Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Hash-Join Algorithmen

Advanced Topics in Databases Ws08/09

Matthias Richly

Einleitung

- Grundlage ist das Paper: "*Join Processing in Database Systems With Large Main Memories*"

- Quelle: ACM Transactions on Database Systems (TODS)
  - Volume 11 , No. 3
  - Pages: 239 – 264
  - September 1986

- Autor: Leonard D. Shapiro
  - Professor of Computer Science

Gliederung

- Hash-Join Algorithmen allgemein
- Einfacher Hash-Join
- GRACE Hash-Join
- Hybrid Hash-Join
- Vergleich der Algorithmen
- Partition Overflow

Hash-Join Algorithmen allgemein

- klassisch: kleinere Relation R passt in Hauptspeicher
  - Erzeuge Hash-Tabelle für Tupel aus R in Hauptspeicher, gehasht auf die Join-Attribute
  - lies andere Relation S sequenziell, berechne für jedes Tupel den Hashwert und prüfe unter diesem Wert in der Hash-Tabelle, ob es passende Tupel gibt

- hier: kleinere Relation R passt nicht in Hauptspeicher
  - partitioniere zunächst R und S in disjunkte Teilmengen, joine dann sich entsprechende Teilmengen
  - Partitionierung: teile Wertebereich der Hashfunktion h in Teilmengen $H_1,\ldots,H_n$; ein Tupel t landet in Partition P_i, wenn $h(t) \in H_i$
  - Algorithmen erfordern, dass die Partitionierung Partitionen bestimmter Größe liefert

