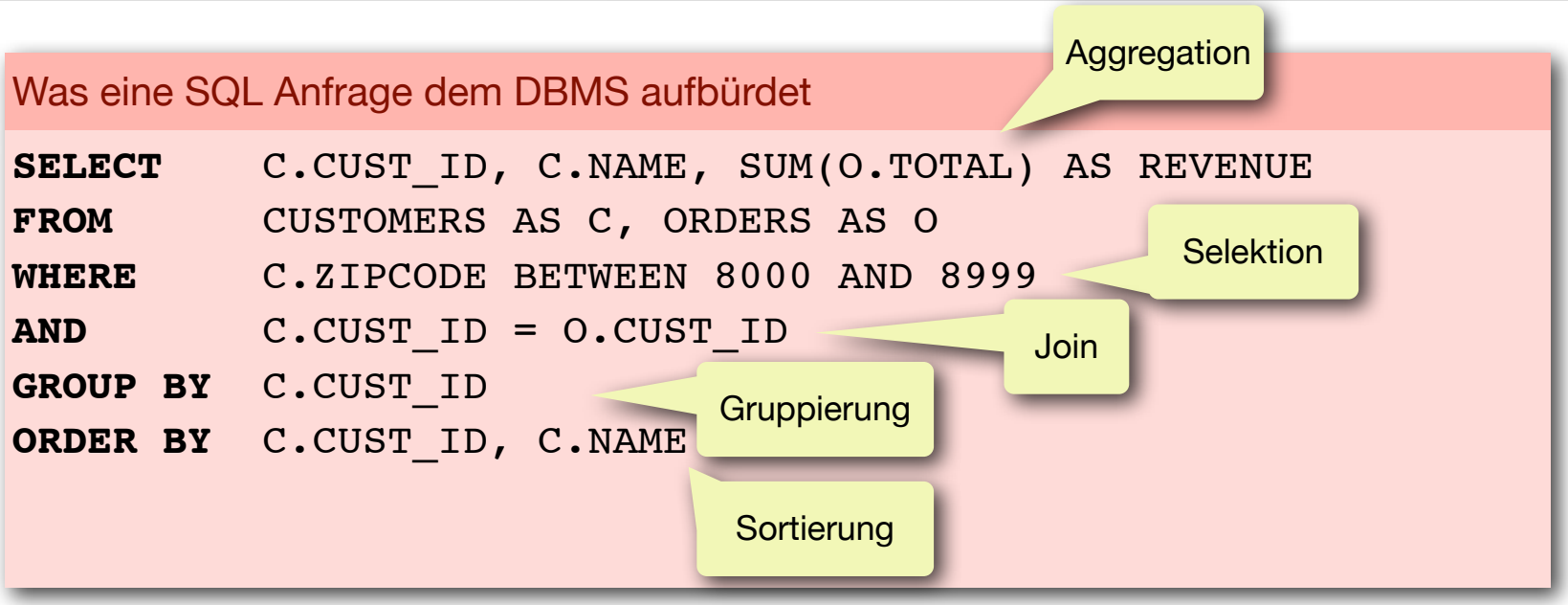
Kapitel 6
Externes Sortieren

- Überblick
- Two-Way Merge Sort
- External Merge Sort
- Kostenminimierung
- B+ Bäume zur Sortierung

Anfragebearbeitung



Der **Query Prozessor** eines DBMSs muss eine Reihe von Aufgaben unter erschwerenden Bedingungen erfüllen, denn

- **Speicherresourcen** sind limitiert,
- die zu verarbeitenden **Datenmengen** sind groß,
- eine Anfrage muss **so schnell wie möglich** beantwortet werden.

