Dateiorganisation und Zugriffsstrukturen

Mögliche Dateiorganisationen

- Viele Alternativen existieren, jede geeignet für bestimmte Situation (oder auch nicht)
  - Heap-Dateien: Geeignet für Zugriffsmuster wie File Scan (d.h. Lesen aller Sätze)
  - Sortierte Dateien: Ideal, wenn Sätze in bestimmter Ordnung zurückgegeben werden sollen oder nur ein bestimmter Bereich (Range) benötigt wird
  - Hash-Dateien: Gut für wahlfreien Zugriff (Lookup-Operation mit Gleichheitsbedingung)
    - Datei ist eine Menge von Buckets. Bucket = Primärseite mit einer oder mehrerer Überlauf-Seiten
    - Hash-Funktion **h**(*r*) = Bucket, in den der Satz r gehört. **h** berücksichtigt nur einige der Felder von *r*, genannt *Suchfelder (können Primärschlüssel sein)*

Kostenmodell

- Zur Vereinfachung werden CPU-Kosten ignoriert (Fokus auf I/O-Kosten):
  - **B:** Anzahl der Datenseiten
  - **R:** Anzahl der Sätze pro Seite
  - **D:** (Durchschnittliche) Zeit zum Lesen oder Schreiben einer Seite auf Platte
  - Messung der Anzahl von Seiten-I/O's ignoriert Vorteile aus dem Prefetching mehrerer hintereinander folgender Seiten; somit sind I/O Kosten nur approximiert.
  - Analyse des Durchschnittsfalls; basiert auf verschiedenen verein-fachten Annahmen.

Operationen im Kostenmodell

- Scan
  - Lesen aller Sätze aus einem File (von Platte in den Puffer)
- Search mit Gleichheitsbedingung (Lookup)
  - Lesen aller Sätze, die eine Gleichheitsbedingung erfüllen
  - Beispiel: “Finde den Studenten-Satz für den Studenten mit der *sid* = 23“
  - Lesen von Seiten, die gesuchte Sätze enthalten
  - Gesuchte Sätze aus der Seite geholt
- Search mit Bereichsauswahl (Range Selection)
  - Lesen aller Sätze, die in einem bestimmten Bereich liegen
  - Beispiel: “Finde alle Studenten-Sätze mit Namen, die im Alphabet hinter Smith liegen“
- Insert
  - Einfügen eines neuen Satzes in die Datei
  - Identifizieren der Seite, Einlesen, Modifizieren und Zurückschreiben
- Delete
  - Löschen eines Satzes mit der Satz-ID
  - Modifizieren der Seite (vgl. Insert)
  - Evtl. Veränderung anderer Seiten erforderlich (je nach Dateiorganisation)

Kosten der Operationen

	Heap Datei	Sortierte Datei	Hash Datei
Scan aller Sätze	**BD**	**BD**	**1.25 BD**
Lookup	**0.5 BD**	**D $\log_2$B**	**D**
Range Search (Bereichssuche)	**BD**	**D ($\log_2$B + # of pages with matches)**	**1.25 BD**
Insert	**2D**	**Search + BD**	**2D**
Delete	**Search + D**	**Search + BD**	**2D**

Verschiedene Annahmen liegen diesen groben Schätzwerten zugrunde (Verarbeitung der einzelnen Sätze ignoriert)

Indexe

- Ein Index auf einer Datei beschleunigt die Suche nach Sätzen entsprechend der Werte von Schlüsselfeldern
  - Jede Teilmenge der Felder einer Relation kann Suchschlüssel für einen Index auf der Relation sein
  - Suchschlüssel ist nicht das gleiche wie Schlüssel (minimale Menge von Attributen, die einen Satz in einer Relation eindeutig identifiziert)
    - Suchschlüssel kann Primärschlüssel sein
    - Suchschlüssel kann auch Sekundärschlüssel sein (der nicht mal eindeutig sein muß)
- Ein Index enthält eine Menge von Dateieinträgen und unterstützt ein effizientes Wiederauffinden aller Dateneinträge **k*** mit einem gegebenen Schlüsselwert **k**.

