

Dateiorganisationen und Indexen

Alternativen für Dateiorganisationen

Viele Alternativen existieren, *jede ist ideal in einer bestimmten Situation, und nicht so gut für eine andere:*

- Haufen [Heap] Dateien: Gut wenn die typische durchgeführte Operation ein Durchsuchen [scan] der gesamten Datei ist, wo alle Datensätze zugegriffen werden müssen.
- Sortierte Dateien: Gut wenn Datensätze in einer bestimmten Ordnung zugegriffen werden müssen, oder wenn nur ein "Bereich" der eingetragenen Werten gebraucht wird.
- Hash Dateien: Gut für Gleichheitsabfragen.
 - Die Datei enthält eine Menge Behältern [buckets]. Behälter = Hauptseite [primary page] plus null oder mehrere Überfluss-Seiten [overflow pages].
 - Hash-Funktion h: $h(r)$ = Behälter, in welchem der Datensatz r gehört. h benutzt nur eine Untermenge der Feldern von r , diese sind dann die Suchfelder [search fields] genannt.

Kostenmodell um Dateioorganisationen zu vergleichen

Wir untersuchen verschiedene Operationen auf den drei Dateitypen

- Scan (alle Datensätze verarbeiten)
- Gleichheitsabfrage
- Bereichsabfrage
- Datensatz einfügen
- Datensatz löschen

Für Einfachheit nehmen wir CPU-Kosten nicht in Betracht.

- **B**: Anzahl Datenseite
- **R**: Anzahl Datensätze per Datenseite
- **D**: (Mittlere) Zeit, um eine Seite zu lesen oder zu schreiben.
- Die Messung der Anzahl Seiten-I/Os vernachlässigt der Gewinn, der mit prefetching gewonnen werden konnte, d.h. sogar die I/O Kosten sind nur angenähert.
- Mittelwert-Analyse, basierend of einfache Annahmen.

Gut genug, um die allgemeinen Trends zu zeigen!

Annahmen im Kostenmodell

- Bei insert und delete werden einzelne Datensätze behandelt
- Heap Dateien:
 - Gleichheitsabfrage findet immer auf dem Schlüssel statt, es gibt genau einen zutreffenden Datensatz.
 - Neue Datensätze werden immer am Ende der Datei zugefügt.
- Sortierte Dateien:
 - Dateien werden nach dem Löschen „verdichtet“.
 - Selektionen werden auf den geordneten Felder(n) durchgeführt.
- Hash Dateien:
 - Kein Overflow-Behälter
 - Seiten sind 80% voll.

Kosten von Operationen auf die Dateien

	Heap Datei	Sortierte Datei	Hash- Datei
Alle Einträge zugreifen	BD	BD	1.25 BD
Gleichheitsabfrage	0.5 BD	$D \log_2 B$	D
Bereichsabfrage	BD	$D (\log_2 B + \# \text{ of pages with matches})$	1.25 BD
Einfügen	2D	Search + BD	2D
Löschen	Search + D	Search + BD	2D

☛ *Nur Annäherungen!*

Kosten von Operationen auf die Dateien

	Heap Datei	Sortierte Datei	Hash- Datei
Alle Einträge zugreifen	BD	BD	1.25 BD
Gleichheitsabfrage	0.5 BD	$D \log_2 B$	D
Bereichsabfrage	BD	$D (\log_2 B + \# \text{ of pages with matches})$	1.25 BD
Einfügen	2D	Search + BD	2D
Löschen	Search + D	Search + BD	2D

☛ *Nur Annäherungen!*

Auswertung der Resultate

- Heap-Dateien:
 - Effiziente Speicherung
 - Gut für Einfügen von Datensätze
 - Gut für Scans (gesamte Datei durchsuchen)
 - Schlecht für Suchen und Löschen
- Sortierte Dateien
 - Effiziente Speicherung
 - Schlecht für Einfügen und Löschen
 - Schneller als Heap-Dateien für Suche
 - (In reelle Welt, Dateien sind nie ganz sortiert)
- Hash- Dateien:
 - Brauch mehr Platz für Speicherung
 - Gut für Einfügen und Löschen
 - Sehr gut für Gleichheitssuche
 - Sehr schlecht für Bereichssuche

Indexen

- Ein *Index* auf einer Datei beschleunigt bedeutend die Selektion von Datensätze auf den *Suchschlüssel-Felder [search key fields]* des Indexes.
 - Jede Untermenge von Felder einer Relation (d.h. Tabelle) kann als Suchschlüssel für ein Index auf dieser Relation benutzt werden.
 - *Suchschlüssel* hat *nichts* zu tun mit dem Schlüssel (kleinste Menge Felder, die eindeutig einen Datensatz in einer Relation definiert)
- Ein Index enthält eine Menge *Datenverweise [data entries]*, und unterstützt effiziente Zugriffe auf allen Datenverweise k^* mit einem gegebenen Schlüssel k .

