universität wien

Kapitel 10

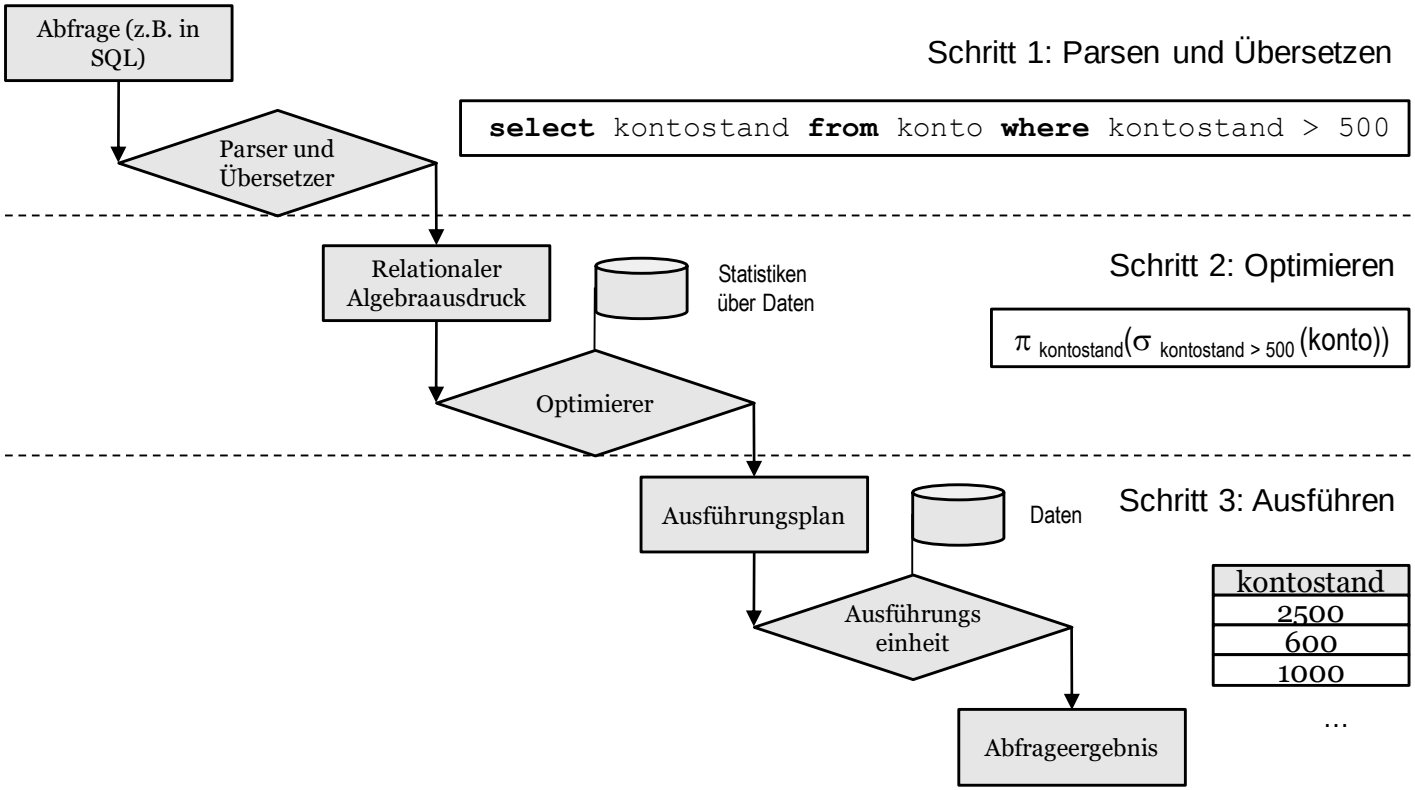
# Query Abarbeitung

# Inhalt



# 10.1 Überblick Query Abarbeitung

1. Parsen und Übersetzen (parser and translator)
2. Optimieren (optimizer)
3. Ausführen (evaluation engine)



# Grundlegender Ablauf der Verarbeitung

## Parsen und Übersetzen

- Parser prüft die **Syntax** und überprüft die Relationen
- Query wird in ihre interne Form übersetzt
- Diese wird weiter in **relationale Algebra** übersetzt
  - eine Menge äquivalenter Ausdrücke

## Optimieren

- Auswahl des „**günstigsten**“ Ausführungsplans aus den äquivalenten relationalen Algebraausdrücken

## Ausführen

- Die Ausführungseinheit (Query-Execution Engine) führt günstigsten **Ausführungsplan** (**Query-Evaluation Plan**) durch und retourniert die Ergebnisse der Query.