Alternativen für Dateneinträge im Index

- Drei Alternativen, wie Dateneinträge aussehen können:
  1. Datensatz mit Schlüsselwert **k**
  2. <**k**, ID des Datensatzes mit demWert des Suchschlüssels **k**>
  3. <**k**, Liste von Datensatz-IDs mit Suchschlüssel **k**>
- Vorteile und Nachteile der Alternativen abwägen (z.B. bessere Speicherauslastung bei 3., dafür aber variable Länge der Dateneinträge)
- Auswahl einer Alternative für Dateneinträge ist orthogonal zur Indexierungstechnik, die genutzt wird, um Dateneinträge mit einem gegebenen Schlüsselwert **k** zu plazieren.
  - Beispiele für Indexierungstechniken: B+ Bäume, hash-basierte Strukturen
  - Typischerweise enthält Index Hilfsinformationen, die die Suche zu den gewünschten Dateneinträgen lenkt

Alternativen für Dateneinträge im Index (Forts.)

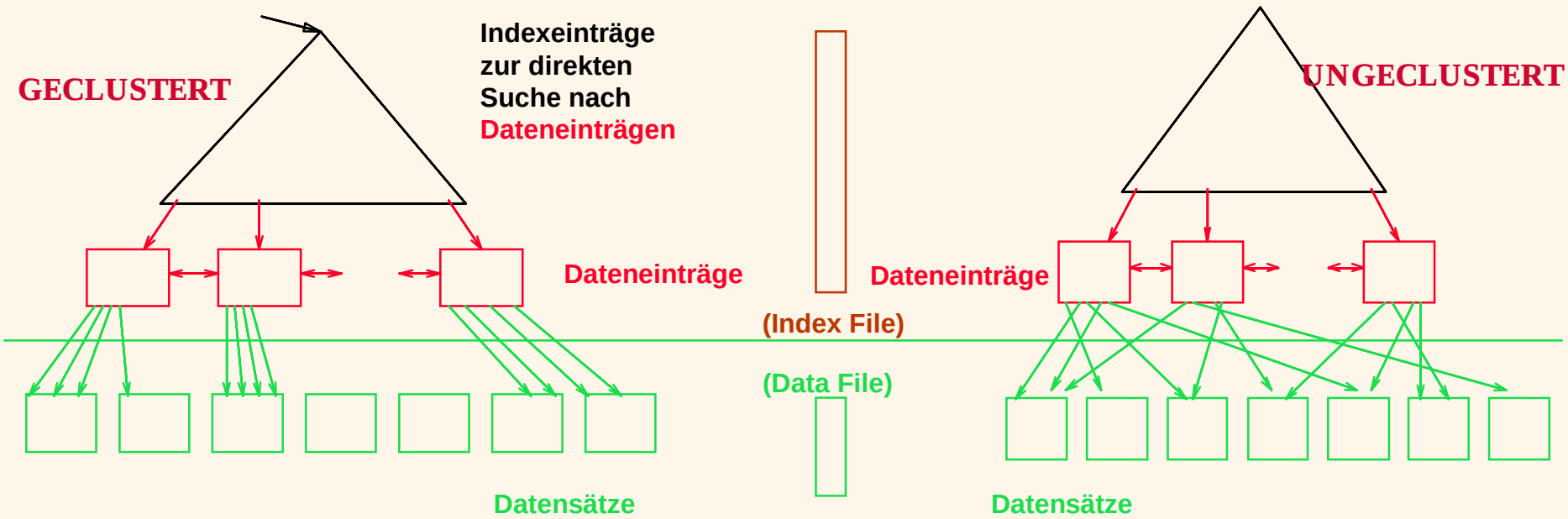
- Alternative 1:
  - Indexstruktur ist gleich Dateiorganisation für die Datensätze (wie Heap-Datei oder sortierte Datei)
  - Höchstens ein Index auf einer gegebenen Menge von Datensätzen kann Alternative 1 nutzen. (Anderenfalls Duplizierung der Datensätze, führt zur redundanter Speicherung und potentieller Inkonsistenz)
  - Wenn Datensätze sehr groß, ist die Anzahl der Seiten, die Dateneinträge enthalten, sehr groß. Folge: Größe der Hilfsinformationen im Index ist dann auch sehr groß.
- Alternative 2 und 3:
  - Dateneinträge viel kleiner als Datensätze. Somit besser als Alternative 1 bei großen Datensätzen und kleinen Suchschlüsseln (Anteil der Indexstruktur für Direktsuche wesentlich kleiner als in 1)
  - Wenn mehr als ein Index auf einer Datei benötigt wird, kann höchstens Index Variante 1 nutzen, die anderen 2 oder 3
  - Alternative 3 kompakter als 2, führt aber zu Dateneinträgen variabler Länge, selbst wenn Suchschlüssel feste Länge haben.