Varianten für Datenverweise k^* im Index

- Drei Varianten:
 - Variante 1: Den Datensatz selbst, mit Suchschlüsselwert k
 - Variante 2: $\langle k, \text{rid vom Datensatz mit Suchschlüsselwert } k \rangle$
 - Variante 3: $\langle k, \text{Liste von rids von Datensätze mit Suchschlüssel } k \rangle$
- Die Wahl von Varianten für Datenverweise ist orthogonal zu den Indizierungsmethoden, die angewendet werden, um einen Datenverweis mit Schlüssel k zu lokalisieren.
 - Beispiele vom Indizierungsmethoden: B+ Bäume, Hash-basierte Strukturen.
 - Normalerweise enthält ein Index zusätzliche Information, die die Suche nach den erwünschten Datensätze richtet.

Varianten für Datensätze (Fortsetzung)

- Variante 1:
 - Wenn diese Variante benutzt wird, ist die Index-Struktur eine Dateiorganisation für Datensätze (wie Heap Dateien oder sortierte Dateien)
 - Höchstens ein Index auf einer bestimmten Sammlung von Datensätze kann diese Variante benutzen. Sonst wären die Datensätze dupliziert, was zu redundante Speicherung führen würde, und potentielle Inkonsistenzen einbringen.
 - Wenn die Datensätze sehr gross sind, sind die Anzahl Seiten mit Datensätze auch sehr gross (Seitengrösse ist fest). Das bedeutet, dass die zusätzliche Information im Index auch gross ist.

Varianten für Datensätze (Fortsetzung)

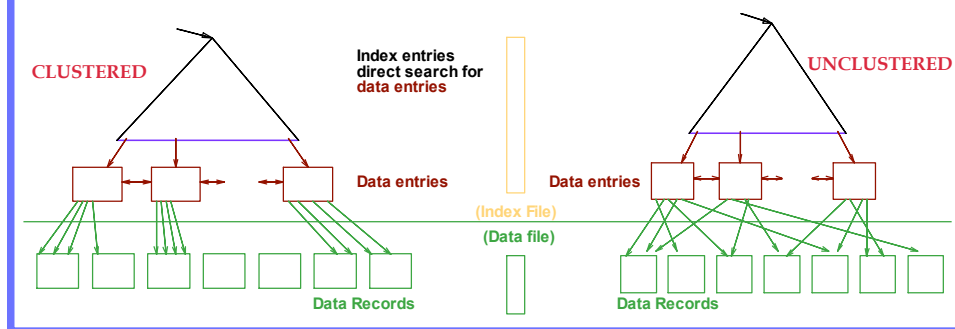
- **Varianten 2 and 3:**
 - Datenverweise sind normalerweise viel kleiner als Datensätze. Deshalb ist es im allgemeinen besser als Alternative 1 für grosse Datensätze, v.a. wenn der Suchschlüssel klein sind. (Anteil Index-Struktur, die benutzt wird, um die Suche zu orientieren, ist viel kleiner als mit Alternative 1.)
 - Wenn mehr als ein Index für eine bestimmte Datei angefordert ist, kann höchstens ein Index Alternative 1 benutzen, alle anderen müssen Alternative 2 und 3 benutzen.
 - Alternative 3 ist mehr kompakt als Alternative 2, aber benötigt Datenverweise mit variablen Länge, auch wenn die Suchschlüssel eine fixe Länge haben.