# Grundlegender Ablauf der Verarbeitung: Optimierung

Ein Ausdruck in relationaler Algebra hat oft **mehrere äquivalente Formulierungen**, z.B.:

$$\sigma_{kontostand>500}(\Pi_{kontostand}(konto))$$

ist äquivalent zu

$$\Pi_{kontostand}(\sigma_{kontostand>500}(konto))$$

Weiters steht für die Operationen der relationalen Algebra meist eine **Auswahl verschiedener Algorithmen** zur Verfügung

- Ein Ausdruck in relationaler Algebra kann also auf sehr viele Arten abgearbeitet werden

Ein annotierter Ausdruck, der die detaillierte Strategie der Durchführung festlegt, heißt **Ausführungsplan**, z.B.:

- Verwendung eines Index auf *kontostand* um Konten zu finden, für die *kontostand* > 500, ODER
- Durchlaufen der gesamten Relation und Verwerfen aller Konten, für die *kontostand* $\leq$ 500

# Grundlegender Ablauf der Verarbeitung: Optimierung (2)

**Query Optimierung**: Aus allen äquivalenten Ausführungsplänen wird derjenige mit den niedrigsten Kosten gewählt.

Die Kosten werden mittels statistischer Daten aus den Datenbanktabellen geschätzt.

z.B. Anzahl der Tupel in jeder Relation, Größe der Tupel, etc.

Dieses Kapitel behandelt:

- wie Query Kosten gemessen werden können
- Algorithmen zur Durchführung von Operationen der relationalen Algebra
- wie Algorithmen für Einzeloperationen kombiniert werden, um einen kompletten Ausdruck abzuarbeiten.

Nächstes Kapitel

- behandelt die Query Optimierung, also das Auffinden des Ausführungsplans mit den niedrigsten geschätzten Kosten

# 10.2 Maße für Query Kosten

Als **Kosten** wird generell die gesamte verstrichene **Zeit** gemessen, die zur Durchführung einer Query benötigt wird,

- **Viele Faktoren** tragen zu den Zeitkosten bei
  - Harddiskzugriffe, CPU, oder auch Netzwerk Kommunikation

**Harddiskzugriffe** sind typischerweise die **dominierenden Kosten**, und können auch relativ leicht geschätzt werden.

Sie werden gemessen unter Berücksichtigung von:

*Anzahl von Blocksuchen * durchschnittliche seek Kosten*

*Anzahl gelesener Blöcke * durchschn. block-read Kosten*

*Anzahl geschriebener Blöcke * durchschn. block-write Kosten*

Kosten zum Schreiben eines Blocks sind üblicherweise größer als die Kosten zum Lesen

- Daten werden nach dem Schreiben noch einmal gelesen, um zu verifizieren, dass das Schreiben erfolgreich war

# Maße für Query Kosten: Vereinfachung

Zur Vereinfachung verwenden wir nur die **Anzahl der Block Transfers** (Lesen und Schreiben) auf der Harddisk als Maß für die Kosten

- Wir ignorieren die unterschiedlichen Kosten von sequential und random I/O
- Wir ignorieren CPU Kosten

Kosten hängen von der **Größe des Puffers im Hauptspeicher** ab

- Mehr Hauptspeicher reduziert die benötigten Diskzugriffe
  - Die Größe des für den Puffer zur Verfügung stehenden Hauptspeichers hängt von gleichzeitig ausgeführten Prozessen ab und ist schwer vor der wirklichen Durchführung vorauszusehen
- Wir verwenden daher oft **worst-case** Abschätzungen, dass nur der minimal notwendige Speicherplatz zur Verfügung steht

Wir berücksichtigen daher nicht (im Gegensatz zu realen Systemen)

- CPU-Kosten,
- Puffergrößen und
- Unterschied zwischen sequentieller und randomisierter I/O

Weiters inkludieren unsere Kostenformeln nicht die Kosten, die beim (finalen) Schreiben der Ergebnisse auf die Harddisk anfallen

# 10.3 Selektion Operation

**File Scan** – Suchalgorithmen, die Tupel finden und zurückliefern, die einem Auswahlkriterium entsprechen.

- Verschiedene Algorithmen möglich

Algorithmus A1 - **Linear Search**:

Durchlaufe alle Blöcke des Files und teste alle Tupel, ob sie die Auswahlbedingung erfüllen.