Überblick

Architektur eines DBMS

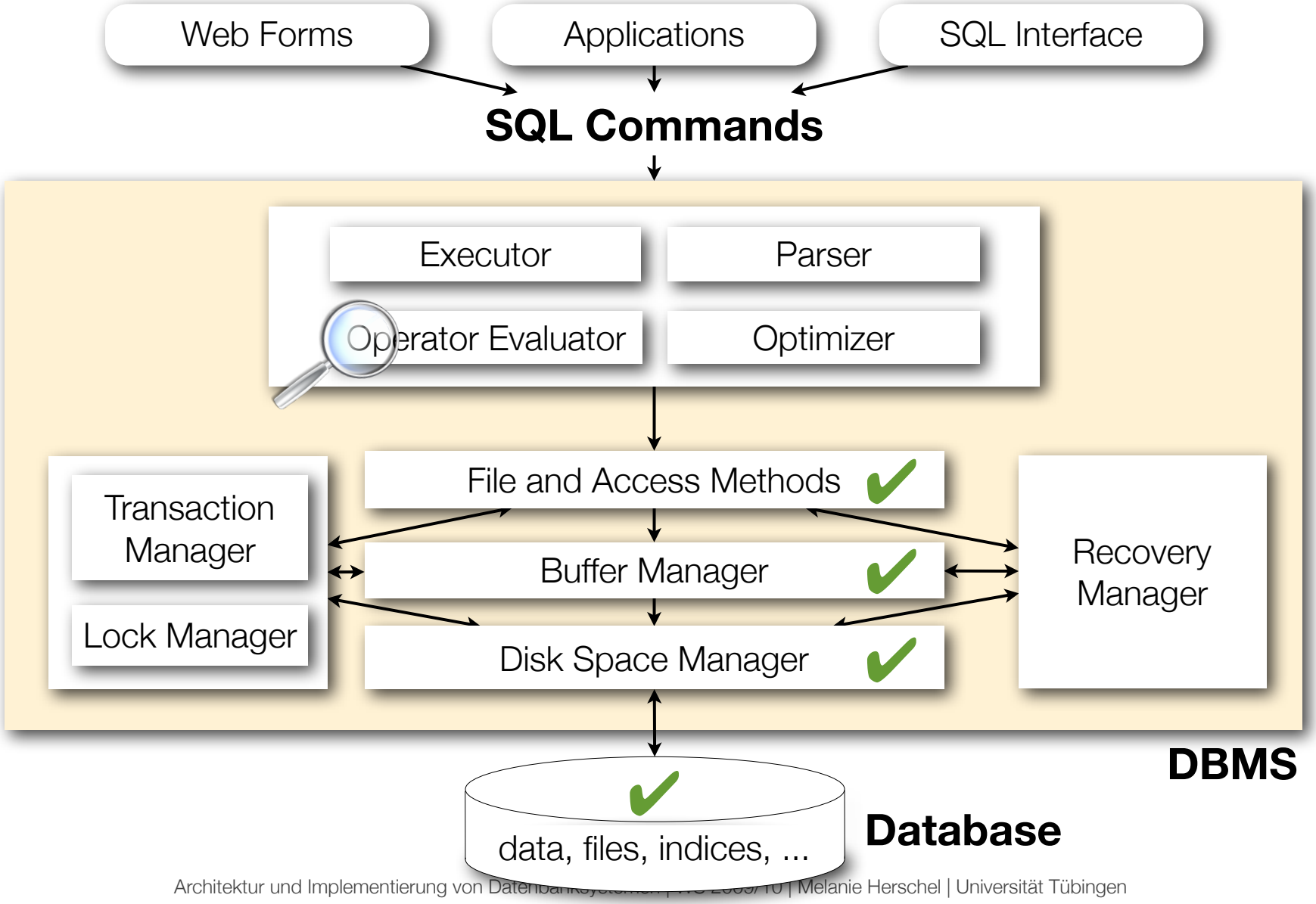


Figure inspired by Ramakrishnan/Gehrke: "Database Management Systems", McGraw-Hill 2003.

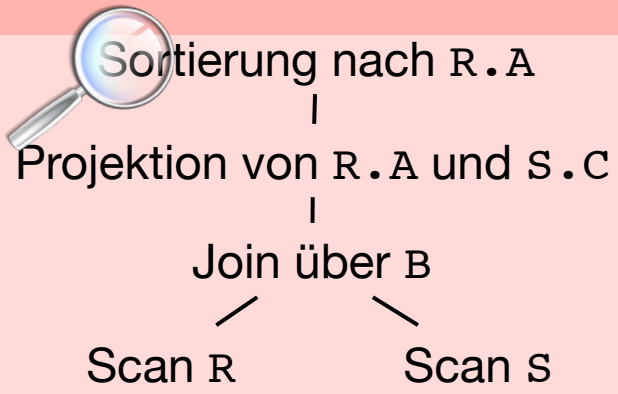
Anfragepläne und Operatoren

Operatoren als Bausteine einer Anfrage

- Ein DBMS sieht eine SQL Anfrage nicht als einen monolithischen Block.
- Stattdessen wird eine Anfrage in kleinere Teile unterteilt: **Anfrageoperatoren**.
- Operatoren können zu einem Anfragegraphen kombiniert werden. Dieser Graph entspricht dem sogenannten **Anfrageplan**.
- Dieser Anfrageplan ist in der Lage, die gestellte **SQL Anfrage korrekt zu beantworten**.
- Für jeden Operator gibt es **spezielle Implementierungen**, die eine Aufgabe gut bewältigen (z.B. Zeit- und Speichereffizienz).

Beispiel für einen Anfrageplan

```
SELECT     R.A, S.C
FROM       R, S
WHERE      R.B = S.B
ORDER BY   R.A
```



Sortierung während der Anfragebearbeitung

Sortierung ist ein nützlicher Operator, der entweder explizit oder implizit verwendet wird.

Explizite Sortierung via SQL `ORDER BY` Klausel

```
SELECT      A, B, C
FROM        R
ORDER BY    A
```

Implizite Sortierung, z.B. zur Duplikateliminierung

```
SELECT DISTINCT    A, B, C
FROM               R
```

Implizite Sortierung, z.B. zur Vorbereitung eines Equijoins.

```
SELECT    R.A, S.C
FROM      R, S
WHERE     R.B = S.B
```

Außerdem: Gruppierung via GROUP BY, Bulk Loading eines B+ Baumes, ...

Definition von Sortierung

Eine Datei ist bezüglich eines **Sortierschlüssels *k*** und einer **Reihenfolge θ** sortiert, wenn für zwei beliebige Records r_i und r_j, wobei r_i dem Record r_j in der Datei vorangeht gilt, dass ihre jeweiligen Schlüsselwerte der θ-Reihenfolge entsprechen:

$$r_i \; \theta \; r_j \quad \Leftrightarrow \quad r_i.k \; \theta \; r_j.k$$

Ein **Schlüssel** kann sowohl **aus einem oder aus mehreren Attributen** bestehen. Im letzteren Fall entspricht die Reihenfolge der **lexikographischen** Reihenfolge.

Gegeben $k = (A, B)$ und $\theta = <$, so gilt

$$r_i < r_j \quad \Leftrightarrow \quad r_i.A < r_j.A$$
$$\vee \; (r_i.A = r_j.A \wedge r_i.B < r_j.B)$$

Externes Sortieren

- Ein fundamentales **Ziel** eines DBMS ist es, keine Beschränkung bezüglich der Größe der zu verarbeitenden Daten festlegen zu müssen.

➡Fundamentales **Problem**:

*Wie können wir die Records einer Datei sortieren, wenn die **Datei deutlich größer ist als der zur Verfügung stehender Hauptspeicherplatz** (bzw. der vom Buffer Manager verwaltete Speicherplatz)?*

- Wir nähern uns der Aufgabe in zwei Schritten:

  (1) Sortierung einer Datei beliebiger Größe ist selbst dann möglich, wenn wir nicht mehr als **drei freie Seiten** im Buffer zur Verfügung haben.

  (2) Erweiterung des in (1) erarbeiteten Algorithmus, um effektiver **größere** (realistischere) **Mengen freien Speichers auszunutzen**.

- Außerdem betrachten wir eine Reihe von Optimierungen, die die **Anzahl nötiger I/O Operationen reduziert**.

