

6. Implementierung relationaler Operationen

■ Selektion

■ Verbundalgorithmen

- Nested-Loop-Join / Nested-Block-Join
- Sort-Merge-Join
- Hash-Join und Varianten
- Mehrwege-Joins

■ Sortierung

■ Weitere Operationen



Implementierung der Selektion

■ Relationen-Scan (Scan-Operator)

- immer möglich
- Definition von einfachen Suchargumenten
- Start- und Stop-Bedingungen für Bereichsanfragen
- Attributprojektion (ohne Duplikateliminierung)

■ Index-Scan

- Auswahl des kostengünstigsten Index
- Spezifikation des Suchbereichs (Start-, Stop-Bedingungen)

■ TID-Algorithmus

- Auswertung aller "brauchbaren" Indexstrukturen
- Auffinden von variabel langen TID-Listen
- Boole'sche Verknüpfung der einzelnen Listen
- Zugriff zu den Tupeln entsprechend der Zielliste
- bei Plattenspeichern TID-Sortierung zur Minimierung von Plattenzugriffsarmbewegungen sinnvoll

■ Boole'sche Verknüpfung von Bit-Indizes



Join-Algorithmen

■ Verbund (Join)

- satztypübergreifende Operation: gewöhnlich sehr teuer
- häufige Nutzung: wichtiger Optimierungskandidat
- typische Anwendung: Gleichverbund
- allgemeiner Θ -Verbund selten

■ Implementierung der Verbundoperation kann gleichzeitig Selektionen auf den beteiligten Relationen R und S ausführen

```
SELECT *
FROM R, S
WHERE R.VA  $\Theta$  S.VA AND PR AND PS
```

```
SELECT P.*
FROM PERS P, ABT A
WHERE P.ANR=A.ANR AND
P.Alter<30 AND A.Ort=„Leipzig“
```

- VA: Verbundattribute
- P_R und P_S: Prädikate definiert auf Selektionsattributen (SA) von R und S

■ mögliche Zugriffspfade

- Scans über R und S (immer möglich)
- Scans über I(R(VA)), I(S(VA)) (wenn vorhanden) → liefern Sortierreihenfolge nach VA
- Scans über I(R(SA)), I(S(SA)) (wenn vorhanden) → ggf. schnelle Selektion für P_R und P_S
- Scans über andere Indexstrukturen (wenn vorhanden) → ggf. schnelleres Auffinden aller Sätze



Nested-Loop-Join

■ Annahme:

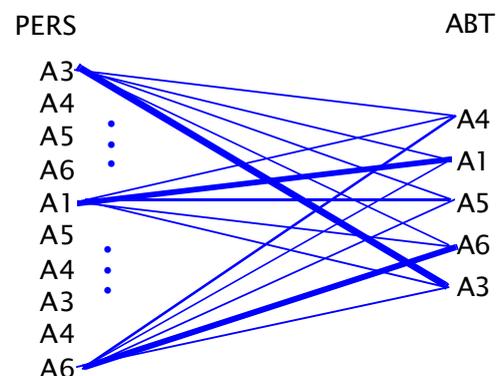
- Sätze in R und S sind nicht nach den Verbundattributen geordnet oder
- es sind keine Indexstrukturen I(R(VA)) und I(S(VA)) vorhanden

■ Berechnung allgemeiner Θ -Joins

Algorithmus

```
Scan über R,
für jeden Satz r, falls PR:
  Scan über S,
  für jeden Satz s,
    falls PS AND (r.VA  $\Theta$  s.VA)
    führe Verbund aus, d.h. übernehme
    kombinierten Satz (r, s) in Resultatmenge
```

Beispiel: Gleichverbund zwischen PERS und ABT über ANR



- quadratische Komplexität: $O(N^2)$ (Kardinalität N für R und S)



Nested-Loop-Join: Indexnutzung

■ Nested-Loop-Join mit Indexzugriff auf innere Relation S (Gleichverbund)

Scan über R,
für jeden Satz r, falls P_R :
ermittle mittels Indexzugriff für S alle TIDs für Sätze s mit $s.VA = r.VA$
für jedes TID,
hole Satz s,
falls P_S : übernehme kombinierten Satz (r, s) in die Resultatmenge