- Kostenabschätzung (Anzahl der geprüften Diskblöcke) = $b_r$
  - $b_r$ Anzahl der Blöcke, die Tupel der Relation *r* enthalten
- Falls über ein **Schlüsselattribut** selektiert wird, *Kosten* = $(b_r/2)$
  - Suche kann bei Finden des gesuchten Tupels beendet werden, da es kein weiteres passendes Tupel mehr geben kann
  - Beachte: Keine Indexverwendung
- Linear search kann verwendet werden, unabhängig von
  - Auswahlbedingung
  - Sortierung der Tupel in der Datei, oder
  - Verfügbarkeit eines Index

# Selektion Operation (2)

A2 - **Binary Search**:

Anwendbar, falls die Selektion ein Attribut auf Gleichheit überprüft (exact match), nach dem die Datei **sortiert** ist

Annahme, dass die Blöcke einer Relation **zusammenhängend** (contiguously) auf der Disk gespeichert sind

Kostenabschätzung (Anzahl der zu prüfenden Diskblocks):

$\lceil log_2(b_r) \rceil$ - Kosten um das erste Tupel mittels binary search in den Blöcken zu finden

*Plus* Anzahl der Blöcke, die Tupel enthalten, welche das Selektionskriterium erfüllen

# Selektionen mit Verwendung eines Index

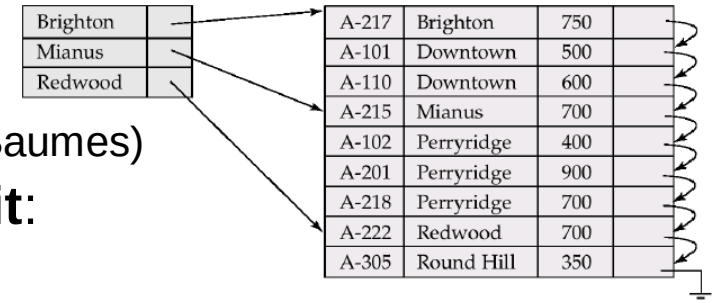
**Index Scan** – Suchalgorithmen, die einen Index benutzen

Auswahlbedingung muss den Suchschlüssel des Index betreffen.

A3 - **Primärindex auf Schlüsselkandidat, Gleichheit**:

Auffinden eines einzigen Tupels, das die zugehörige Gleichheitsbedingung erfüllt

*Kosten* = $HT_i + 1$ (HT … Height of Tree, Höhe des Baumes)




A4 - **Primärindex auf Nichtschlüssel, Gleichheit**:

Auffinden mehrerer Tupel

Tupel befinden sich in aufeinander folgenden (consecutive) Blöcken

*Kosten* = $HT_i$ + *Anzahl der Blöcke, die gefundene Tupel enthalten*

A5 - **Gleichheit zum Suchschlüssel eines Sekundärindex**:

Auffinden eines einzigen Tupels, falls der Suchschlüssel Schlüsselkandidat ist

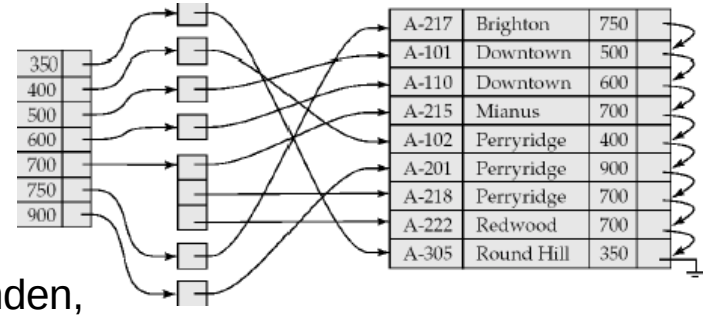
*Kosten* = $HT_i + 1$

Auffinden mehrerer Tupel, falls der Suchschlüssel kein Schlüsselkandidat ist

*Kosten* = $HT_i$ + *Anzahl gefundener Tupel*

Kann sehr teuer sein!

Jedes Tupel kann sich in einem anderen Block befinden, d.h. ein Blockzugriff für jedes gefundene Tupel



# Selektionen mit Vergleichen

Selektionen der Form $\sigma_{A \leq V}(r)$ oder $\sigma_{A \geq V}(r)$ können realisiert werden durch Verwendung von

Linear Search oder Binary Search,

oder durch Nutzung von Indizes in einer der folgenden Arten:

A6 - **Primärindex, Vergleich**: (Relation ist nach A sortiert)

Für $\sigma_{A \geq V}(r)$ verwende den Index um das erste Tupel $\geq v$ zu finden und durchsuche die Relation von da an sequentiell

Für $\sigma_{A \leq V}(r)$ arbeite die Relation sequentiell ab, bis das erste Tupel $> v$; verwende nicht den Index

A7 - **Sekundärindex, Vergleich**:

Für $\sigma_{A \geq V}(r)$ verwende Index um den ersten Indexeintrag $\geq v$ zu finden und durchlaufe den Index sequentiell von da an, um Zeiger auf die Tupel zu finden.