Kapitel 6
Externes Sortieren

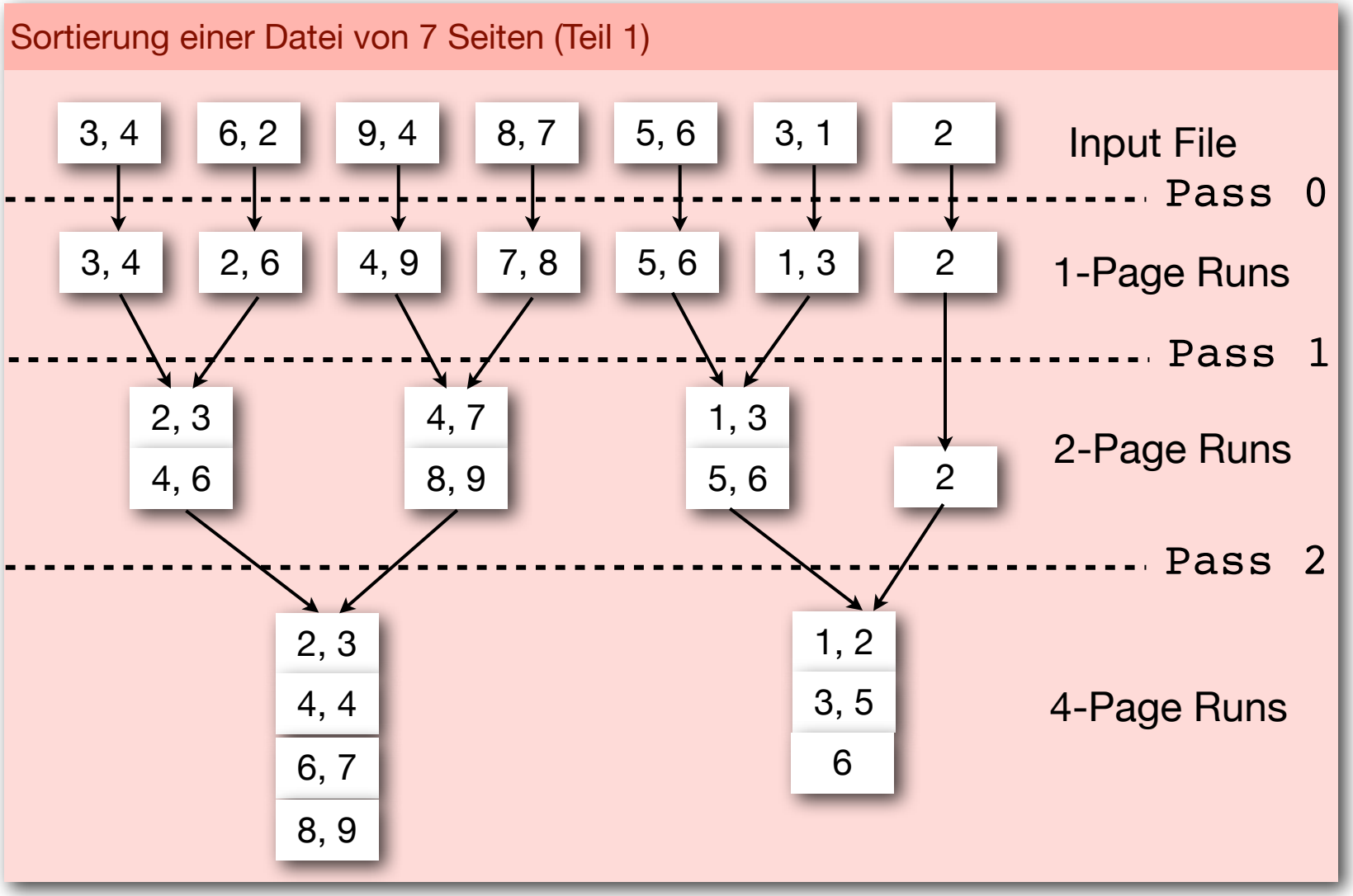
- Überblick
- Two-Way Merge Sort
- External Merge Sort
- Kostenminimierung
- B+ Bäume zur Sortierung

Two-Way Merge Sort

- Benötigt nur **drei Seiten** im Buffer Pool.
- Sortiert eine Datei mit 2^k Seiten in mehreren **Passes**.
- In jedem Pass werden eine Menge von sortierten Teildateien, sog. **Runs**, generiert.
  - Pass 0 sortiert jede der 2^k Seiten separat, und schreibt jede sortiere Datei auf Platte ➡ 2^k Runs.
  - Jeder weitere Pass (**Merge-Pass**) vereinigt Paare von Runs zu jeweils neuen, doppelt so großen Runs ➡ Pass n produziert 2^{k-n} Runs.
  - Pass k generiert einen einzigen Run, der der sortierten Datei entspricht.
- In jedem Pass betrachten wir jede Seite der Input Datei
  ➡Es sind $k \times N$ Lese- und $k \times N$ Schreiboperationen nötig, um die Datei zu sortieren.