Nested-Block-Join

Scan über R,
für jede Seite (bzw. Menge benachbarter Seiten) von R:
Scan über S,
für jede Seite (bzw. Menge benachbarter Seiten) von S:
für jeden Satz r der R-Seite, falls P_R :
für jeden Satz s der S-Seite,
falls P_S AND $(r.VA \Theta s.VA)$:
übernehme kombinierten Satz (r, s) in die Resultatmenge

■ Beispiel: R 1 Million Sätze; S 20.000 Sätze; Blockungsfaktor $B = 50$

- 20 Milliarden Satzvergleiche wenn keine Reduktion über Selektionsprädikate
- Anzahl Blöcke: $n = |R| / B = 20\,000$, $m = |S| / B = 400$

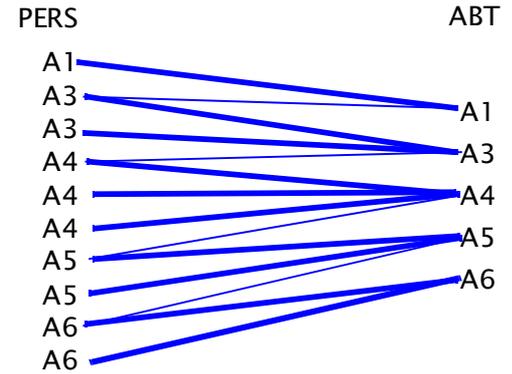
#Blockzugriffe abhängig von nutzbarer Hauptspeichergröße

- nutzbare Hauptspeichergröße $M=2$ Blöcke
 - $n*m$ Blockzugriffe $\rightarrow 20000 * 400 = 8$ Millionen Blockzugriffe
- $M=101$ Seiten
 - 100 Seiten für S, 1 Seite für R
 - >
- $M=m+1$ (401 Seiten): $m+n$ Seitenzugriffe \rightarrow Zugriffe (lineare Kosten für E/A)

(Sort-) Merge-Join

- nur für Gleichverbund
- Algorithmus besteht aus 2 Phasen:
 - Phase 1: Sortierung von R und S nach $R(VA)$ und $S(VA)$, falls nicht bereits vorhanden. dabei frühzeitige Eliminierung nicht benötigter Sätze ($\rightarrow P_R, P_S$)
 - Phase 2: schritthaltende Scans über sortierte R- und S-Sätze mit Durchführung des Verbundes bei $r.VA = s.VA$

Beispiel: Gleichverbund zwischen PERS und ABT über ANR

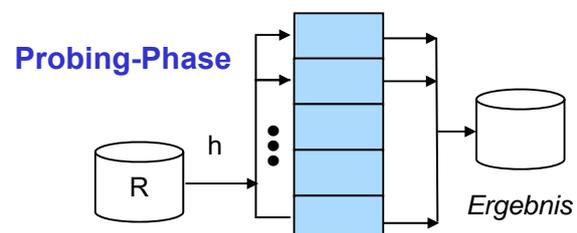
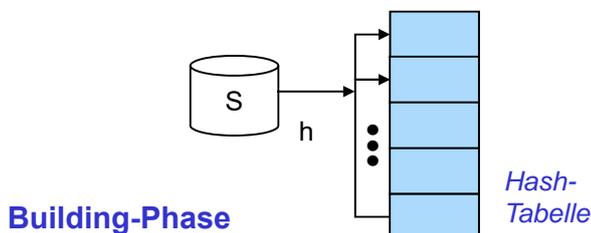


- **Komplexität:** $O(N)$ bei vorliegender Sortierung, ansonsten $O(N \log N)$
- Spezialfall: Ausnutzung von Indexstrukturen auf Verbundattributen
 - (Annahme: $I(R(VA))$ und $I(S(VA))$ vorhanden)
 - schritthaltende Scans über $I(R(VA))$ und $I(S(VA))$
 - falls $R(VA) = S(VA)$, Überprüfung von P_R und P_S in den zugehörigen Sätzen
 - falls P_R und P_S , Bildung des Verbundes