Für $\sigma_{A \leq V}(r)$ durchlaufe die Blatt-Seiten des Index um Zeiger auf die Tupel zu finden, bis der erste Eintrag $> v$ auftritt

In jedem Fall, Zugriff auf die Tupel, auf die gezeigt wird

benötigt eine I/O Operation für jedes Tupel

Linear search kann billiger sein, falls sehr viele Tupel geliefert werden müssen

# 10.4 Sortieren

Wir können einen **Index** über die Relation **aufbauen**, und diesen benutzen, um die Relation in sortierter Folge zu lesen

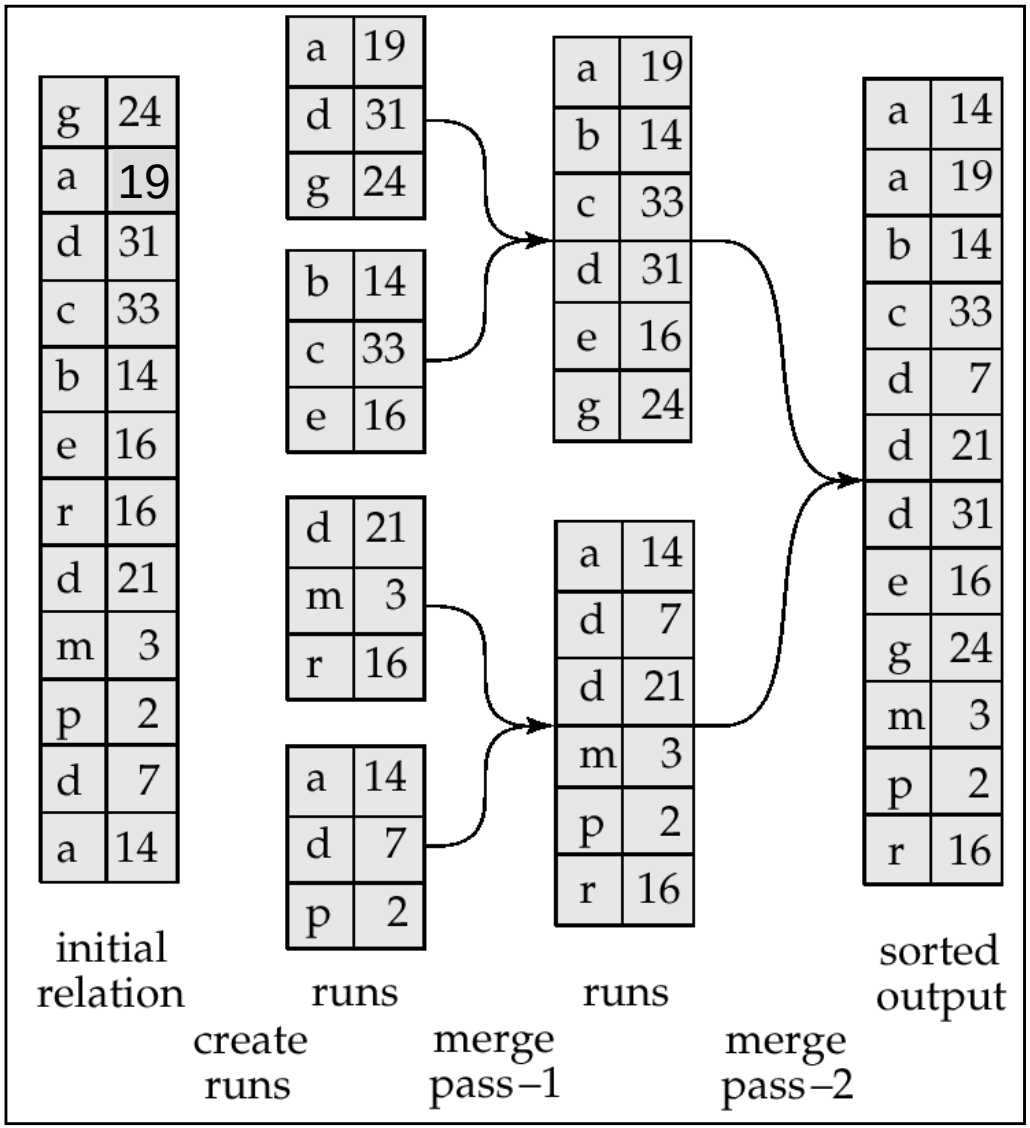
- Da Tupel nur logisch aber nicht physisch organisiert sind, kann das zu einem Diskzugriff für jedes gefundene Tupel führen.