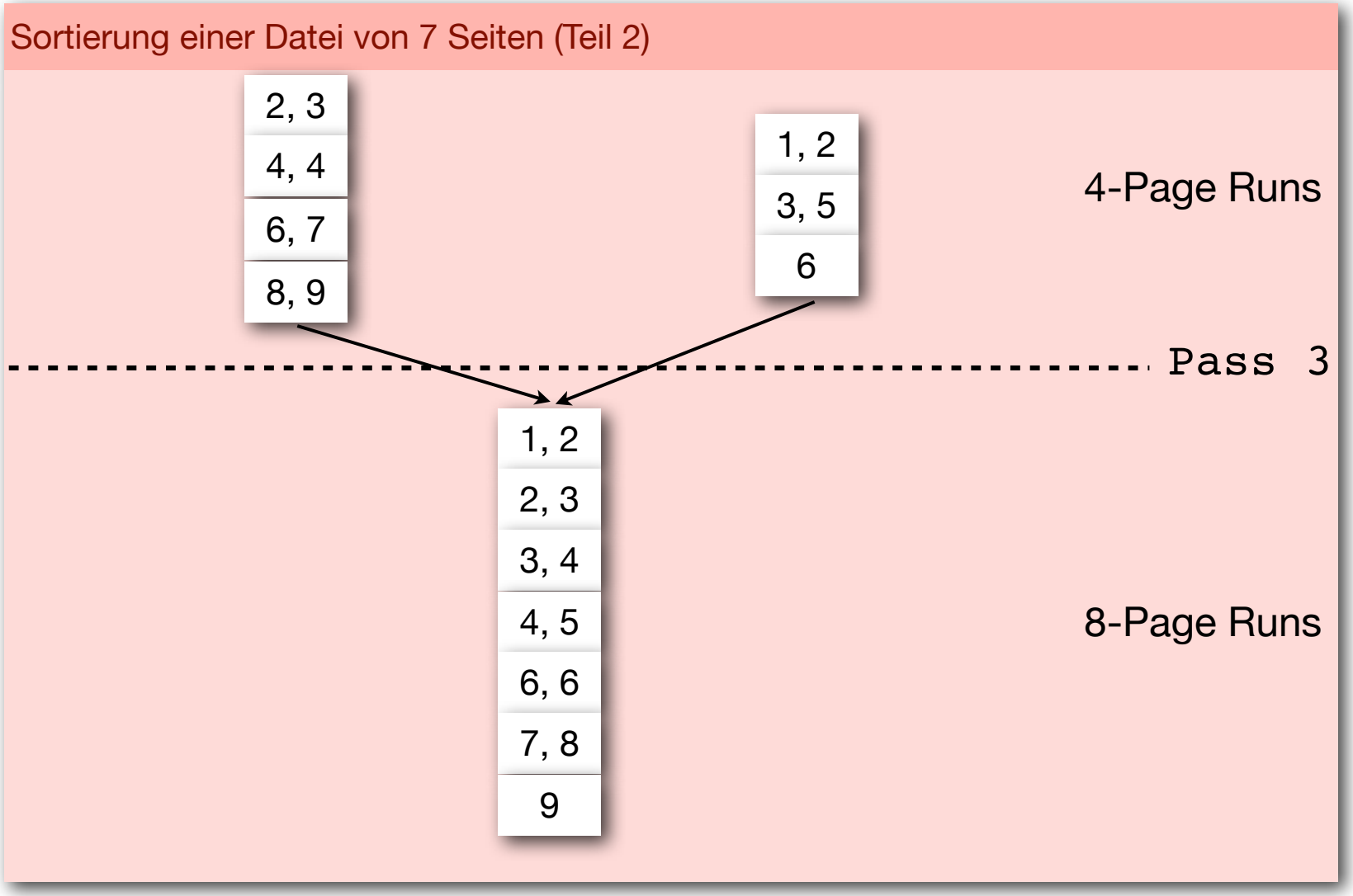
Two-Way Merge Sort

Beispiel



Two-Way Merge Sort

Beispiel



Two-Way Merge Sort

Algorithmus

Two-Way Merge Sort einer Datein von N = 2k Seiten

```
function two_pass_merge_sort(file, N)
begin
   for each Seite p in file do
      Lese p in den Hauptspeicher, sortiere p, und schreibe p auf Platte
      zurück.
   for n in 1 ... k do
      for r in 0 ... 2^(k-n) - 1 do
      begin
      Vereinige Runs 2r und 2r + 1 aus verhergehendem Pass in einen neuen
      Run, indem die Eingabe-Runs Seitenweise gelesen werden;
      Lösche Eingabe-Runs 2r und 2r + 1;
      end
   Gebe letzten verbleibenden Run zurück;
end
```

Two-Way Merge Sort

Anzahl I/O Operationen

- Um eine Datei von *N* Seiten zu sortieren, führen wir pro Pass durch:
  - Das Lesen von *N* Seiten
  - Sortieren und Mergen
  - Das Schreiben von *N* Seiten

➡2 *N* I/O Operationen pro Pass.

- Anzahl Passes: $1 + \lceil log_2 N \rceil$

➡Anzahl I/O Operationen **insgesamt**: $2\,N\,(\,1 + \lceil log_2 N \rceil\,)$

Wie viele I/O Operationen brauchen wir, um eine 8GB Datei (Seitengröße = 8 KB) zu sortieren?

Kapitel 6
Externes Sortieren

- Überblick
- Two-Way Merge Sort
- External Merge Sort
- Kostenminimierung
- B+ Bäume zur Sortierung

External Merge Sort

Annahme

- Wir haben *B* Seiten im Buffer zur Verfügung.
- Wie sortieren eine Datei, die aus *N* Seiten besteht.

Grundidee

- Wie bei Two-Way Merge Sort führen wir mehrere Passes über die Seiten der zu sortierenden Datei aus.
- Durch Verwendung von *B* statt 3 Frames des Buffers minimieren wir die Anzahl Passes (und somit die Anzahl I/O Operationen).

Lösungsansatz

(1) Pass 0: Reduktion der Anzahl Runs, die initial im ersten Pass produziert werden.

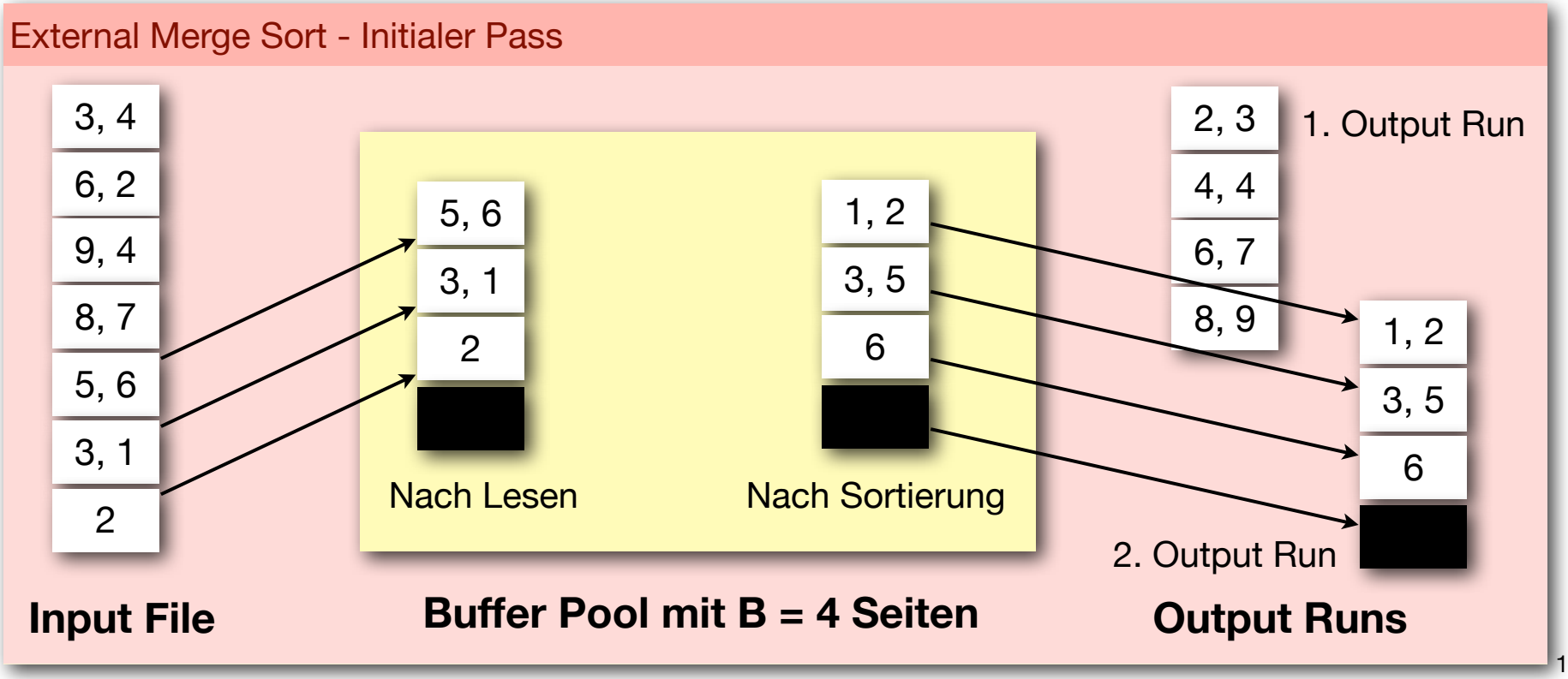
(2) Pass i, i > 0: Reduktion der Anzahl weiterer Passes, indem während eines Passes mehr als zwei Runs auf einmal miteinander gemerged werden.