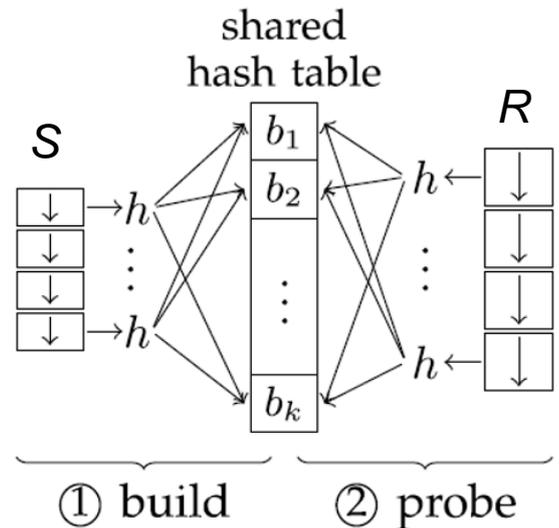
Hash-Join

- nur für Gleichverbund
- Idealfall: kleinere (innere) Relation S passt vollständig in Hauptspeicher
 - **Building-Phase:** Einlesen von S und falls P_S Speicherung in einer Hash-Tabelle unter Anwendung einer Hash-Funktion h auf dem Join-Attribut
 - **Probing-Phase:** Einlesen von R und falls P_R Überprüfung für jeden Join-Attributwert, ob zugehörige S-Tupel vorliegen (wenn ja, erfolgt Übernahme ins Join-Ergebnis)
- Vorteile
 - lineare Kosten $O(N)$
 - Partitionierung des Suchraumes: Suche nach Verbundpartnern nur innerhalb 1 Hash-Klasse
 - Nutzung großer Hauptspeicher
 - auch für Joins auf Zwischenergebnissen gut nutzbar



Hash-Join: Multi-Core Optimierung

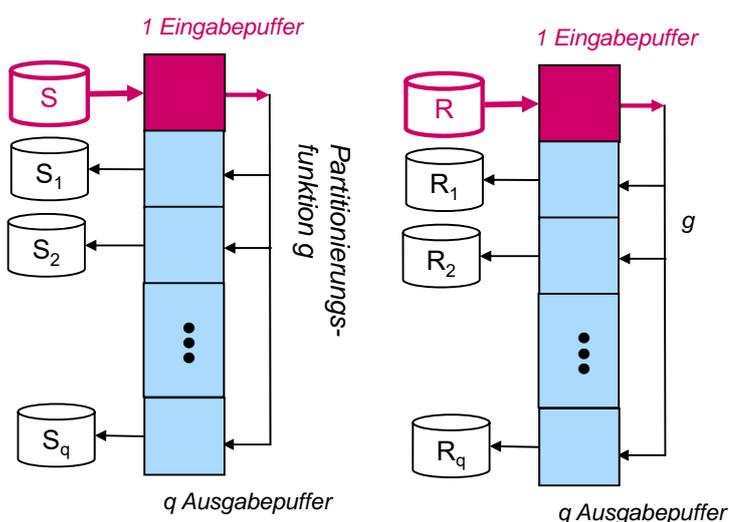
- parallele Build- und Probing-Phasen durch p Threads
- Zerlegung beider Eingabetabellen in p gleich große Partitionen
- Schreibsynchronisation auf gemeinsame Hash-Tabelle in Build-Phase
 - Semaphore (Latch) pro Hash-Klasse
 - geringe Konfliktgefahr vbei vielen (Millionen) von Hash-Klassen
- paralleles Probing (nur Lesezugriffe auf Hash-Tabelle)
- Partitionierung der Eingabetabellen ermöglicht auch bessere Cache-Tefferaten



Hash-Join (2)

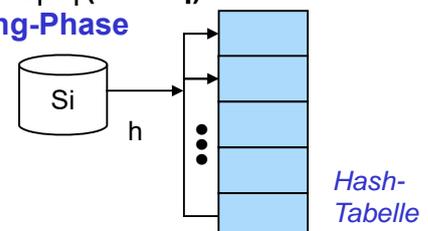
- allgemeiner Fall: kleinere Relation passt nicht vollständig in Hauptspeicher => **Überlaufbehandlung** durch Partitionierung der Eingaberelationen
 - Partitionierung von S und R in q Partitionen über (Hash-)Funktion g auf dem Join-Attribut, so dass jede S -Partition in den Hauptspeicher passt
 - q -fache Anwendung des Basisalgorithmus' auf je zwei zusammengehörigen Partitionen
- rund 3-facher E/A-Aufwand gegenüber Basisverfahren ohne Überlauf
 - Lesen und partitioniertes Zurückschreiben von S und R , Lesen von S und R zur Join-Berechnung