Für Relationen, die ins Memory passen, können Techniken wie Quicksort verwendet werden

Für Relationen, die nicht ins Memory passen, ist **externes merge-sort** eine gute Wahl

Vergleiche mit Balanced Multiway Merging und Polyphase Merging aus AD

# Beispiel: Externes Merge-Sort




Annahme:

Ein Tupel pro Block

3 Blöcke im Memory

# 10.5 Verbund (Join) Operation

Verschiedene Algorithmen zur Durchführung von Joins

- Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
- Merge-join
- Hash-join

Auswahl basierend auf Kostenabschätzung

Beispiele verwenden folgende Informationen

| | *kunde* | *kontoinhaber* |
|---|---|---|
| Anzahl der Tupel: | 10000 | 5000 |
| Anzahl der Blöcke: | 400 | 100 |
| Anzahl der Tupel pro Block: | 25 | 50 |

# Nested-Loop Join

Zur Berechnung des Theta Join $r \bowtie_\theta s$

**for each** Tupel $t_r$ **in** $r$ **do begin**
  **for each** Tupel $t_s$ **in** $s$ **do begin**
    teste Paar $(t_r, t_s)$ auf Erfüllung der Join Bedingung $\theta$
    **if** Bedingung erfüllt, füge $t_r \bullet t_s$ zum Ergebnis hinzu.
  **end**
**end**

$r$ heißt die **äußere Relation** und $s$ die **innere Relation** des Join.

Benötigt keine Indizes und kann mit jeder beliebigen Join Bedingung verwendet werden.

**Teuer**, da **jede möglich Paarung von Tupeln** der beiden Relationen untersucht wird.

# Nested-Loop Join (2)

Im worst case, wenn pro Relation nur ein Block ins Memory passt, sind die geschätzten Kosten

$$n_r * b_s + b_r$$

Diskzugriffe.

Falls die kleinere Relation komplett ins Memory passt, wird diese als innere Relation verwendet. Die Kosten reduzieren sich damit zu $b_r + b_s$ Diskzugriffen.

Unter worst case Annahmen ist die Kostenabschätzung

5000 ∗ 400 + 100 = 2,000,100 Diskzugriffe *kontoinhaber* äußere Relation,
10000 ∗ 100 + 400 = 1,000,400 Diskzugriffe *kunde* als äußere Relation.

Falls die kleinere Relation (*kontoinhaber)* komplett ins Memory passt, ergibt die Kostenabschätzung 500 Diskzugriffe.

Block nested-loop Algorithmus ist vorzuziehen.

# Block Nested-Loop Join

Variante des nested-loop Join bei der **jeder Block** der inneren Relation **mit jedem Block** der äußeren Relation gepaart wird.

```
for each Block B_r of r do begin
    for each Block B_s of s do begin
        for each Tuple t_r in B_r do begin
            for each Tuple t_s in B_s do begin
                Teste, ob (t_r,t_s) die Join Bedingung erfüllt
                Falls ja, füge t_r • t_s zum Ergebnis hinzu.
            end
        end
    end
end
```

Worst case Schätzung: $b_r * b_s + b_r$ Blockzugriffe.

Jeder Block der inneren Relation *s* wird für jeden Block der äußeren Relation einmal gelesen (statt einmal für jedes Tupel der äußeren Relation)

Best case: $b_r + b_s$ Blockzugriffe.

# Indexed Nested-Loop Join

**Indexzugriffe** können Filedurchläufe ersetzen, falls

- Join ein **equi-join** *oder* **natural join** ist und
- ein **Index** für das Join Attribut der inneren Relation **verfügbar** ist
  - Index kann für die Berechnung eines Joins extra erzeugt werden.

Für jedes Tupel $t_r$ der äußeren Relation *r,* verwende den Index zum Suchen von Tupeln in *s,* **die die Join Bedingung erfüllen**.

Worst case: Puffer hat nur Platz für eine Seite aus *r* und für jedes Tupel aus *r*, führen wir einen Indexzugriff auf *s* durch.

Kosten des Join: $b_r + n_r * c$

- Wobei *c* die Kosten für das Durchlaufen des Index und das Holen aller zu einem Tupel von *r* passenden Tupel von *s* bezeichnet
- *c* kann geschätzt werden als Kosten einer einzelnen Selektion auf *s* unter Verwendung der Join Bedingung.

Sind Indizes verfügbar für die Join Attribute beider Relationen *r* und *s,* verwende die Relation mit weniger Tupeln als äußere Relation.

# Beispiel der Kosten für Indexed Nested-Loop Join

*kontoinhaber* ⋈ *kunde,* mit *kontoinhaber* als äußere Relation.