External Merge Sort

Initialer Pass

- Wir lesen *B* Seiten in den Hauptspeicher ein.
- Die Daten werden im Hauptspeicher sortiert (nach beliebigem in-memory Sortieralgorithmus, z.B. Quicksort).

➡ $\lceil N/B \rceil$ Runs, jeweils der Größe *B* (letzter Run eventuell auch kleiner).

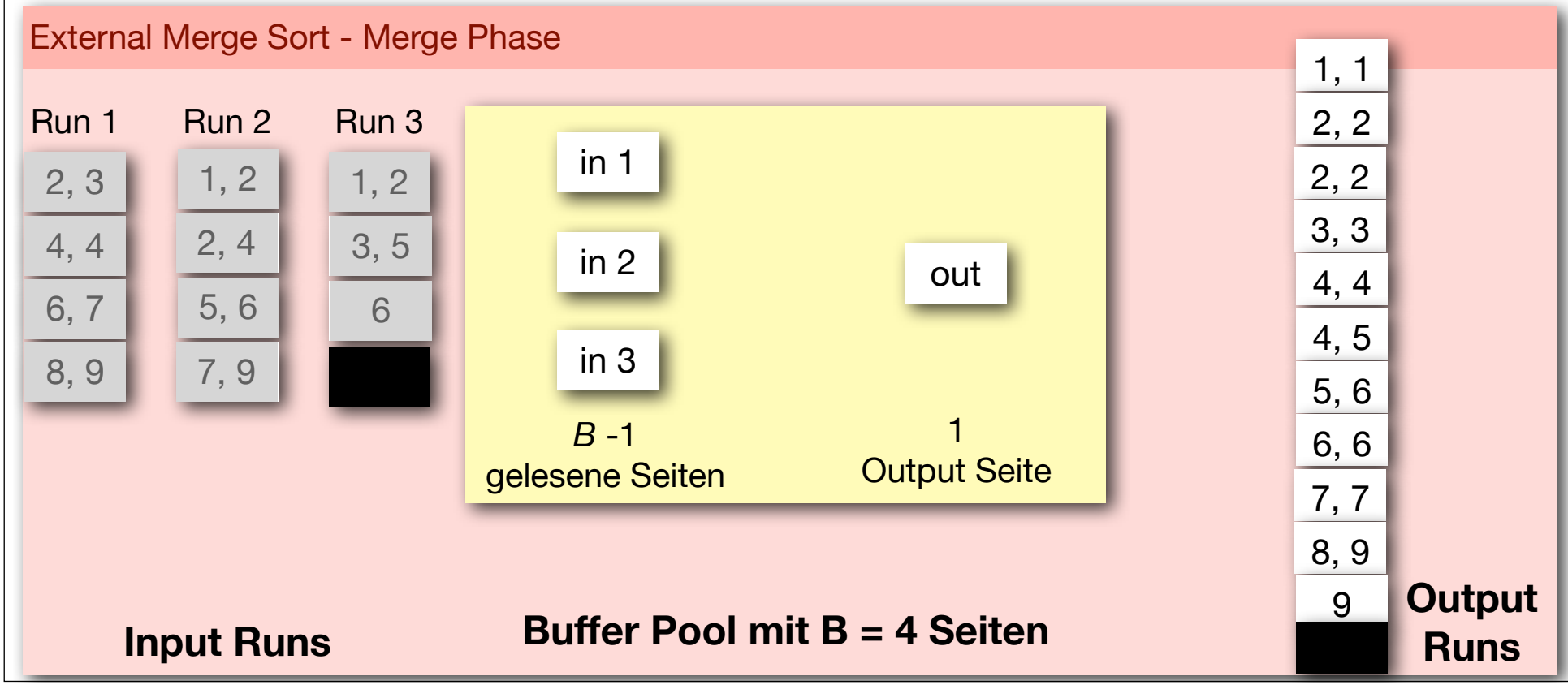


External Merge Sort

Merge-Passes (Pass *i, i > 0*)

- Verwende *B* - 1 Seiten des Buffers, um Seiten einzulesen.
- Verwende verbleibende Seite um die Ausgabe (seitenweise) zu produzieren und zu schreiben.

➡ Kein 2-Way Merge Sort, sondern ein (*B*-1)-Way Merge Sort.



External Merge Sort

Anzahl Passes

Schritt (1): Reduktion der in Pass 0 initialisierten Runs

- In Pass 0 lesen wir die gesamte Datei (*N* Seiten) und produzieren $N_1 = \lceil N/B \rceil$ Runs statt *N* beim Two-Way Merge Sort.

Schritt (2): Reduktion der Anzahl der Merge Passes

- Statt der $\lceil log_2 N \rceil$ Passes, die in Merge Phase beim Two-Way Merge Sort nötig sind, benötigen wir nun $\lceil log_{B-1} N_1 \rceil$ Passes.

➡Insgesamt 1 + ⌈logB-1 ⌈N/B⌉ ⌉ Passes

- Da in der Praxis B >> 2 ist, kann die Anzahl Passes über die Datei, und somit die Anzahl I/O Operationen, reduziert werden.