Partitionierungs-Phase für S und R

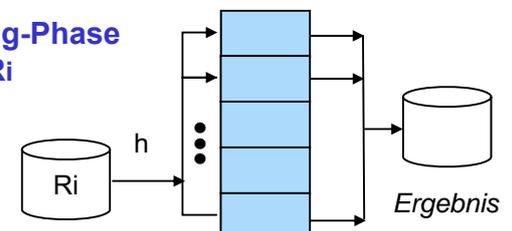


für jedes S_i/R_i ($i = 1..q$):

Building-Phase für S_i



Probing-Phase für R_i



Hash-Join: Varianten (1)

■ TID-Hash-Join

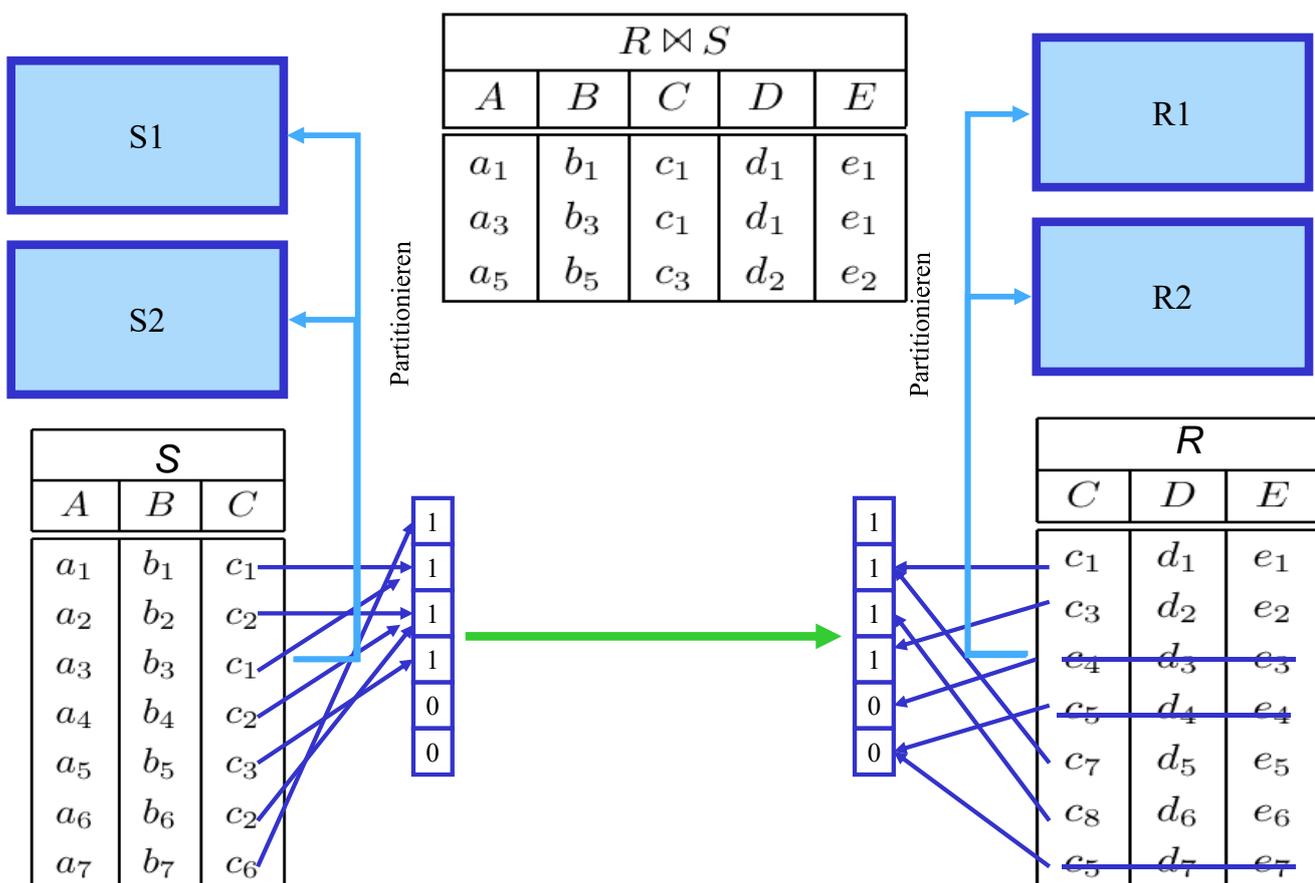
- zur Platzersparnis werden in Hash-Tabelle nur Kombinationen: (Verbundattributwert, TID) gespeichert
- Idealfall ohne Überlaufbehandlung wird eher erreicht (bzw. weniger Partitionen)
- separate Materialisierungsphase für Ergebnistupel erforderlich