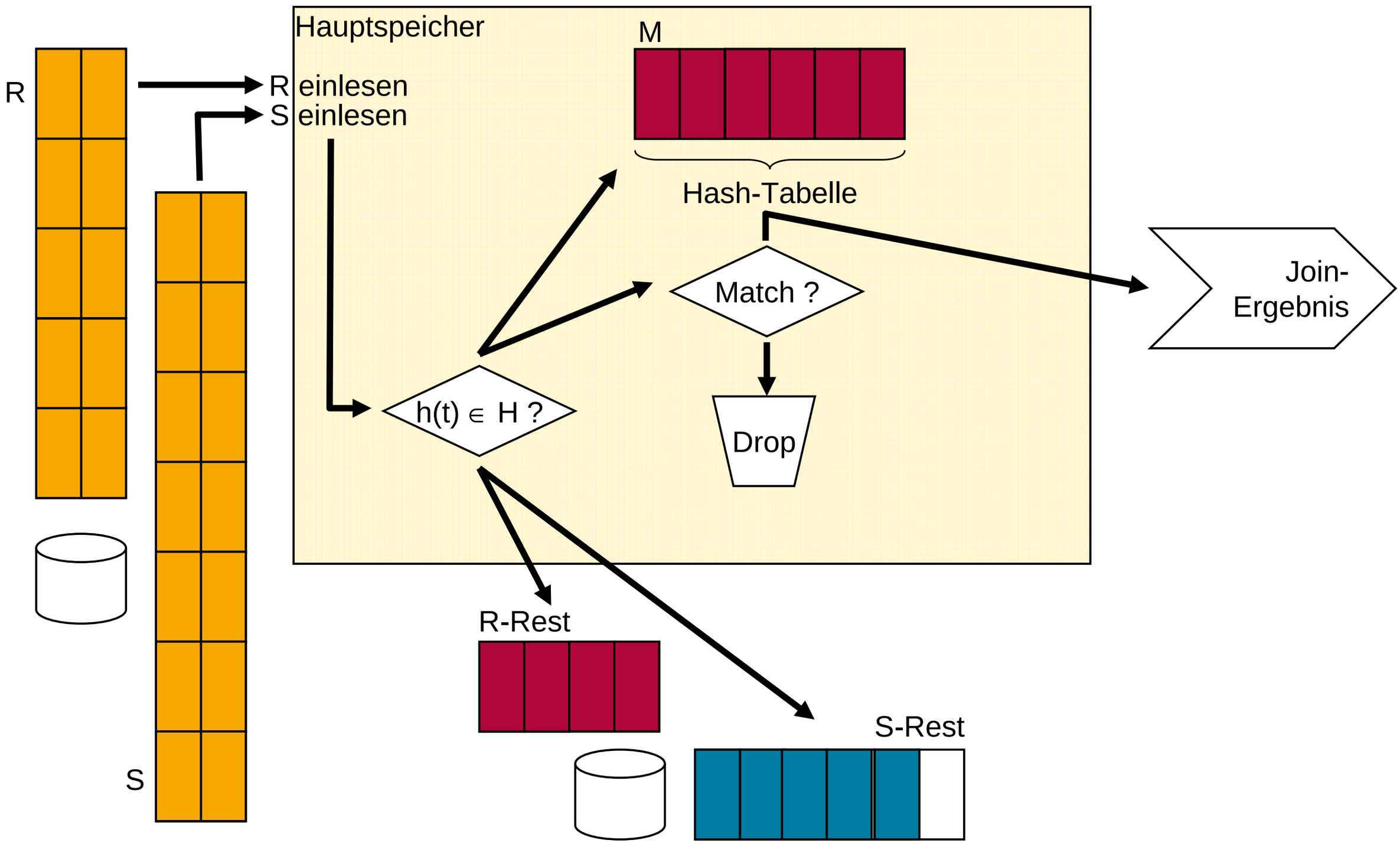
Notation/Annahmen

- zu berechnen ist der Equijoin der Relationen R und S, beide sind weder geordnet noch indiziert
- |R| - Anzahl der Blöcke, die R belegt (entsprechend für S)
- |R| ≤ |S|
- M - für Join zu Verfügung stehender Hauptspeicher
- |M| - Anzahl der zu Verfügung stehenden Blöcke im HS
- $|M| \geq \sqrt{|R|}$ („*Large Main Memory*“)
- die Partitionierung funktioniert perfekt, keine Partition wird größer als erwartet
- bei I/O-Kosten: erstes Einlesen und Ausgabe des Join-Ergebnisses zählen nicht mit

Einfacher Hash-Join

- Algorithmus:
  - wähle eine Hashfunktion h und eine Teilmenge H ihres Wertebereichs, so dass alle R-Tupel, die in H gehasht werden, gerade in M passen
  - lies R, für jedes Tupel r: wenn h(r) $\in$ H, füge es in Hashtabelle in M ein; sonst schreibe es in eine extra Datei auf Disk
  - lies S, für jedes Tupel s: wenn h(s) $\in$ H, suche in Hashtabelle nach Matches und gib diese aus; sonst schreibe s in eine extra Datei auf Disk
  - wiederhole die Schritte für die übrig gebliebenen Tupel aus den beiden extra Dateien, solange bis keine R-Tupel mehr übrig bleiben

Einfacher Hash-Join - Animation



Einfacher Hash-Join – I/O-Kosten

- Algorithmus benötigt $A = \left\lceil \frac{|R|}{|M|} \right\rceil$ Durchläufe
- Schreiben und wieder Einlesen der R-Tupel in/aus extra Datei:

$$2 \cdot \left(\sum_{i=1}^{A} |R| - i \cdot |M| \right) = 2 \cdot \left(\sum_{i=1}^{A} |R| - \sum_{i=1}^{A} i \cdot |M| \right) = 2 \cdot \left((A-1) \cdot |R| - \frac{A \cdot (A-1)}{2} \cdot |M| \right)$$

- Schreiben und wieder Einlesen der S-Tupel in/aus extra Datei:

$$2 \cdot \left(\sum_{i=1}^{A} |S| - i \cdot |M| \cdot \frac{|S|}{|R|} \right) = 2 \cdot \left(\sum_{i=1}^{A} |S| - \sum_{i=1}^{A} i \cdot |M| \cdot \frac{|S|}{|R|} \right) = 2 \cdot \left((A-1) \cdot |S| - \frac{A \cdot (A-1)}{2} \cdot |M| \cdot \frac{|S|}{|R|} \right)$$

  - Annahme: Gleichverteilung der Join-Attribute in R und S
- gesamt:

$$= 2 \cdot \left((A-1) \cdot (|R| + |S|) - \left(\frac{A \cdot (A-1)}{2} \cdot |M| \right) \cdot \left(1 + \frac{|S|}{|R|} \right) \right)$$

Einfacher Hash-Join – Fazit

- gut, wenn fast ganz R in Hauptspeicher passt
  - → große Teile von R und S werden dann nur einmal gelesen

- schlecht, wenn nur wenig von R in Hauptspeicher passt
  - → viele Durchläufe nötig

GRACE Hash-Join

- Algorithmus:

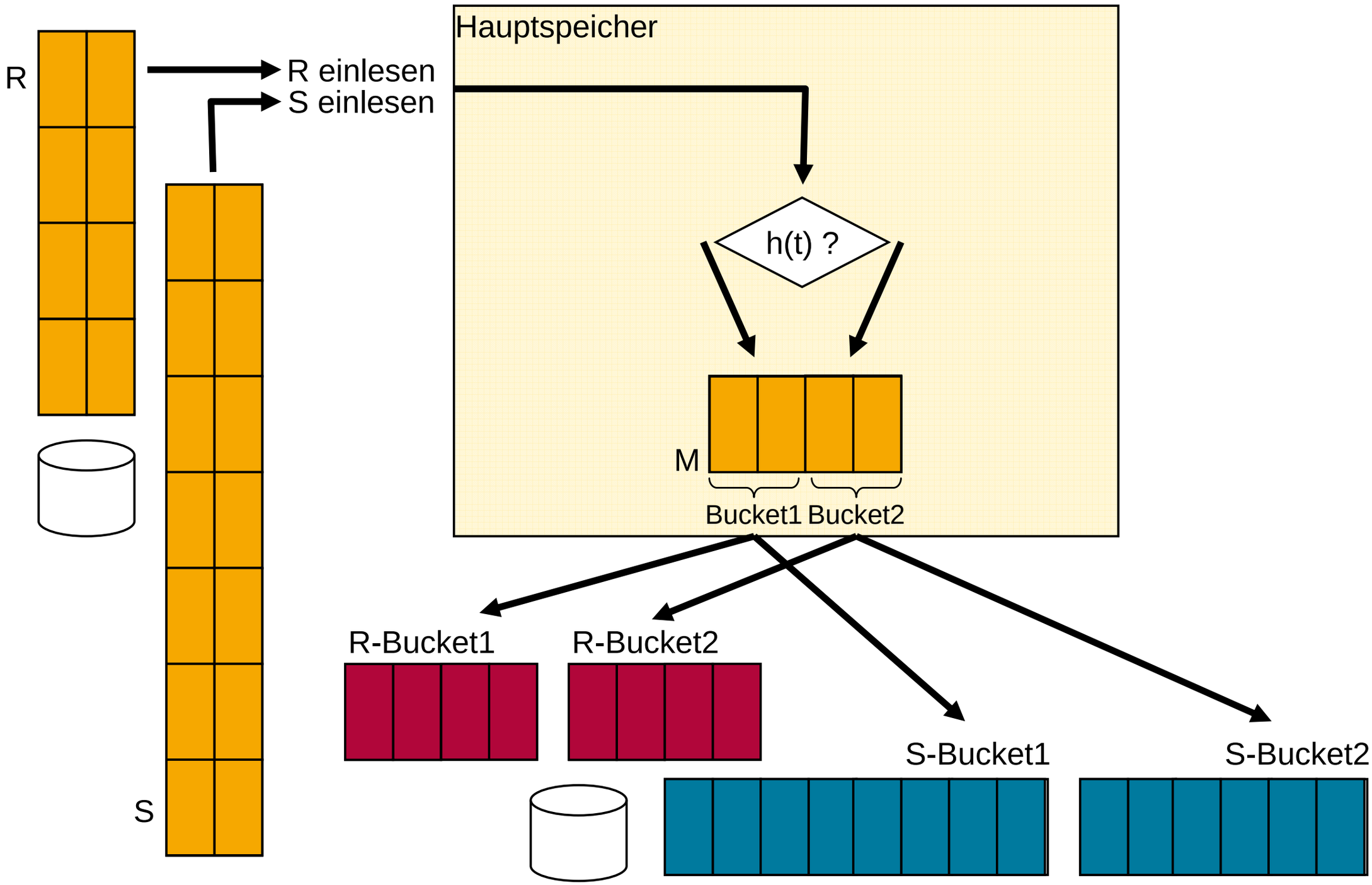
Phase 1:

- wähle Hashfunktion h und $\sqrt{|R|}$ Teilmengen ihres Wertebereichs, so dass alle R-Partitionen etwa gleich groß werden, lege für jede Partition einen Ausgabepuffer in M an
- lies R, für jedes Tupel r: berechne h(r) und füge r in den entsprechenden Ausgabepuffer ein; wenn ein Puffer voll ist, schreibe ihn auf Disk; am Ende schreibe alle Puffer auf Disk
- lies S, verfahre wie bei R