External Merge Sort

Anzahl Passes

Anzahl Passes bei External Merge Sort

N	B = 3	B = 5	B = 9	B = 17	B = 129	B = 257
100	7	4	3	2	1	1
10,000	13	7	5	4	2	2
1,000,000	20	10	7	5	3	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

External Merge Sort

Algorithmus

Algorithmus External Merge Sort gegeben B zur Verfügung stehende Buffer Frames

```
procedure external_merge_sort(file)
begin
    Lese B Seiten in den Buffer ein;
    Sortiere Daten dieser B Seiten;
    Schreibe sortierte Daten als einen Run der Größe B;
    while Anzahl Runs am eine eines Passes > 1 do
        while es gibt Runs des verhergehenden Pass, die gemerged werden können do
            Wähle nächste B - 1 Runs (des vorhergendenden Pass);
            Lese jeden der gewählten Runs in einen Input Buffer seitenweise ein;
            Merge diese B - 1 Runs seitenweise und schreibe Ergebnis in den Output Buffer;
            Schreibe den Output Buffer seitenweise auf Platte zurück;
        end
end
```

External Merge Sort

Minimierung der Anzahl Initialer Runs

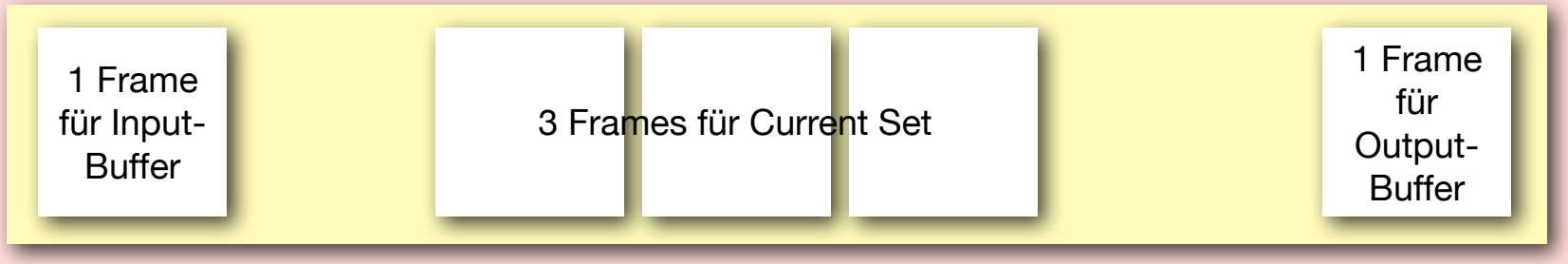
- Wir haben gesehen, dass in Pass 0 Runs der konstanten Größe *B* produziert werden.
- In der Merge-Phase profitieren wir von dieser geringeren Anzahl Runs (im Vergleich zu Two-Way Merge Sort), da weniger Runs (in weniger Passes) gemerged werden müssen.

➡ Idee: können wir die Anzahl initialer Runs weiter verringern?

Replacement Sort

- Vergrößert die Größe eines initialen Runs von *B* auf durchschnittlich *2B*.
- Teilt den zur Verfügung stehenden Hauptspeicher (*B* Frames) in einen **Input-Buffer**, ein **Current Set** und einen **Output-Buffer** auf.

Beispielhafte Aufteilung des Hauptspeichers bei Replacement Sort, gegeben B = 5



External Merge Sort

Algorithmus für Replacement Sort

Replacement Sort

Annahmen

Input-Buffer = 1 Frame;

Output-Buffer = 1 Frame;

Current Set = B - 2 Frames;

(1) Öffne eine leere Datei, in die wir aktuellen Run schreiben.

(2) **Lade** die jeweils nächste Seite der zu sortierenden Datei **in den Input-Buffer**. Gibt es keine Seiten mehr zu laden, gehe zu Schritt (4).

(3) Solange im Current Set freier Speicher existiert, so **verschiebe ein Record aus dem Input-Buffer in das Current Set** (wenn Input-Buffer leer, gehe zu Schritt (2)).

(4) Aus dem Current Set, wähle ein Record *r* mit dem **kleinsten Sortierschlüsselwert *k*, so dass $k \geq k_{out}$** ist. Dabei ist k_{out} **der maximale Sortierschlüsselwert im Output-Buffer** (oder -∞, falls dieser leer ist). Ist der Output-Buffer voll, hänge dessen Inhalt dem aktuellen Run an.

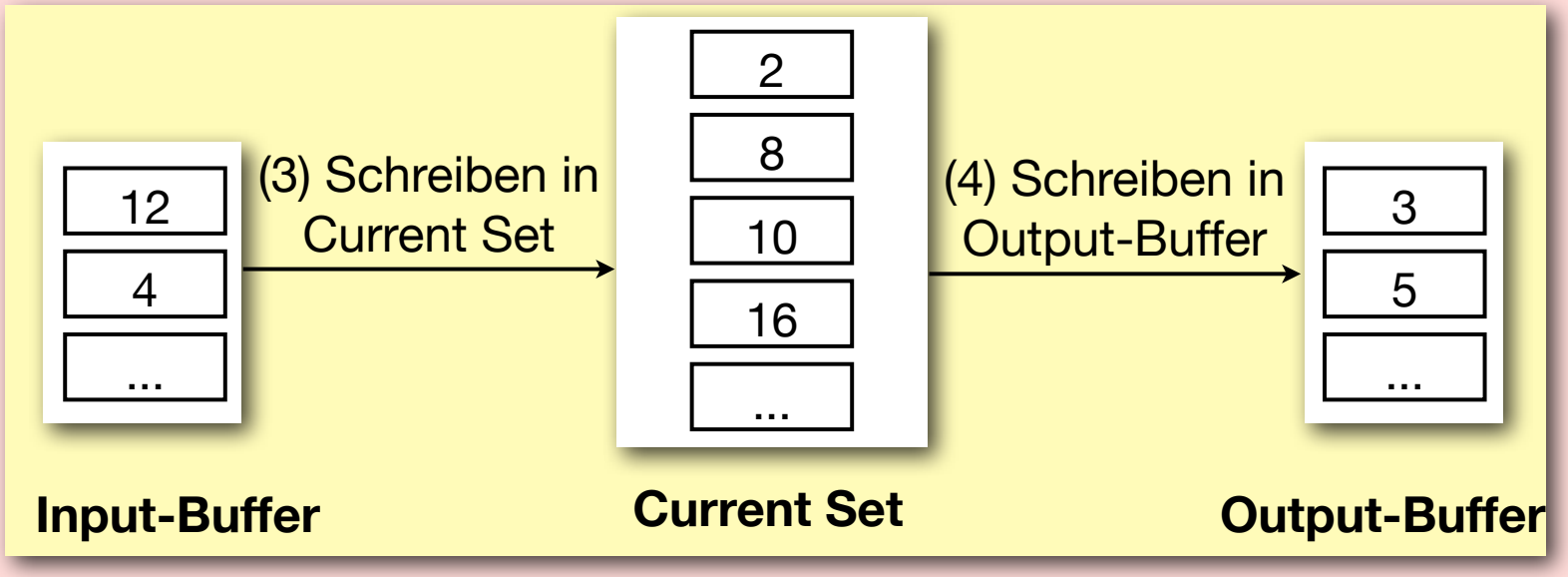
(5) Sind **alle *k* des Current Set < k_{out}**, so hänge den Output-Buffer dem aktuellen Run an, und **schließe den aktuellen Run**. Ein neuer Run wird nun begonnen.