Index-Klassifikation

- *Primärschlüssel* vs. *Sekundärschlüssel*: Wenn der Suchschlüssel den Primärschlüssel enthält, dann heißt das Primärindex.
  - *Unique* Index: Suchschlüssel enthält einen Schlüsselkandidaten.
- *Geclustert* vs. *ungeclustert*:
- Geclustert: wenn die Ordnung der Datensätze die gleiche oder annähernd die gleiche ist wie die Ordnung der Dateneinträge im Index (mit anderen Worten: Index ist in der gleichen Form sortiert wie die interne Relation, auf die der Index verweist)
- Ungeclustert: Index ist anders organisiert als die interne Relation
  - Alternative 1 impliziert geclustert, aber nicht umgekehrt.
  - Eine Datei kann nur auf höchstens einem Suchschlüssel geclustert sein.
  - Retrieval-Kosten von Datensätzen über Indexe variieren sehr stark, abhängig, ob der Index geclustert ist oder nicht!

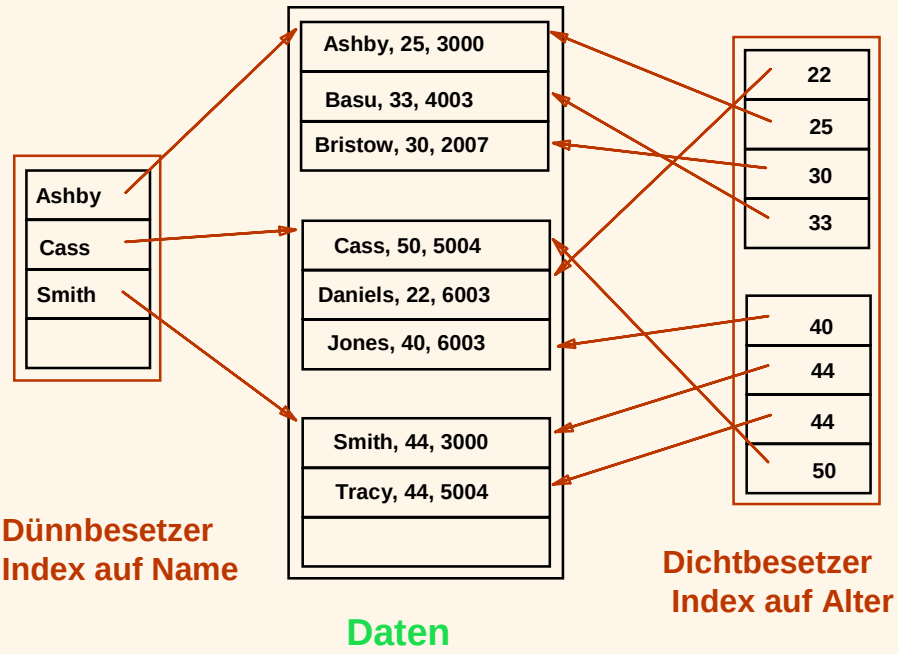
Geclustert vs. Ungeclustert

- Annahme: Alternative (2) wird für Dateneinträge genutzt, Datensätze sind in Heap-Datei gespeichert.
  - Anlegen eines geclusterten Index: erst Heap-Datei sortieren (mit etwas Freispeicher auf jeder Seite für zukünftige Inserts)
  - Überlaufseiten können für Inserts benötigt werden (deshalb ist die physische Ordnung der Datensätze nahe, aber nicht identisch mit der Sortierreihenfolge im Index)



