie würde der Ablauf im Merge-Join Iterator aussehen?

Merge Join

iterator MergeJoin_p

open

- Öffne beide Eingaben
- Setze *akt* auf linke Eingabe
- Markiere rechte Eingabe

next

- Solange Bedingung p nicht erfüllt
  - setzte *akt* auf Eingabe mit dem kleinsten Wert
  - Rufe **next** auf *akt* auf
  - Markiere andere Eingabe
- Gib Verbund der aktuellen Tupel zurück
- Bewege andere Eingabe vor
- Ist Bedingung nicht mehr erfüllt od. Andere Eingabe erschöpft?
  - Rufe **next** auf akt auf
  - Wert des Joinattributs in *akt* verändert?
    - > Nein, dann setzte andere Eingabe auf Mark zurück
    - > Ansonsten markiere andere Eingabe

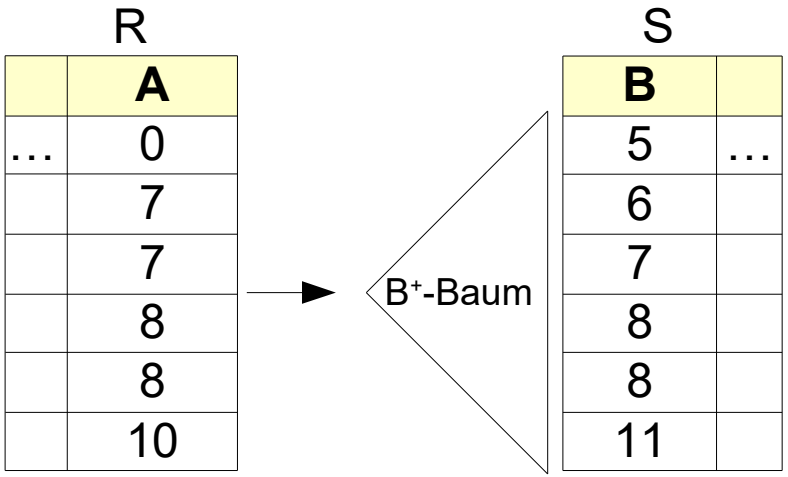
close

- Schließe beide Eingaben

Index Join

Beim Index Join wird ein Index auf einem der Join-Attribute genutzt.

Es müssen für jedes Tupel aus R somit nur die passenden Tupel aus B im Index gesucht werden.



iterator IndexJoin$_p$

open
- Öffne die linke Eingabe
- Hole erstes Tupel aus linker Eingabe
- Suche Joinattributwert im Index

next
- Bilde Join, falls Index ein (weiteres) Tupel zu diesem Attributwert liefert
- Ansonsten bewege linke Eingabe vor und suche Attributwert im Index

close
- Schließe beide Eingaben

Hash Join

Ziel ist, die Eingabedaten so zu partitionieren, dass die Verwendung einer Hauptspeicher-Hashtabelle möglich ist.