(6) Wurden alle Daten der Eingabedatei verarbeitet, stop. Sonst gehe zurück zu Schritt (3).

External Merge Sort

Beispiel für Replacement Sort

Beispiel: Record mit Schlüssel $k = 8$ wird als nächstes in den Output-Buffer geschrieben. Momentan haben wir $k_{out} = 5$.



- Im Current Set ist 8 der kleinste Schlüssel, der ≥ dem maximalen Schlüssel im Output-Buffer (aktuell 5) ist
  ➡ 8 wird als nächstes in den Output Buffer geschrieben.

- Das Record mit Schlüsselwert $k = 2$ kann nicht mehr dem aktuellen Output-Buffer (und somit Run) hinzugefügt werden, da es kleiner 5 ist.
  ➡Wird in nächsten Run geschrieben.

External Merge Sort

Beispiel für Replacement Sort

Replacement Sort Schritt für Schritt verfolgen

Wir nehmen an, dass *B* = *6* und die Größe des Current Sets = 4. Die Eingabedatei enthält die folgenden `INTEGER` Werte:

503, 087, 512, 061, 908, 170, 897, 275, 426, 154, 509, 612

Schreiben Sie einen Trace für Replacement Sort, indem Sie die unten stehende Tabelle ausfüllen. Markieren Sie das Ende eines Runs durch <EOR>. Das Current Set wurde bereits bis Schritt (3) des Algorithmus gefüllt.

current set	output
503, 087, 512, 061	

Kapitel 6
Externes Sortieren

- Überblick
- Two-Way Merge Sort
- External Merge Sort
- Kostenminimierung
- B+ Bäume zur Sortierung

Kostenminimierung

I/O Kostenminimierung vs. Minimierung der I/O Operationen

Bisher: Kostenbetrachtung nur auf Basis der Anzahl I/O Operationen

- Anzahl I/O Operationen nur eine Approximation der wirklichen Kosten.
- Ignoriert
  - Optimierungen, wie z.B. Blocked I/O (sequentielles Lesen).
  - Zusätzliche Kosten, wie z.B. CPU Kosten.

Verfeinerung von External Merge Sort

- Berücksichtigung von Blocked I/O ➡ Buffer Blocks
- Berücksichtigung der CPU Kosten ➡ Double Buffering

Kostenminimierung

Berücksichtigung von Blocked I/O

- Wenn nur Anzahl **I/O Operationen minimiert** werden, ist das Ziel, die **Anzahl Passes zu minimieren**.
- Betrachten wir zusätzlich den Vorteil von **blocked I/O**, der die durchschnittliche Lesezeit eines zusammenhängenden Blocks von Seiten minimiert, so kann es von Vorteil sein, Blöcke von zusammenhängenden Seiten, sog. **Buffer Blocks, in den Buffer einzulesen**.
- Grundidee von blocked I/O: Alloziiere *b* Seiten für jeden Input-Buffer (statt nur einer Seite).
- Dies **reduziert die durchschnittlichen I/O-Kosten** zum Lesen einer Seite um Faktor $\approx b$.
- Diese Änderung führt zu einem **geringeren Fan-In**, da wir mit dem gleichen freien Speicherplatz von *B* Seiten nur noch $\lceil (B - b) / b \rceil$ Runs in einem Pass mergen können.
- In der Praxis sind Hauptspeichergrößen groß genug, um Dateien in **einem einzigen Merge-Pass zu sortieren**, selbst mit blocked I/O.

Kostenminimierung

Double Buffering

Bisher: CPU Berechnungen und I/O Operationen werden sequentiell durchgeführt.

1. Daten werden zuerst in Input Buffer gelesen (I/O).

2. Danach wird im Hauptspeicher sortiert bzw. gemerged (CPU).

3. Seiten des Output-Buffers werden auf die Festplatte geschrieben (I/O).

Verwendung von Double Buffering

- Grundidee: führe **I/O und CPU Operationen parallel** aus.
- Verfahren: Lege jeden Buffer **doppelt** an *(shadow buffers)*. Die **CPU wechselt zwischen den beiden Kopien** sobald ein Input-Buffer leer ist. In diesem Fall wird **asynchron eine I/O Operation gestartet**, die den “verbrauchten” Buffer neu füllt. Der Output-Buffer wird analog verwaltet.
- Ist die Zeit, einen Block zu lesen kleiner als die Zeit, einen Block zu konsumieren, so ist mit dieser Double Buffering Technik die CPU immer beschäftigt.
- Nachteil: doppelt soviel Hauptspeicher ist nötig.

Kapitel 6
Externes Sortieren

- Überblick
- Two-Way Merge Sort
- External Merge Sort
- Kostenminimierung
- B+ Bäume zur Sortierung

B+ Bäume zur Sortierung

Frage: Gegeben ein B+ Index über einen Sortierschlüssel k, können wir Die Daten mit Hilfe des Index effizient in sortierter Reihenfolge abrufen?