Index-Klassifikation

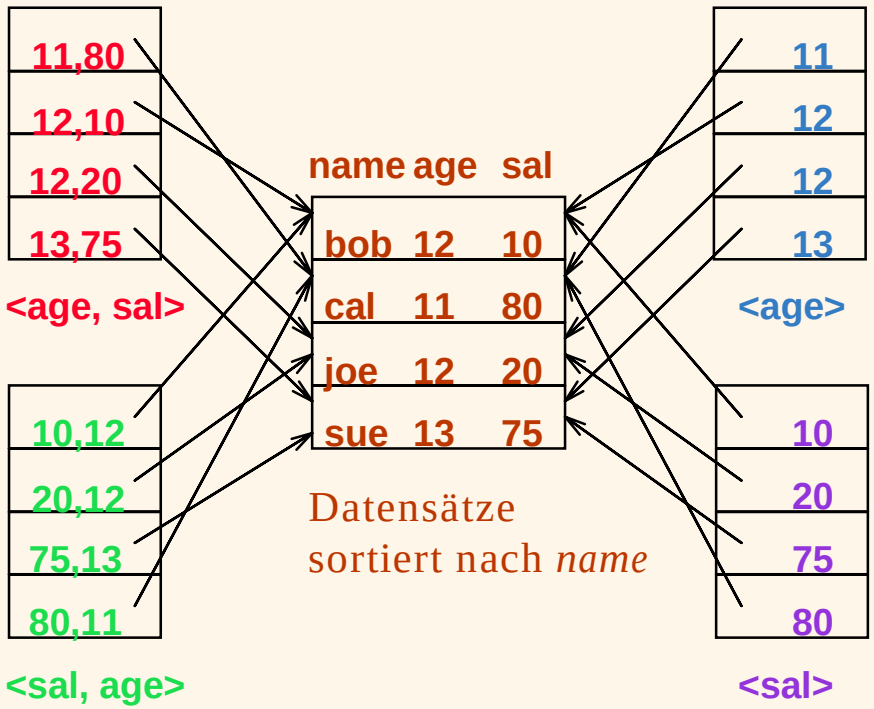
- *Dichtbesetzt (Dense)* vs. *Dünnbesetzt (Sparse)*:
  Dichtbesetzt, wenn es (wenigstens) einen Eintrag in der Indexdatei pro Zugriffsattributwert gibt (in irgendeinem Datensatz).
  - Alternative 1 führt immer zu dichtbesetztem Index.
  - Jeder dünnbesetzte Index ist geclustert.
  - Dünnbesetzte Indexe sind kleiner. Sinnvolle Optimierungen beruhen jedoch auf dichtbesetzten Indexen.



Index-Klassifikation (Forts.)

- *Zusammengesetzte Suchschlüssel (Mehr-Attribut-Index):* Suche auf einer Kombination von Feldern.
  - Gleichheitsanfrage (*Lookup*): Jeder Feldwert ist gleich einer Konstanten, z. B. im Index <sal,age>:
    - age=20 and sal =75
  - Bereichsanfrage: Feldwerte in einem bestimmten Bereich, z.B.
    - age=20 and sal > 10
- Einträge im Index sortiert nach Suchschlüssel, um Range Queries zu unterstützen.
  - Lexikographische Ordnung oder
  - Mehrdimensionaler Index.

Beispiele eines zusammengesetzten Schlüssels mit lexikographischer Ordnung



Zusammenfassung

- Es gibt viele Varianten einer Dateiorganisation: Jede ist in bestimmter Situation geeignet
- Wenn Lookup-Operationen häufig sind, sind sortierte Dateein oder ein Index wichtig
  - Hash-basierter Index nur gut für Lookup-Operationen
  - Sortierte Dateien und baum-basierter Index am besten für Bereichs-anfragen; auch gut für Lookups (Dateien bleiben praktisch nicht sortiert; B+ Baum daher besser)
- Index ist eine Menge von Dateneinträgen sowie ein Verfahren, um schnell Einträge mit einem vorgegebenen Schlüsselwert zu finden
- Index-Einträge können sein: vollständige Datensätze, <Schlüssel,Satz-ID> Paare oder <Schlüssel,Satz-ID-Liste>
- Indexe können klassifiziert werden: geclustert vs. ungeclustert, dichtbesetzt vs. dünnbesetzt (hat große Konsequenzen auf Nutzbarkeit / Performance)