*kunde* mit primären $B^+$-Baum Index über Join Attribut *kunden_name,* mit 20 Einträgen in jedem Indexknoten

Da *kunde* 10,000 Tupel enthält, ist die Höhe des Baumes 4, plus ein zusätzlicher Zugriff um eigentliche Daten zu finden

*kontoinhaber* hat 5000 Tupel

Kosten eines block nested loop Join

100*400 + 100 = 40,100 Diskzugriffe unter worst case Memory Bedingungen (kann bedeutend weniger sein mit mehr Memory)

Kosten eines indexed nested loop Join

100 + 5000*5 = 25,100 Diskzugriffe.

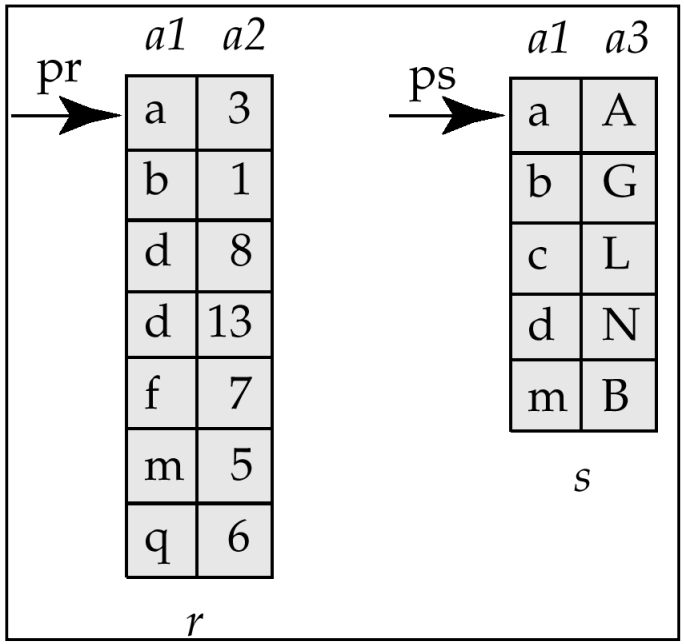
CPU Kosten wahrscheinlich geringer als für block nested loop Join

# Merge-Join

**Sortiere** beide Relationen nach dem Join Attribut (falls sie nicht bereits sortiert sind).

**Mische** die sortierten Relationen um Join durchzuführen

- Join Schritt ist ähnlich zum Mischen im merge-sort Algorithmus.
- Hauptunterschied ist die Behandlung doppelter Werte im Join Attribut — Jedes Paar mit gleichen Werten bezüglich des Join Attributs muss berücksichtigt werden.




Kann **nur für equi-joins und natural joins** verwendet werden

Jeder Block wird nur einmal gelesen

- Annahme, dass alle Tupel für jeden Wert des Join Attributs ins Memory passen

Die Anzahl der Blockzugriffe für merge-join ist daher

- $b_r + b_s$ + *Kosten für Sortierung falls Relationen unsortiert*

# Hash-Join

Anwendbar für **equi-joins und natural joins**.

Eine **Hashfunktion** *h* partitioniert die Tupel beider Relationen r und s in n Partitionen

- Wähle Hashfunktion so, dass die einzelnen Partitionen einer Relation (r oder s) vollständig in den Hauptspeicher passen