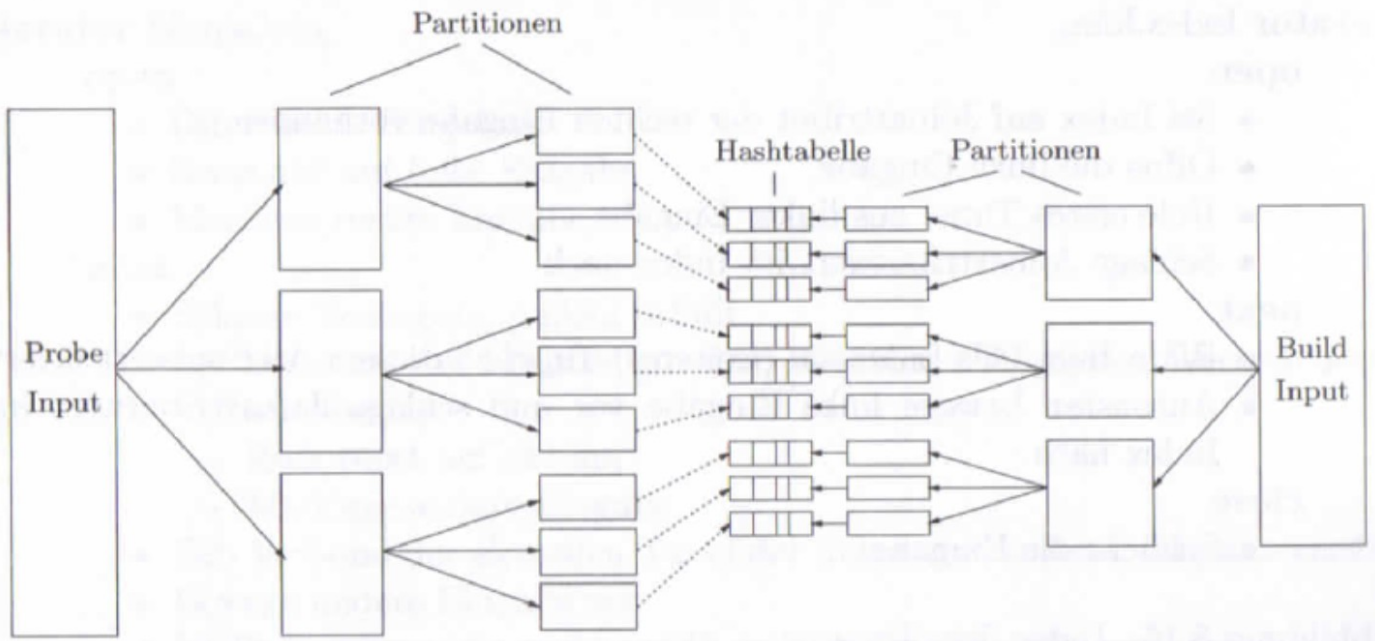
Die **kleinere Relation** wird zum sog. ***Build Input***.
Dabei wird sie so lange partitioniert, bis die Partitionen in den Hauptspeicher passen.
Stehen *m* Seiten zur Verfügung, werden *m-1* Seiten für die Ausgabe verwendet und 1 für die Eingabe.
Die Aufteilung erfolgt rekursiv mittels Hashfunktion.

Als nächstes wird die **größere Relation (Probe Input)** mit den gleichen Hashfunktionen partitioniert. Diese Partitionen müssen aber nicht zwingend in den Hauptspeicher passen.

Hash Join

Im nächsten Schritt (Probe- od. Matching-Phase) werden die Partitionen abgeglichen:

- Lade jeweils 1 Bucket des Build-Inputs in den Hauptspeicher und baue eine Hashtabelle auf
- Lade vom entsprechenden Bucket des Probe Inputs eine Seite nach der anderen in den Puffer und teste jedes Tupel



Quelle: Kemper, Alfons & Eickler, André: Datenbanksysteme (9. Auflage), Oldenbourg Verlag, 2013, Seite 268

Gruppierung u. Duplikateliminierung

Wiederum 3 Methoden:

- **Brute-Force** mit geschachtelten Schleifen
- Bei bestehender **Sortierung** muss nur einmal von Beginn bis Ende gelesen werden und Duplikate entfernt werden
- Nutzen von **Sekundärindex**: z.B. bei B^+-Baum sind in den Blättern entweder Tupel oder Zeiger in sortierter Reihenfolge

Zur Duplikatelimination wird oft auch **Hashing** eingesetzt:

- Build-Phase: Hash-Funktion für Kombination aller Attr.
- Für jedes Bucket wird dann eine in-memory Hash-Tabelle erzeugt und Duplikate sofort verworfen

Projektion und Vereinigung

Bei Projektion und Vereinigung ist in der phyischen Algebra keine autom. Duplikatseliminierung vorgesehen. Wird diese gewünscht, muss sie explizit angegeben werden!