■ Nutzung von Bitvektoren (Hash-Filter)

- während Partitionierung von S wird Bitvektor erstellt, in dem über Hash-Funktion vorhandenen Join-Attributwerten zugeordnete Bits gesetzt werden
- nur solche R-Tupel werden weiter berücksichtigt (während Partitionierung von R), für deren Join-Attributwert zugehöriges Bit gesetzt ist
- kleinere R-Partitionen beschleunigen Join-Verarbeitung



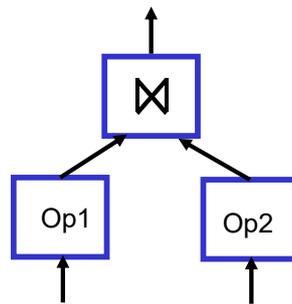
Hash-Filter: Beispiel



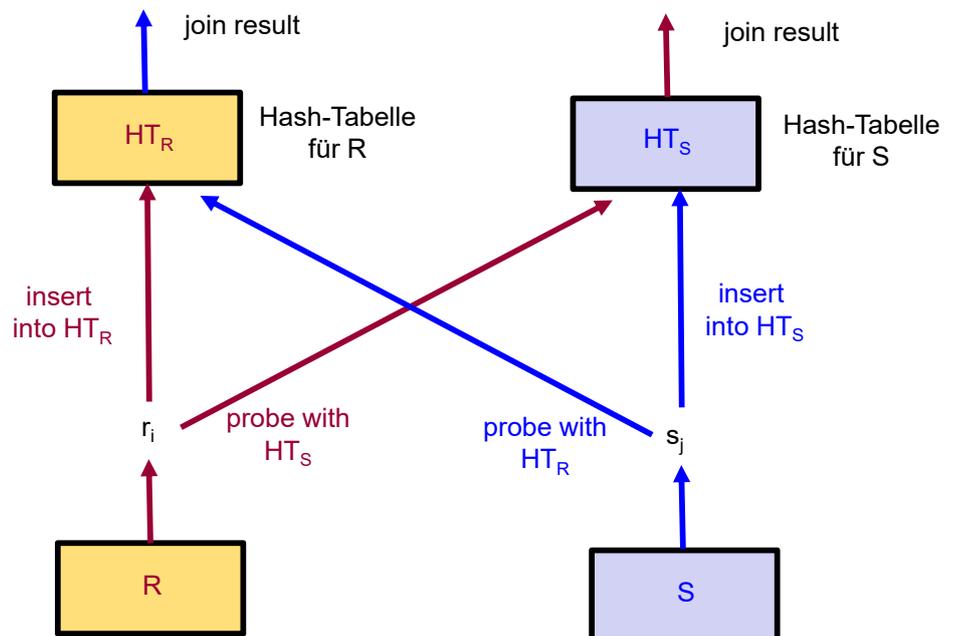
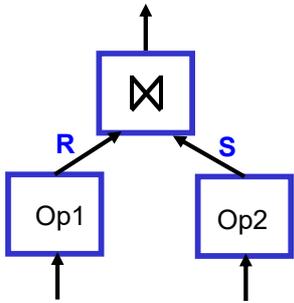
Hash-Join: Varianten (2)

■ symmetrischer Hash-Join (double-pipelined Hash Join)