*h* bildet *JoinAttrib*-Werte auf { 0, 1, ..., *n-1* } ab

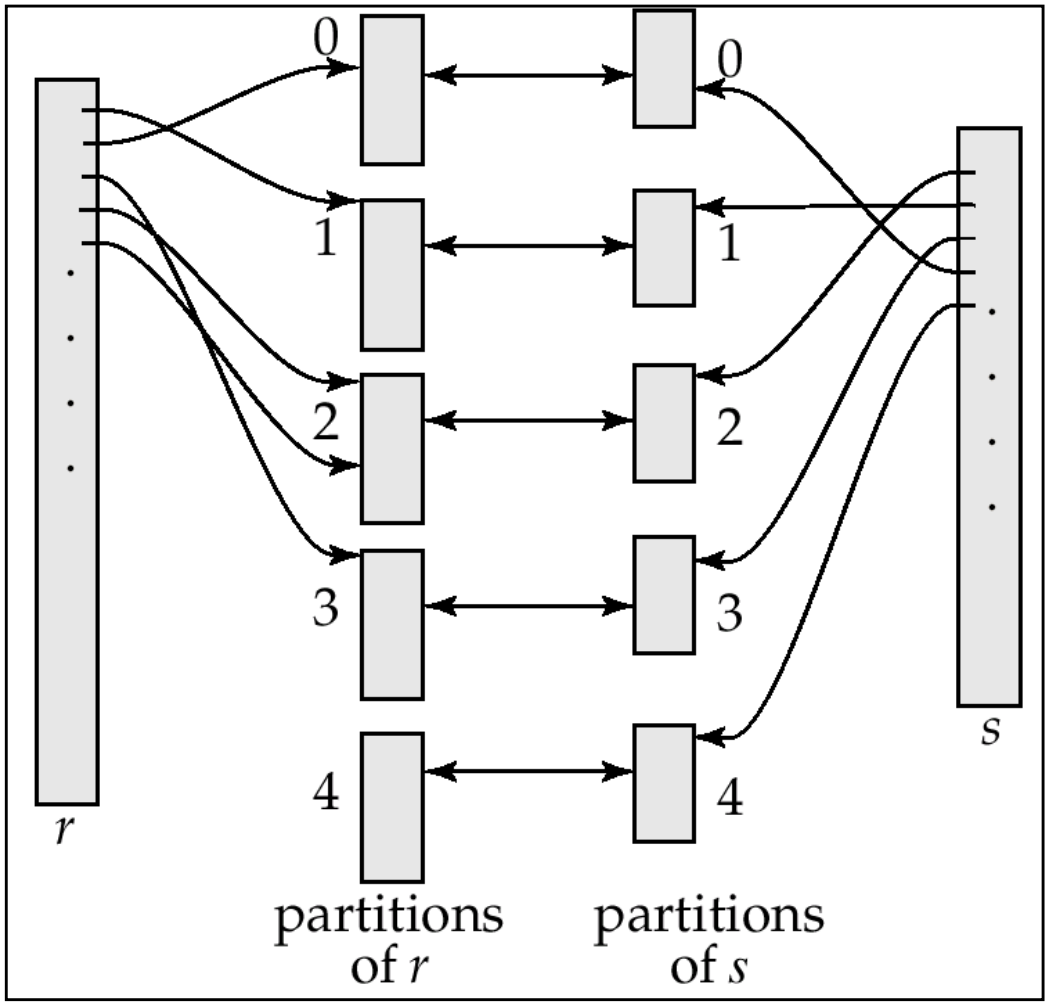
- *JoinAttrib* bezeichnet die gemeinsamen Attribute von *r* und *s*, die im natural join verwendet werden.
- $r_0, r_1, \ldots, r_{n-1}$ bezeichnen Partitionen von Tupeln aus *r*
  - Jedes Tupel $t_r \in r$ fällt in Partition $r_i$ mit $i = h(t_r\ [JoinAttrs])$.
- $s_0, s_1, \ldots, s_{n-1}$ bezeichnen Partitionen von Tupeln aus *s*
  - Jedes Tupel $t_s \in s$ fällt in Partition $s_i$, mit $i = h(t_s\ [JoinAttrs])$.

***r* Tupel in $r_i$ müssen nur mit *s* Tupeln in $s_i$ verglichen werden**

Vergleiche mit *s* Tupeln aus anderen Partitionen sind nicht notwendig:

- ein *r* Tupel und ein *s* Tupel, die die Join Bedingung erfüllen, haben den gleichen Wert für die Join Attribute.
- Ist der Hashwert für diesen Wert gleich *i*, dann muss das *r* Tupel in $r_i$ liegen und das *s* Tupel in $s_i$.

# Hash-Join (2)



# Hash-Join Algorithmus

Hash-Join von *r* und *s* wird wie folgt berechnet:

1. Partitioniere Relation *s* mit Hashfunktion *h*.
2. Partitioniere *r* genauso.
3. For each Partition *i:*

   (a) Lade $s_i$ ins Memory und berechne einen in-memory Hash Index über dem Join Attribut. Dieser Hash Index verwendet eine andere Hashfunktion als die vorher verwendete *h.*

   (b) Lese Tupel aus $r_i$ eines nach dem anderen von der Disk. Für jedes Tupel $t_r$ finde passende Tupel $t_s$ in $s_i$ mit Hilfe des in-memory Hash Index. Gib die Konkatenation der Attribute aus.

Relation *s* heißt **build input** und
*r* heißt **probe input.**

# Beispiel für Kosten von Hash-Join

*kunde* ⋈ *kontoinhaber*

Annahme der Memorygröße mit 20 Blöcken

$b_{kontoinhaber}$= 100 und $b_{kunde}$ = 400.

*kontoinhaber* wird als build input verwendet. Partitionierung in fünf Teile, jeder 20 Blöcke groß. Diese Partitionierung kann in einem Durchlauf vorgenommen werden.

Demgemäß Partitionierung von *kunde* in fünf Teile mit einer Größe von jeweils 80 Blöcken. Ebenfalls in einem Durchlauf machbar.