Daher relativ einfache Implementierung:

- Projektion:
  Tupel der Eingabe wird auf die gewünschten Attribute reduziert und an die Ausgabe weitergeleitet.
  Wird oft mit einem anderen Schritt kombiniert (Join, ...)
- Vereinigung:
  Es werden nacheinander alle Tupel der linken und rechten Eingabe ausgegeben.

Sortieren

Anwendungsfälle:

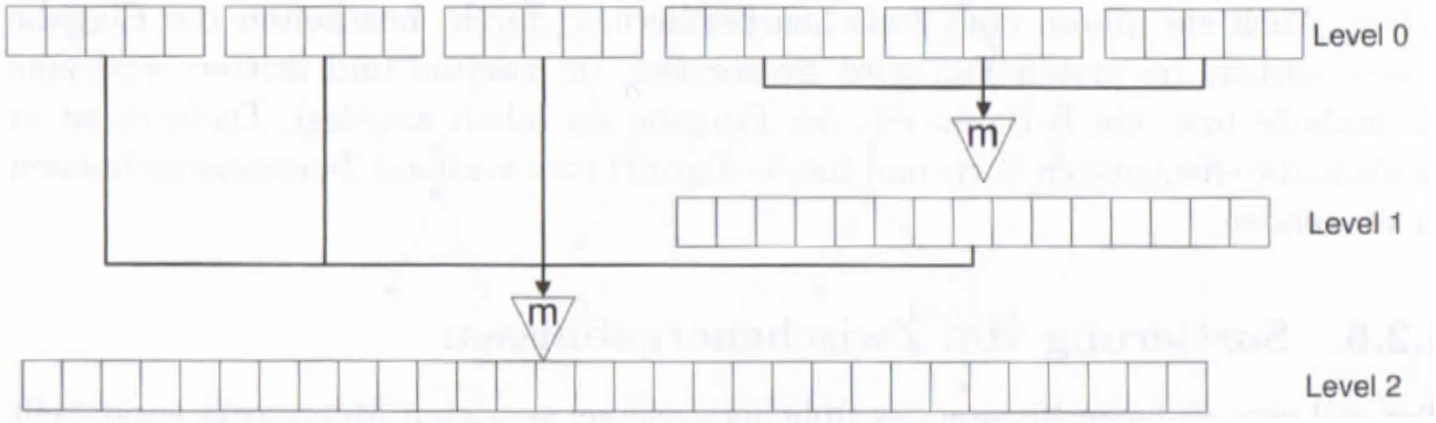
- Sortieren ist keine relationale Operation, wird aber für manche Implementierungen benötigt (zB Sort-Merge Join)
- Bei „Order by“-Klausel
- Möglichkeit zur Duplikat-Elimination

Problem ist die Sortierung von großen Datenmengen, die großteils auf dem Hintergrundspeicher abgelegt sind → kein in-memory Sortieren (QuickSort, …) möglich.