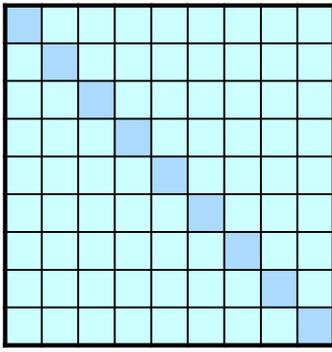
- Hash-Tabellen für beide Tabellen
- Nutzung für Joins auf Zwischenergebnissen im Operatorbaum
- Pipelining: inkrementeller Aufbau der Hash-Tabellen und Test auf Verbundpartner
- keine Blockierung der Operator-Pipeline bis Eingabedaten für Building-Phase vollständig vorliegen



Symmetrischer Hash-Join

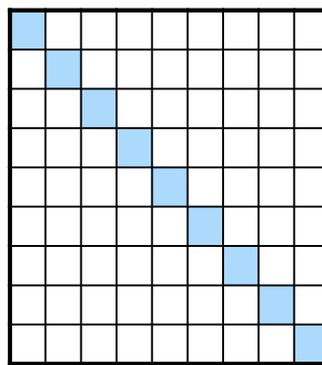


Verbundalgorithmen - Vergleich



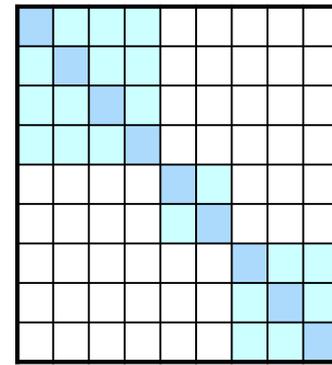
Nested-Loop

Elementvergleich



Merge

Elementvergleich, der zu Join-Ergebnis führt



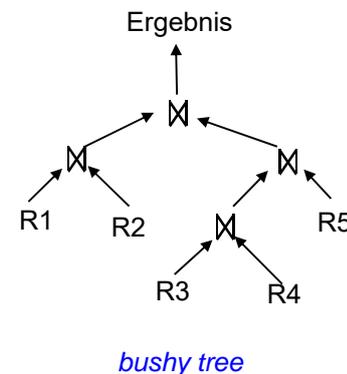
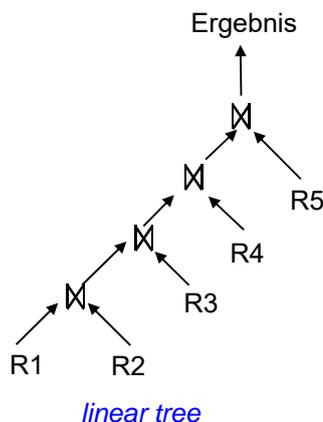
Hash-Join

- **Nested-Loop-Join:** immer anwendbar, vollständiges Durchsuchen des gesamten Suchraums
- **Sort-Merge-Join:** geringste Suchkosten für Gleichverbund, falls Indexstrukturen auf beiden Verbundattributen vorhanden. Sortieren beider Relationen nach Verbundattributen reduziert ansonsten den Kostenvorteil erheblich
- **Hash-Join:** Partitionierung des Suchraums für Gleichverbund



Mehrwege-Join

- N-Wege-Verbund (N Relationen) kann durch N-1 2-Wege-Joins realisiert werden



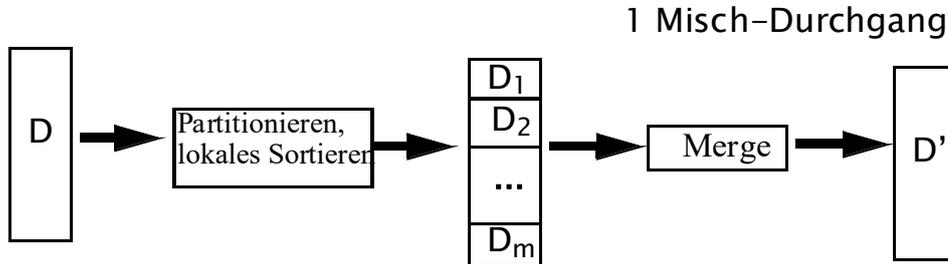
- **komplexe Optimierung**

- N! mögliche lineare Verbundreihenfolgen
- lineare (left-deep, right-deep) vs. unbeschränkte (bushy) Join-Bäume
- Festlegung von N-1 Verbundmethoden
- Nutzung von Pipelining, um Speicherung temporärer Zwischenergebnisse zu reduzieren