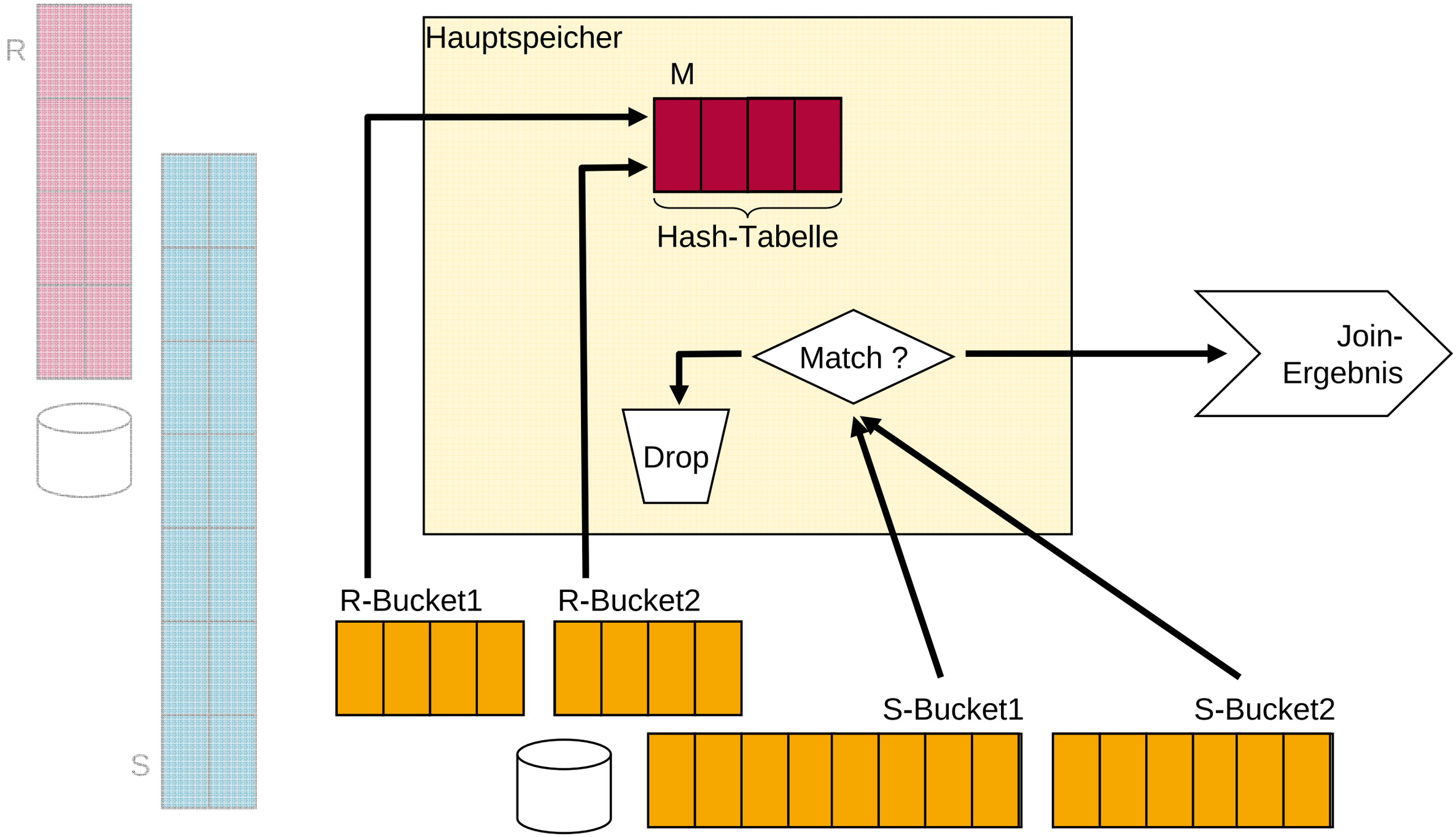
Phase 2:

- für jede R-Partition R_i:
  - lies R_i in M und erzeuge dabei Hashtabelle
  - lies S_i, für alle Tupel s: berechne h(s) und prüfe in Hashtabelle auf Matches, gib diese ggf. aus

GRACE Hash-Join – Animation (Phase 1)



GRACE Hash-Join – Animation (Phase 2)



GRACE Hash-Join – I/O-Kosten

- Schreiben der partitionierten Relationen auf Disk
  - |R| + |S|
- Einlesen der partitionierten Relationen von Disk
  - |R| + |S|
- I/O Einsparungen, wenn mehr als $\sqrt{|R|}$ Blöcke HS zur Verfügung stehen → Blöcke können zwischengespeichert werden
  - $\min(|R| + |S|, |M| - \sqrt{|R|}) \cdot 2$
- gesamt:
  - $2 \cdot ((|R| + |S|) - \min(|R| + |S|, |M| - \sqrt{|R|}))$

GRACE Hash-Join – Fazit

- gut, wenn wenig (nicht viel mehr als $\sqrt{|R|}$) Blöcke Hauptspeicher zur Verfügung stehen
  - vermeidet im Ggs. zum einfachen Hash-Join, dass Blöcke wiederholt eingelesen werden müssen