$Lesen_{part} + Schreiben_{part} + Lesen_{vergleichen}$

Gesamtkosten daher: 3(100 + 400) = 1500 Blocktransfers

wenn man das Schreiben von nur teilweise gefüllten Blocks ignoriert

# 10.6 Andere Operationen

**Eliminierung von Duplikaten** kann mittels Hashing oder Sortieren durchgeführt werden.

- Durch Sortieren werden alle Duplikate benachbart, und alle bis auf eines kann gelöscht werden.
- **Optimierung***:* Bei externem merge-sort können Duplikate schon während des Generierens eines Run und auch während der Mischschritte beseitigt werden.
- Hashing ist ähnlich – Duplikate kommen in das selbe Bucket.

**Projektion** wird implementiert durch Ausführen der Projektion auf jedem Tupel und anschließende Eliminierung von Duplikaten.

# 10.7 Auswertung von Ausdrücken

Bisherige Algorithmen bewerkstelligen individuelle Operationen

Zwei Alternativen um einen ganzen Ausdrucksbaum abzuarbeiten

**Materialisation**

Generiere das Ergebnis eines (Teil-) Ausdrucks, dessen Inputs Relationen sind, oder bereits berechnet wurden. Materialisiere (speichere) das Ergebnis auf der Disk. Wiederhole.

**Pipelining**

Gib Tupel an die aufrufende Operation zurück, während die Operation selbst noch weiter ausgeführt wird.

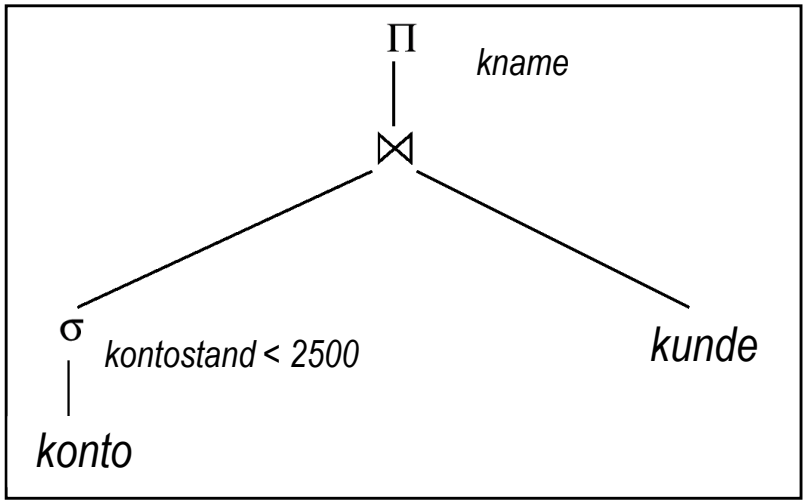
## Materialisierte Auswertung

Auswertung einer Operation zu jedem Zeitpunkt, beginnend in der tiefsten Ebene. Verwende die in temporären Relationen materialisierten Zwischenergebnisse zur Auswertung der Operationen der nächsten Stufe.

z.B. in der Abbildung: Berechnen und Speichern von

$\sigma_{kontostand < 2500}(konto)$

dann Berechnen u. Speichern des Joins mit *kunde,* und schließlich Berechnen der Projektion auf *kname.*



# Materialisation (2)

Auswertung mit Materialisation kann immer angewendet werden

Kosten des Schreibens der Ergebnisse auf die Disk und des Zurücklesens können ziemlich hoch werden.

Unsere Formeln für die Kosten von Operationen ignorieren die Kosten für das Schreiben, daher:

*Gesamtkosten = Summe der Kosten der einzelnen Operationen + Kosten des Schreibens der Zwischenergebnisse*

**Double Buffering**

Verwende zwei Outputpuffer für jede Operation, sobald einer voll ist, wird er auf die Disk geschrieben, während der andere weiter gefüllt werden kann.

Erlaubt Overlap der Schreibzugriffe mit der Berechnung und verringert daher die für die Ausführung der Operation benötigte Zeit

# Pipelining

Beim **Pipelining** werden mehrere Operationen gleichzeitig durchgeführt, wobei eine Operation die Ergebnisse direkt an die nächste weitergibt.

- z.B. im vorherigen Beispiel, kein Speichern des Ergebnisses von

$$\sigma_{kontostand < 2500}(konto)$$

  sondern direkte Weitergabe der Tupel an den Join. Ergebnisse des Join werden auch nicht gespeichert, sondern direkt an die Projektion weitergegeben.
- Viel billiger als Materialisation, da kein Speichern einer temporären Relation auf der Disk notwendig.

Pipelining ist nicht immer möglich – z.B.: sort, hash-join.

- Damit Pipelining effektiv ist, verwende Algorithmen, die bereits Ausgabetupel produzieren können, wenn noch nicht alle Eingabetupel der Operation gelesen wurden.