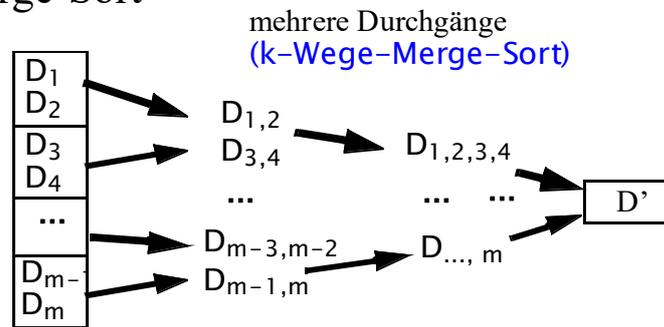


Externes Sortieren: Merge-Sort

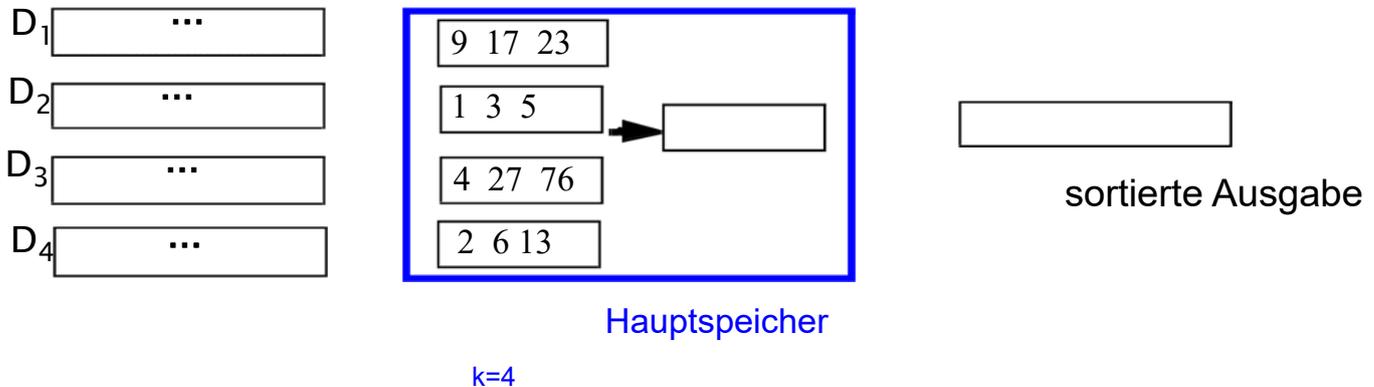
- große Datenmengen können nicht im Hauptspeicher sortiert werden
 - Einlesen und Zerlegung der Eingabe in m Läufe (runs)
 - Sortieren und Zwischenspeichern (Zurückschreiben) der sortierten Läufe
 - Einlesen und sukzessives Mischen der Läufe bis 1 sortierter Lauf entsteht
- optimaler Fall: nur 1 Misch-Durchgang



- allgemeiner Fall: k -Wege-Merge-Sort



Mischen: Beispiel



Externes Sortieren (3)

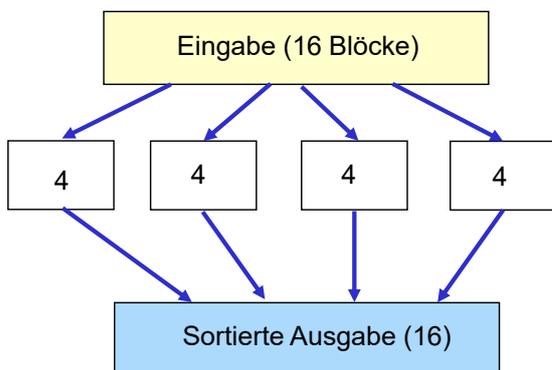
- Anzahl der initialen Läufe m hängt vom verfügbaren Hauptspeicher (HS) ab
 - bei $M+1$ HS-Seiten können M Läufe gemischt werden (1 Seite zur Generierung der Ausgabe)
 - der Umfang eines initialen Laufes kann höchstens M Seiten umfassen, um eine interne Sortierung zu ermöglichen $m \geq N/M$ ($N = \text{\#Seiten der Eingaberelation}$)
- wünschenswert (1 Misch-Durchgang, $\text{\#Durchgänge } d=1$): Zerlegung der unsortierten Eingabe in höchstens M Läufe ($m \leq M$):
 - $M \geq m \geq N/M \rightarrow M^2 \geq N$
- Idealfall: $M \geq \sqrt{N}$, $m = \sqrt{N} \Rightarrow$ 1 Misch-Durchgang ausreichend ($d=1$)
 - linearer E/A-Aufwand für Partitionierung und Mischen
- allgemein gilt für $k=M$ Seiten: $d = \log_M N - 1$