Klassifikation von Indexen

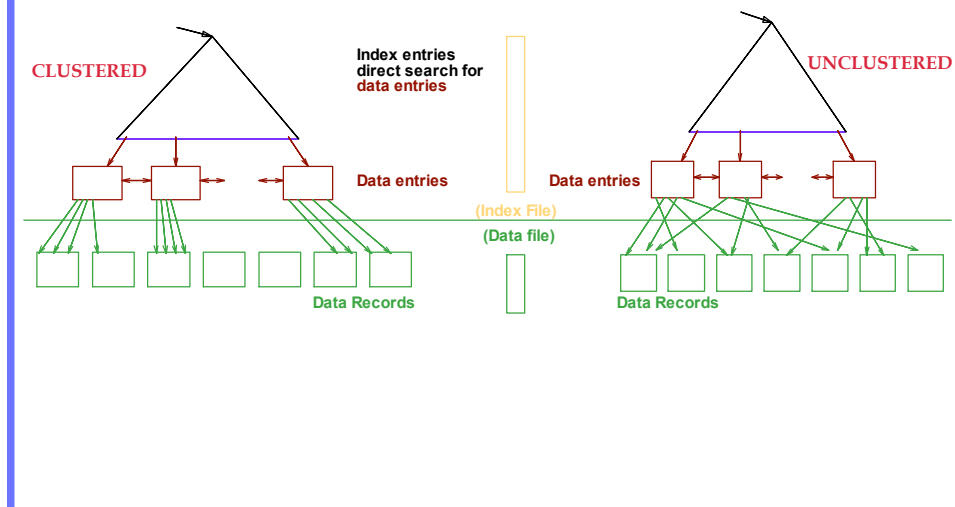
- **Primär oder sekundär:** Wenn der Suchschlüssel der Hauptschlüssel enthält, dann ist er der Index *primär*.
 - *Einzigartiges [unique] Index:* Suchschlüssel enthält ein Schlüsselkandidat.
- **Gesammelt [clustered] oder zerstreut [unclustered]:** Falls die Ordnung von Datensätze die gleiche oder „nah“ ist von der Ordnung der Datenverweise, dann ist der Index gesammelt oder clustered.
 - Variante 1 impliziert einen clustered Index, aber nicht umgekehrt.
 - Eine Datei kann höchstens mit einem Suchschlüssel clustered sein
 - Die Kosten, Datensätze mit einem Index zu erhalten, variiert sehr damit, ob der Index clustered ist oder nicht!

Clustered und unclustered Indexen

- Nehmen wir an, Variante (2) ist für Datenverweise gebraucht, und Datensätze sind in einer Heap-Datei gespeichert
 - Um den clustered Index zu bauen, muss die Datei sortiert werden, (mit einigen freien Plätze für zukünftige Datensätze auf jeder Seite)
 - Überfluss-Seiten können fürs Einfügen gebraucht, deshalb sind die Datensätze „nah“, aber nicht identisch mit der Ordnung im Index.

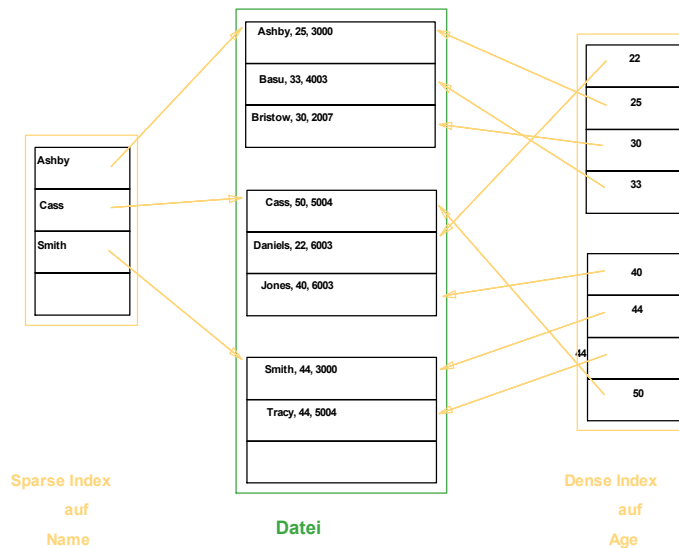
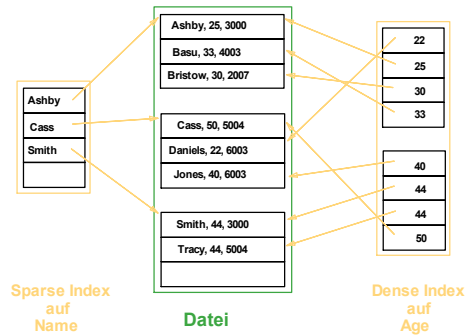


Clustered und unclustered Indexen



Klassifikation von Indexen (Forts.)