- schlecht, wenn große Teile von R in den Hauptspeicher passen
  - immer 2 Durchläufe

Hybrid Hash-Join

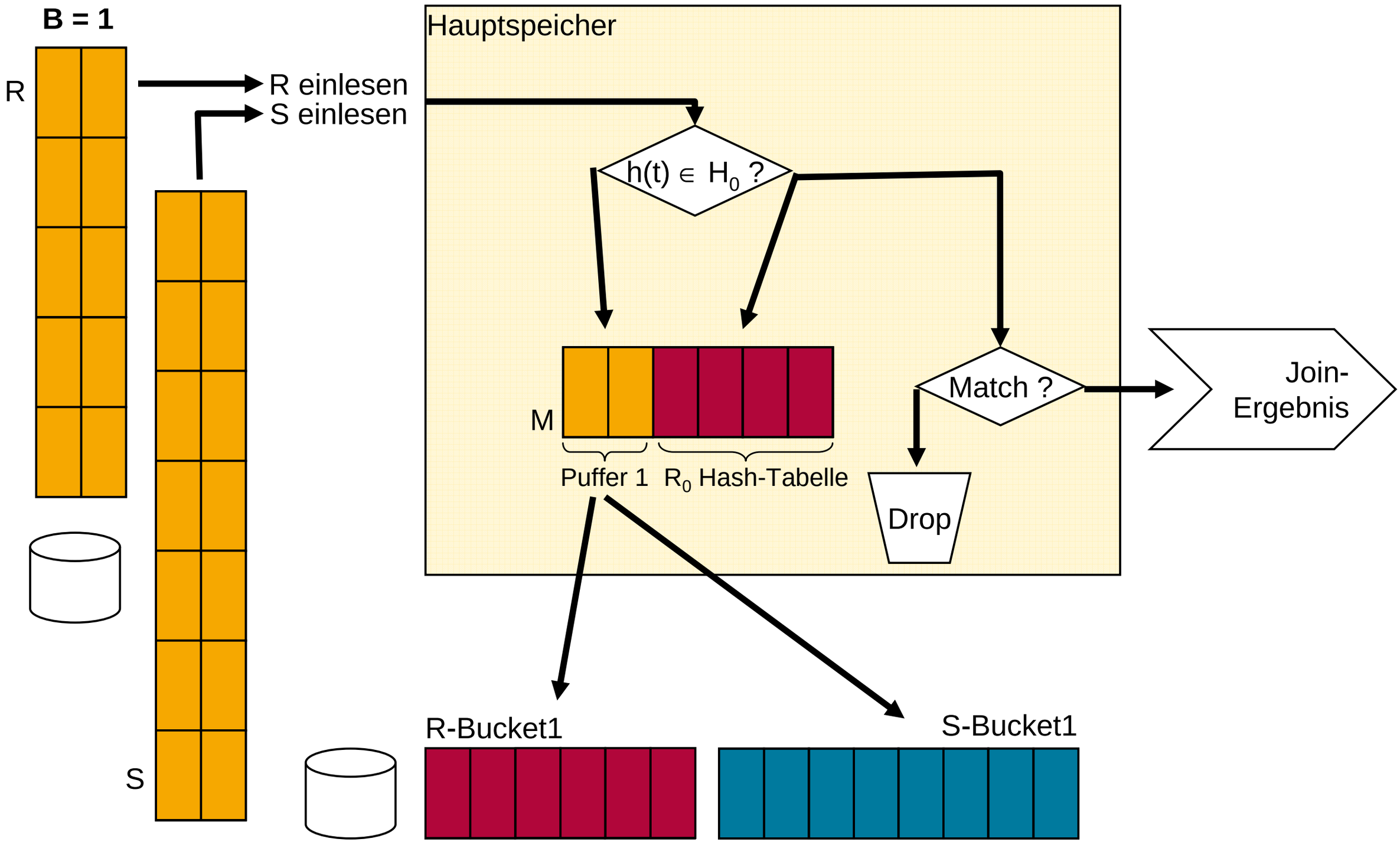
- Idee:
  - kombiniere die Vorteile von den beiden vorangegangenen Algorithmen
  - nutze so wenig wie möglich Blöcke von M, um in Phase 1 Partitionen zu erzeugen, die in Phase 2 gerade in M passen; verfahre mit diesen wie beim GRACE Hash-Join
  - nutze den Rest der Blöcke von M, um für eine Partition eine Hashtabelle in M aufzubauen, die schon in Phase 1 zum Joinen verwendet wird (einfacher Hash-Join)