Clustered Index

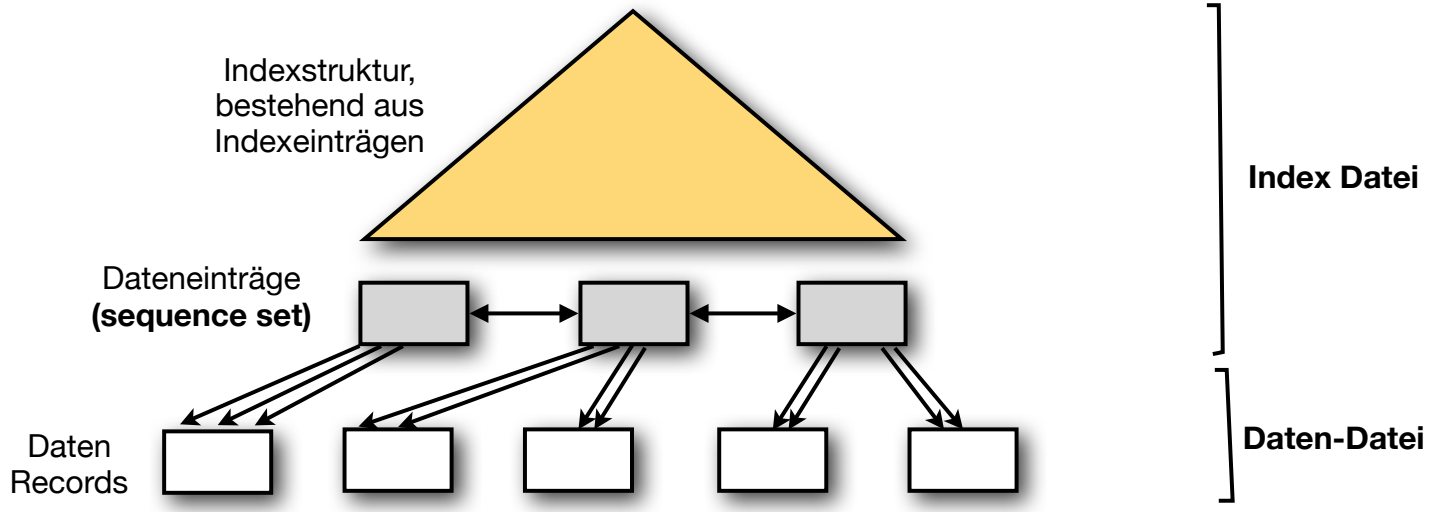
✔

Unclustered Index

✘

B+ Bäume zur Sortierung

Clustered Index



Sortieralgorithmus

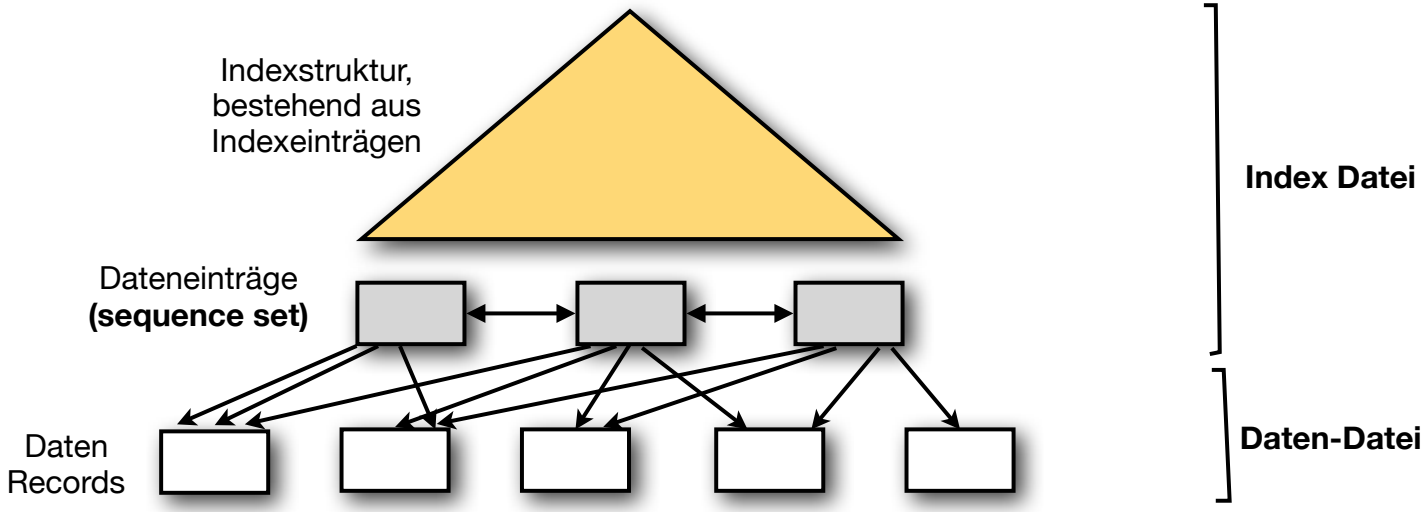
(1) Finde am weitesten links liegenden Dateneintrag im Sequence Set.
Kosten: Baumhöhe, 3 - 4 I/Os

(2) Traversiere jede Seite im Sequence Set.
Kosten: Anzahl Seiten des Sequence Set = Bruchteil der Seiten der Daten-Datei.

(3) Ermittle Daten Records, die Dateneinträgen entsprechen.
Kosten: Da Sequence Set und Daten-Datei die gleiche Reihenfolge aufweisen (clustered Index), benötigen wir nur einen Pass über die Daten Datei.

B+ Bäume zur Sortierung

Unclustered Index



Sortieralgorithmus

- Schritt (1) und (2) wie bei clustered Index.

(3) Ermittle Daten Records, die Dateneinträgen entsprechen.
Kosten: Da Sequence Set und Daten-Datei unterschiedliche Reihenfolge aufweisen (unclustered Index), benötigen wir im schlimmsten Fall 1 I/O pro Dateneintrag!

- Worst Case muss nicht erreicht sein, die tatsächlichen Kosten hängen davon ab, wie sich die Reihenfolgen unterscheiden, welche Seiten im Buffer Pool sind, ...

Zusammenfassung

Two-Way Merge Sort

- Minimale Nutzung des Hauptspeichers (3 Seiten).
- Hohe Anzahl I/O Operationen, da viele Passes nötig sind.

External Merge Sort

- Minimierung der Anzahl Passes durch verstärkte Nutzung des Hauptspeichers.
- Minimierung der initialen Anzahl Runs durch Replacement Sort.

Kostenminimierung

- Anzahl I/O Operationen vs. tatsächliche Kosten.
- Blocked I/O
- Double Buffering

B+ Bäume zur Sortierung

- Clustered Indizes eignen sich dafür.
- Die Eignung von unclustered Indizes hängt von vielen Faktoren ab und in der Praxis werden diese nicht zur Sortierung von Daten verwendet.