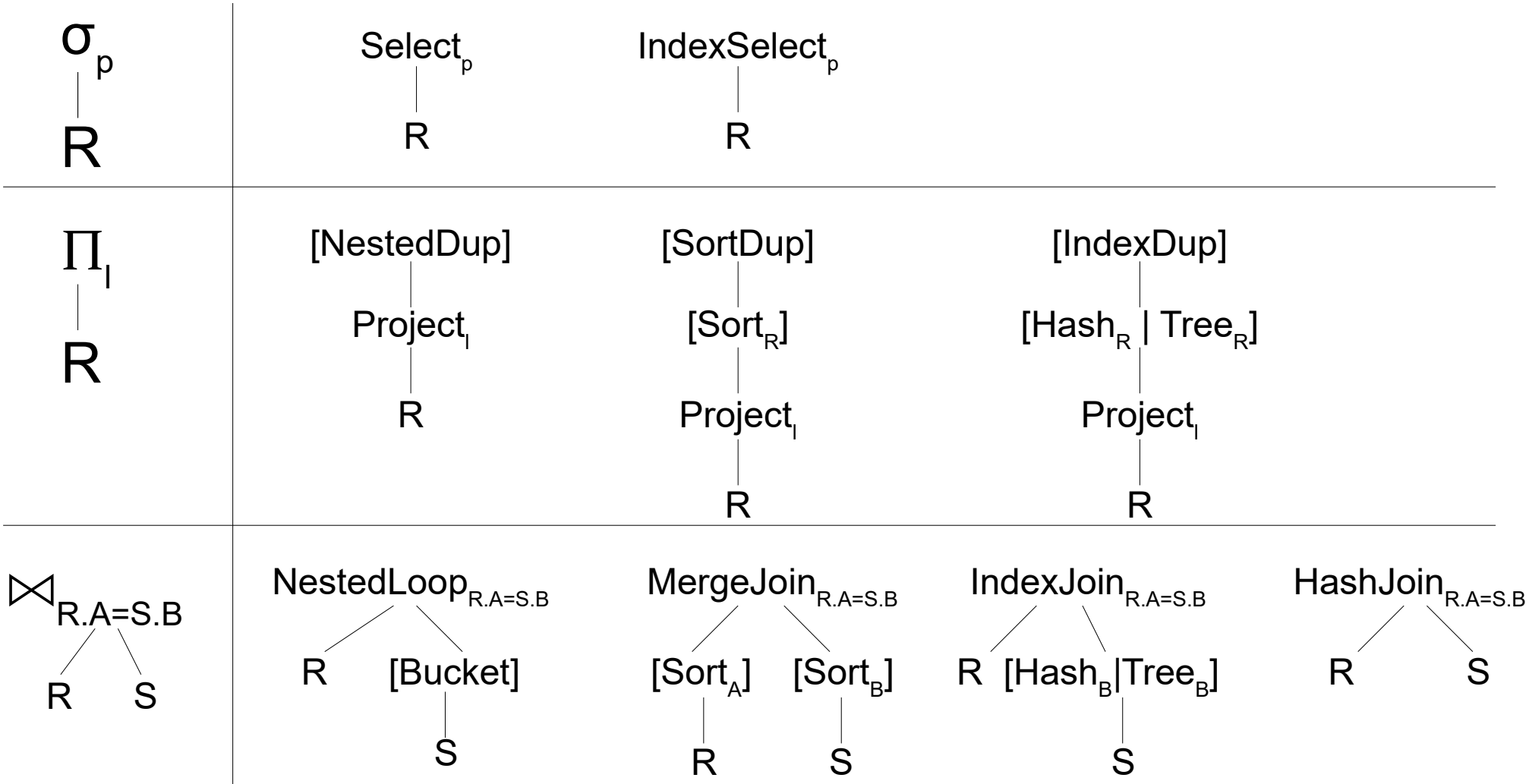
→ Merge Sort: Stückweise Sortieren und dann zusammenfügen

Merge Sort

- Initialisierung, Level-0 Läufe
  Jede Seite wird eingelesen, im Hauptspeicher sortiert und das Ergebnis in eine temp. Relation geschrieben
- Mischen
  Stehen m Seiten im Hauptspeicher zur Verfügung, werden m-1 Läufe gemischt. Die freie Seite steht für die Ausgabe zur Verfügung.
- In den Zwischenphasen sollen so wenig Läufe wie nötig gemischt werden.



Übersetzung der logischen Algebra



Nehmen Sie die Anfrage von Folie 22 und leiten Sie einen physischen Auswertungsplan ab, wobei folgende physischen Gegebenheiten gelten: Primärindex (Hashtabelle) auf allen PrimaryKeys, *gelesenVon* ein B⁺-Baum.