  - Konstante B – mindestens benötigte Anzahl an Partitionen, die in Phase 1 auf Disk erzeugt werden, so dass jede entstehende R-Partition nicht mehr als |M| Blöcke verbraucht

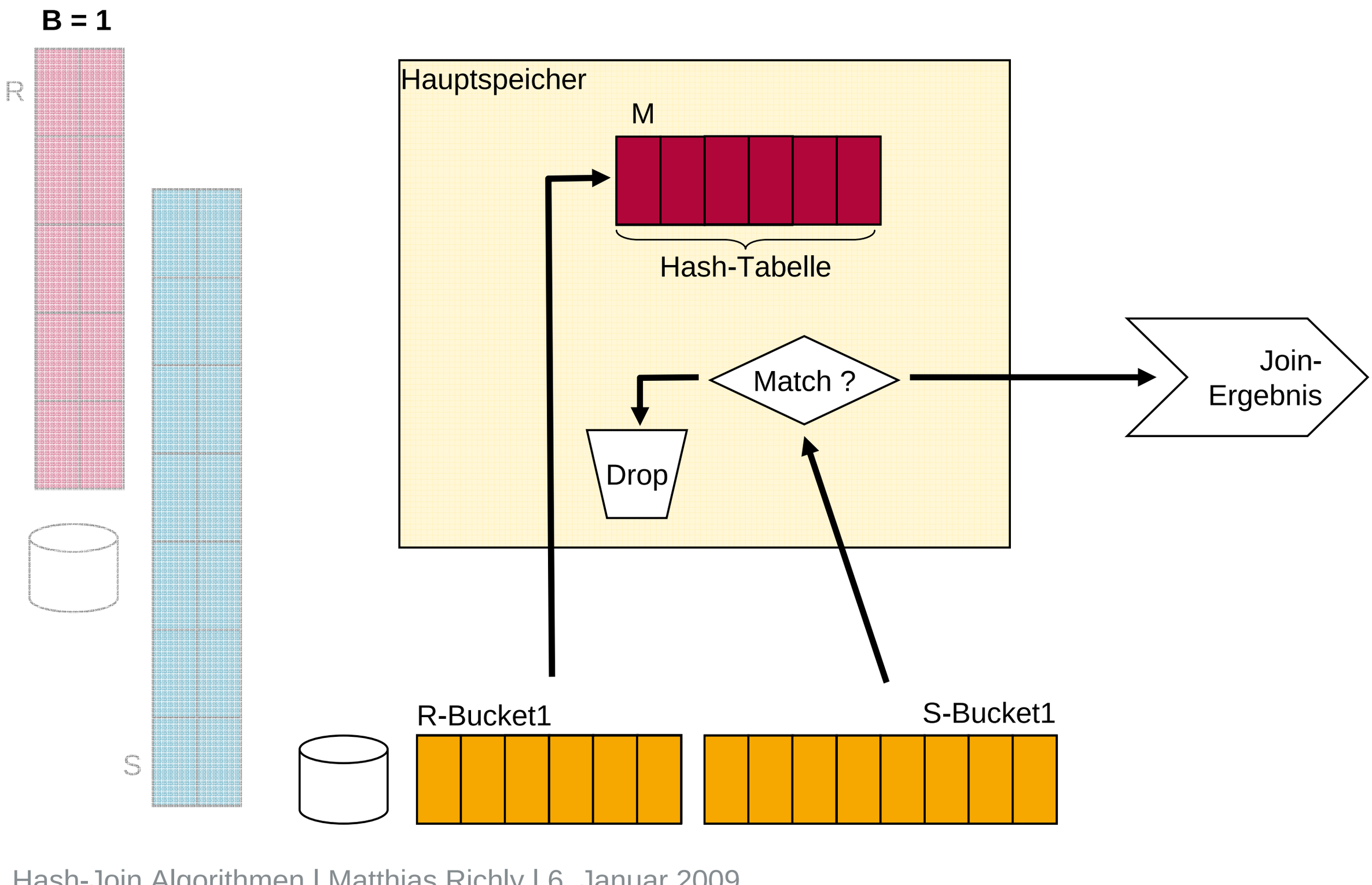
Hybrid Hash-Join

- Algorithmus:
  - Phase 1:
    - wähle Hashfunktion h und Teilmengen $H_0,\ldots,H_B$ ihres Wertebereichs, so dass für R eine Partition R_0 mit |M|-B Blöcken entsteht, und Partitionen $R_1,\ldots,R_B$ gleicher Größe (≤ |M| Blöcke) entstehen
    - verwende B Blöcke von M als Ausgabepuffer und den Rest für eine Hashtabelle der Partition R_0
    - lies R, für jedes Tupel r: berechne h(r); wenn $h(r) \in H_0$, füge r in Hashtabelle ein, sonst in den entsprechenden Puffer; ist Puffer voll: schreibe auf Disk, am Ende alle Puffer auf Disk schreiben
    - lies S, für jedes Tupel s: berechne h(s); wenn $h(s) \in H_0$, prüfe auf Matches in der Hashtabelle von R_0 und gib Tupel ggf. aus, sonst schreibe s in den entsprechenden Puffer, …
  - Phase 2 (=GRACE):
    - wiederhole für i = 1,…,B
      - lies R_i und erzeuge Hashtabelle davon in M
      - lies S_i, für jedes Tupel s: berechne h(s) und prüfe in Hashtabelle von R_i auf Matches, gib ggf. Tupel aus

Hybrid Hash-Join – Animation (Phase 1)



Hybrid Hash-Join – Animation (Phase 2)



Hybrid Hash-Join

- Herleitung der Formel für Konstante B:
  - B – mindestens benötigte Anzahl an Partitionen, die in Phase 1 auf Disk erzeugt werden, so dass jede entstehende R-Partition nicht mehr als |M| Blöcke verbraucht

$$B = \frac{\text{Anzahl der Blöcke von R, die in Phase 2 bearbeitet werden müssen}}{\text{Blöcke im Hauptspeicher}}$$

$$B = \frac{|R| - (|M| - B)}{|M|}$$

$$B \cdot |M| = |R| - |M| + B$$

$$B \cdot |M| - B = |R| - |M|$$

$$B \cdot (|M| - 1) = |R| - |M|$$

$$B = \frac{|R| - |M|}{|M| - 1} \text{ ist das theoretische Minimum, also: } B = \left\lceil \frac{|R| - |M|}{|M| - 1} \right\rceil$$

Hybrid Hash-Join – I/O-Kosten

- q: Anteil von R_0 an R
  - $q = |R_0| / |R| = (|M|-B) / |R|$
- Größe von $S_0 = q \cdot |S|$
  - Annahme: Gleichverteilung der Join-Attribute in R und S
  - → d.h. der Anteil von R und S, der auf Disk geschrieben wird ($R_1 \ldots R_B$ und $S_1 \ldots S_B$), ist (1-q)

- Schreiben und wieder Einlesen der Partitionen von R und S auf Disk
  - $2 \cdot (|R| + |S|) \cdot (1-q)$

Hybrid Hash-Join – Fazit

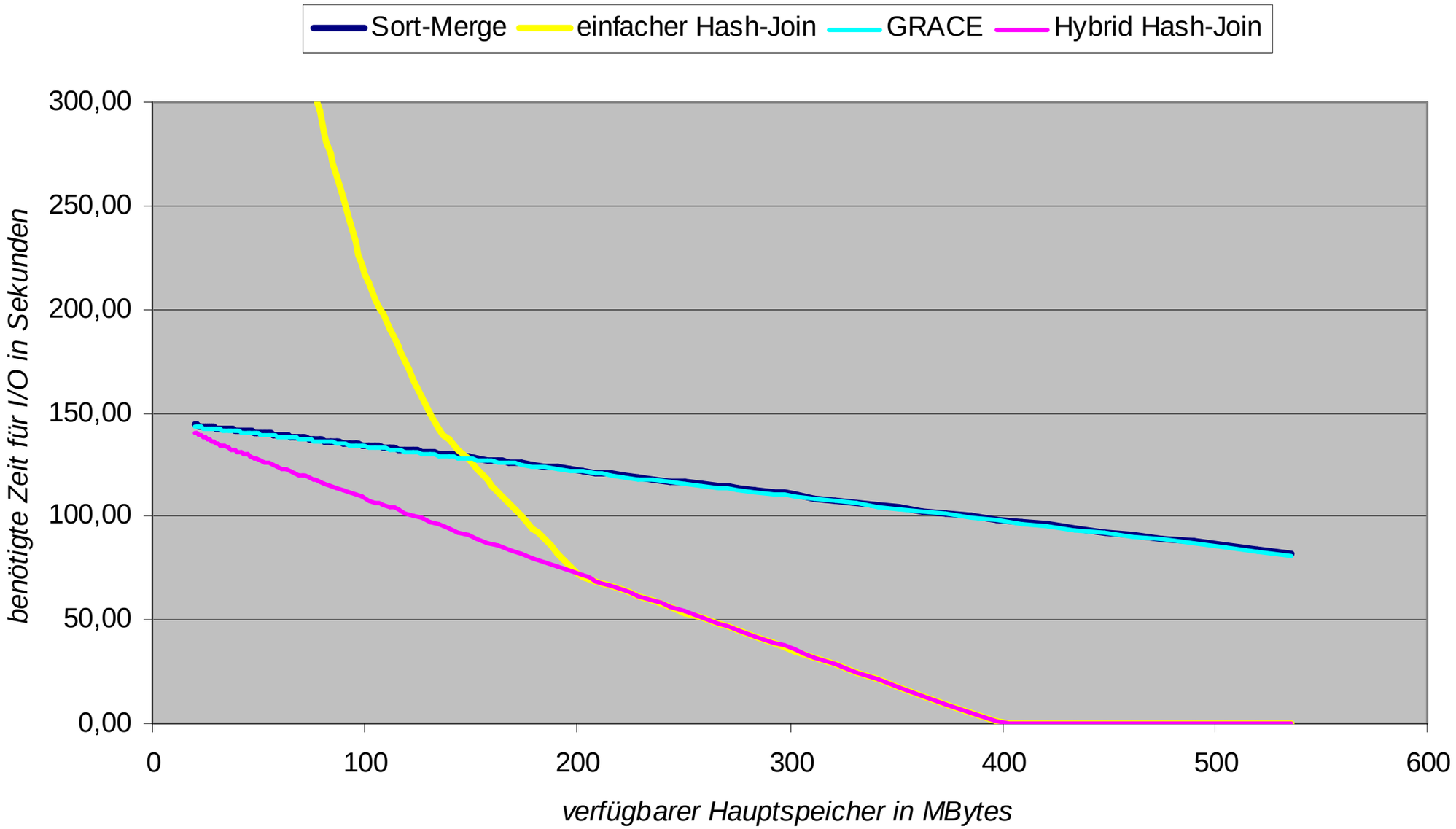
- verbindet die Vorteile von einfachen Hash-Join und GRACE-Join
  - wenn wenig Hauptspeicher ($\sqrt{|R|}$) zur Verfügung steht = GRACE-Join
  - wenn viel Hauptspeicher zur Verfügung steht, können große Teile schon in Phase 1 verjoint werden (→ einfacher Hash-Join)