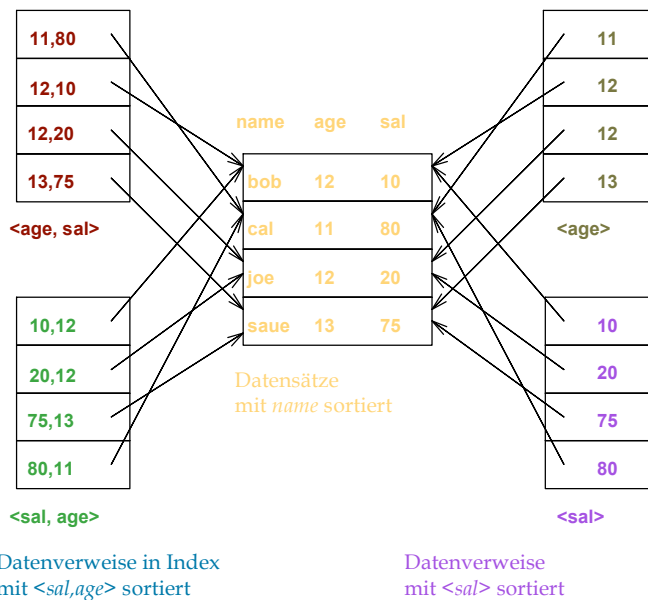
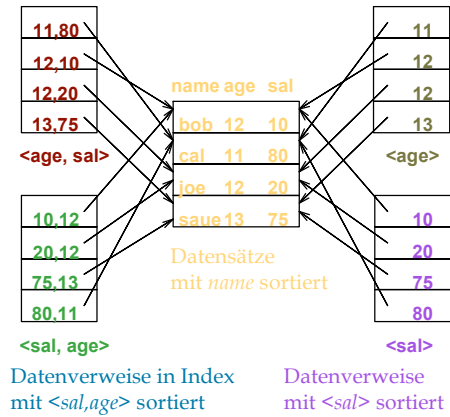
- Dicht [*dense*] oder spärlich [*sparse*]: Falls es mindestens ein Datensatz per Suchschlüssel gibt, dann ist der Index dicht.
 - Variante 1 bedeutet immer einen dichten Index.
 - Jeder spärlicher Index ist clustered.
 - Spärliche Indexen sind kleiner, aber Optimierungen benötigen manchmal einen dichten Index



Klassifikation von Indexten (Fortsetzung)

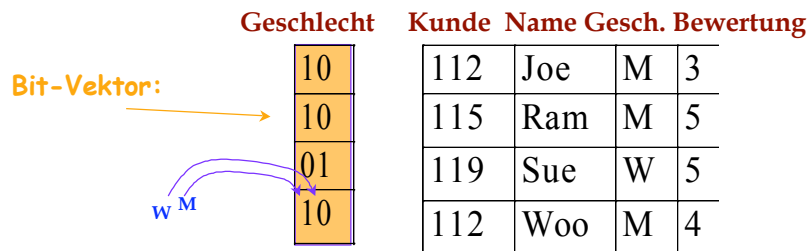
- *Zusammengesetzte Suchschlüssel* [composite search keys]: Suche ist auf eine Kombination von Felder.
 - Gleichheitsabfrage: Jeder Wert eines Feldes ist gleich einer konstanten Wert: z.B. <sal,age> Index:
 - age=20 und sal =75
 - Bereichsabfrage: bestimmte Felder sind nicht konstant: z.B.
 - age =20; oder age=20 und sal > 10
- Die Datenverweise im Index sind mit dem Suchschlüssel sortiert, um Bereichsabfrage zu unterstützen.
 - Lexikographisch, oder
 - Räumlich

Beispiele von Indexten mit zusammengesetzten Schlüssel, lexikographische Reihenfolge.



Bitmap Index

1 Bit für jeden möglichen Wert



*Viele Abfragen
können mit
Operationen auf
bit-Vektoren
direkt
beantwortet werden!*

Zusammenfassung

- Es gibt viele Varianten für Dateioorganisationen, jede ist für eine bestimmte Situation angemessen
- Falls Abfragen auf einer bestimmten Schlüssel häufig sind, muss die Datei nach dieser Schlüssel sortiert werden, oder ein Index auf dieser Schlüssel muss generiert werden
 - Hash-basierte Indexen sind gut für Gleichheitsabfragen
 - Sortierte Dateien und baumbasierte Indexen sind das beste für Bereichsabfragen, auch gut für Gleichheitsabfragen. (In der Praxis sind deshalb Dateien selten sortiert. B+-Baum ist meistens benutzt.)
- Ein Index ist eine Sammlung von Datenverweise mit einem Weg, um schnell Datensätze mit einem bestimmten Suchschlüssel zu finden.

Zusammenfassung (Fortsetzung)

- Datenverweise können entweder die Datensätze selbst sein, oder Paare <Schlüssel, rid>, oder Listen <Schlüssel, rid-list>.
 - Die Wahl der Datenverweise ist orthogonal zu der Wahl der Indizierungstechnik, die gebraucht wird, um Datensätze mit einer bestimmten Suchschlüssel zu finden.
- Man kann mehrere Indexten auf einem bestimmte Datei haben, jeder mit einem anderen Suchschlüssel.
- Indexten können als clustered oder unclustered, primär oder sekundär, dicht oder spärlich klassifiziert werden. Unterschiede haben wichtige Folgen für den Gebrauch / Effizienz.