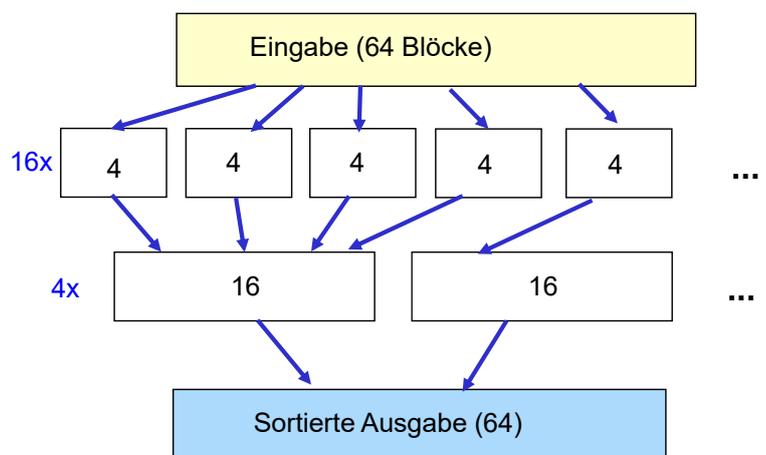
Externes Sortieren (4)

- Beispiele für $M=4$

a) $N=16 : m=4$



b) $N=64 : m=16$



- Sortieren von 1 TB Daten:

- $N = 125$ Millionen Seiten, $\sqrt{N} = 11\,180$ Seiten



Weitere Operationen

- skalare Aggregatberechnungen: MIN, MAX, COUNT, SUM, AVG
 - Nutzung von Indexstrukturen bzw. sequentielle Abarbeitung
- Kreuzprodukt: Realisierung gemäss Nested-Loop Join
- Durchschnitt / Vereinigung / Differenz: als Spezialfälle von Join realisierbar
 - Beispiel: innere Relation S in Hauptspeicher-Hash-Tabelle
 - modifiziertes Probing mit R zur Bestimmung von Duplikaten und des Ergebnisses
- Duplikat-Eliminierung
 - 1 Eingaberelation
 - Realisierung über Sortierung oder Hash-basierte Strategien
- Gruppierung
 - Sortierung bzw. Hashing bezüglich Gruppierungsattribut
 - pro Attributwert Bestimmung der in Anfrage geforderten Aggregatwerte (SUM, COUNT, MAX etc.)



Zusammenfassung

- Join-Algorithmen
 - Nested-Loop-Join: Basisverfahren, allgemein einsetzbar (Theta-Join; keine Voraussetzungen bezüglich Indexierung)
 - E/A-Reduzierung über Nested-Block-Join (Nutzung größerer Hauptspeicher)
 - Sort-Merge-Join: effizienter Gleichverbund bei Vorliegen einer Sortierung / Indexierung auf Verbundattribut
 - Hash-Join: schneller Gleichverbund mit linearer Komplexität; effiziente Nutzung großer Hauptspeicher
 - zahlreiche Varianten, u.a. zur TID-Optimierung, Indexnutzung, Hash-Filter, kombinierte Auswertung von Selektionen ...
- externe Sortierung
 - $O(N \log N)$, aber lineare Komplexität falls nur 1 Misch-Durchgang
 - möglich falls bei N zu sortierenden Blöcken Hauptspeicher-Seitenzahl $M > \sqrt{N}$
- Implementierung Mengenoperationen, Duplikateliminierung etc.
 - abbildbar auf Join-artige Verfahren
 - Einsatz von Sortierung oder Hashing