I/O-Kosten-Vergleich der Algorithmen

- Hybrid Hash-Join dominiert einfachen Hash-Join
  - identisch, wenn mehr als |R|/2 Blöcke in M passen
  - sonst: einfacher Hash-Join schreibt und liest einige Blöcke mehrfach
- Hybrid Hash-Join dominiert GRACE Hash-Join
  - GRACE: $2 \cdot ((|R| + |S|) - \min(|R| + |S|, |M| - \sqrt{|R|}))$
  - Hybrid: $2 \cdot ((|R| + |S|) - q \cdot (|R| + |S|))$
- GRACE Hash-Join dominiert Sort-Merge-Join in typischen Fällen
  - Ausnahme: R und S sind ähnlich groß
- → Hybrid Hash-Join dominiert Sort-Merge-Join
  - Einfluss von Partition Overflows auf Laufzeit von Hash-basierten Algorithmen nicht im Paper beschrieben; Kostenabschätzungen sind zu optimistisch

I/O-Kosten-Vergleich der Algorithmen

- Beispiel
  - |R| = 800, |S| = 1600
  - Blockgröße: 512 kBytes, Lesen / Schreiben eines Blocks: 30 ms



Partition Overflow

- Annahme, dass man die Hashfunktion/Teilmengen so wählen kann, dass entstehende Partitionen genau die gewünschte Größe haben, ist unrealistisch
  - benötigt genaues Wissen über die Werteverteilung der Joinattribute in den Relationen
  - bei speziellen Verteilungen auch gar nicht möglich
- in der Realität werden manche Partitionen zu leer bleiben und andere ihre Größenbeschränkung überschreiten, also nicht mehr in M passen
  - nur relevant für Partitionen der kleineren Relation R

Partition Overflow - Lösungsansätze

- Partition Overflow auf Disk:
  - zu große Partitionen überarbeiten
    - teilen in 2 (oder mehr) kleinere Partitionen
    - oder: teilen in eine große Partition, die gerade in M passt und Rest zu einer anderen, zu leer gebliebenen Partition hinzufügen
- Partition Overflow in M (einfacher Hash):
  - einige Buckets der Hashtabelle in M zu den anderen, nicht bearbeiteten Tupeln, auf Disk schreiben
- Partition Overflow in M (Hybrid Hash): (R_0 wird zu groß)
  - einige Buckets zu einer neuen Partition R_{B+1} zuordnen und auf Disk schreiben

Abschluss

ENDE...

FRAGEN ???