Datenbanksysteme

Wintersemester 2016/17

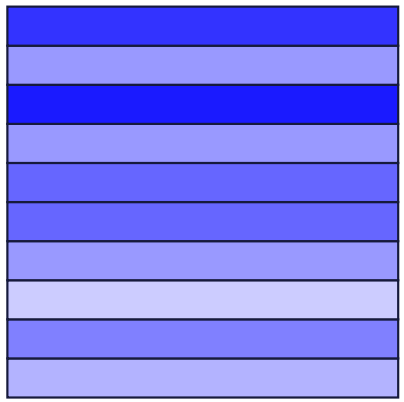
Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Organisation von Daten-Dateien

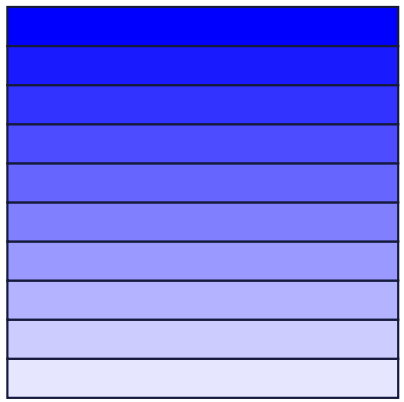
Haufen

- Einfachster Weg eine Datei zu organisieren ist neu ankommende Sätze in die Seite (den Block) am Ende der Datei einzufügen.
- Einfügen ist sehr effizient.
- Suche ist teuer.
- U.U. in Zusammenspiel mit einem (sekundären) Index



Sortierte Dateien

- Gegeben ein Schlüssel anhand dessen Sätze geordnet werden können
- Effizientere Suche (binär) - Obwohl so nicht realisiert (siehe Primär-Indexe, z.B. B+ Baum)



Beispiel: Sortierte Datei (Hier nach Name)

	Name	MatrNr	Semester
Block 1	Aaron	443421	10
	Adam	233499	1
	...		
	Acosta	561921	1
Block 2	Allen	581722	9
	Anderson	339163	8
	...		
	Archer	965492	1
Block 3	Arnold	672961	3
	Arnold	759311	1
	...		
	Atkins	173522	8
	⋮		
Block n	Wright	672961	4
	Wyatt	197646	7
	...		
	Zimmer	524145	12

Suche in Dateien

"If you don't find it in the index,
look very carefully through the entire catalog"
— Sears, Roebuck, and Co., Consumers' Guide, 1897

(aus dem Buch von Ramakrishnan & Gehrke)

Lineare Suche

Binäre Suche in sortierten Dateien

Suche via Indexen

Möglichkeiten was ein Index enthalten kann

Drei Möglichkeiten:

Möglichkeit 1: Den eigentlichen Datensatz

- Eintrag ist ein tatsächlicher Datensatz (kein Verweis auf einen) für Suchschlüssel k

Möglichkeit 2: Einen Verweis auf den Datensatz

- Eintrag hat die Form <k, rid>, wobei rid die ID des Datensatzes mit Suchschlüssel k ist.

Möglichkeit 3: Eine Liste von Verweisen auf Datensätze

- Eintrag hat die Form <k, rid-list>, wobei rid-list eine Liste ist, die IDs der Datensätze mit Suchschlüssel k enthält.

Klassifizierung von Indexen

Primäre, Sekundäre und Clustering Indexe

- **Jede Datei kann nur in einer Art und Weise sortiert sein.**
- **Primär-Index**: Index über gegebener Ordnung der Datei - via Primärschlüssel. Ein Index-Eintrag pro Block.
- **Clustering-Index**: Ordnung wie in Datei, aber keine unique Werte pro Attribut nach dem sortiert wird. Ein Index-Eintrag pro distinct Wert.
- **Sekundär-Index**: Index über anderen Attributen (nicht wie Ordnung der Datei). Es kann mehrere dieser sek. Indexe geben.

Single-Level und Multi-Level Indexe

- **Single-Level**: Einfacher Index, Verweise auf Sätze oder Blöcke
- **Multi-Level**: Verweise auch auf andere Index-Einträge (siehe B+ Baum)

Clustering- vs. Non-Clustering

Eine Relation kann nur einen clustered Index haben. Es muss nicht immer der Index auf dem Primärschlüssel sein.

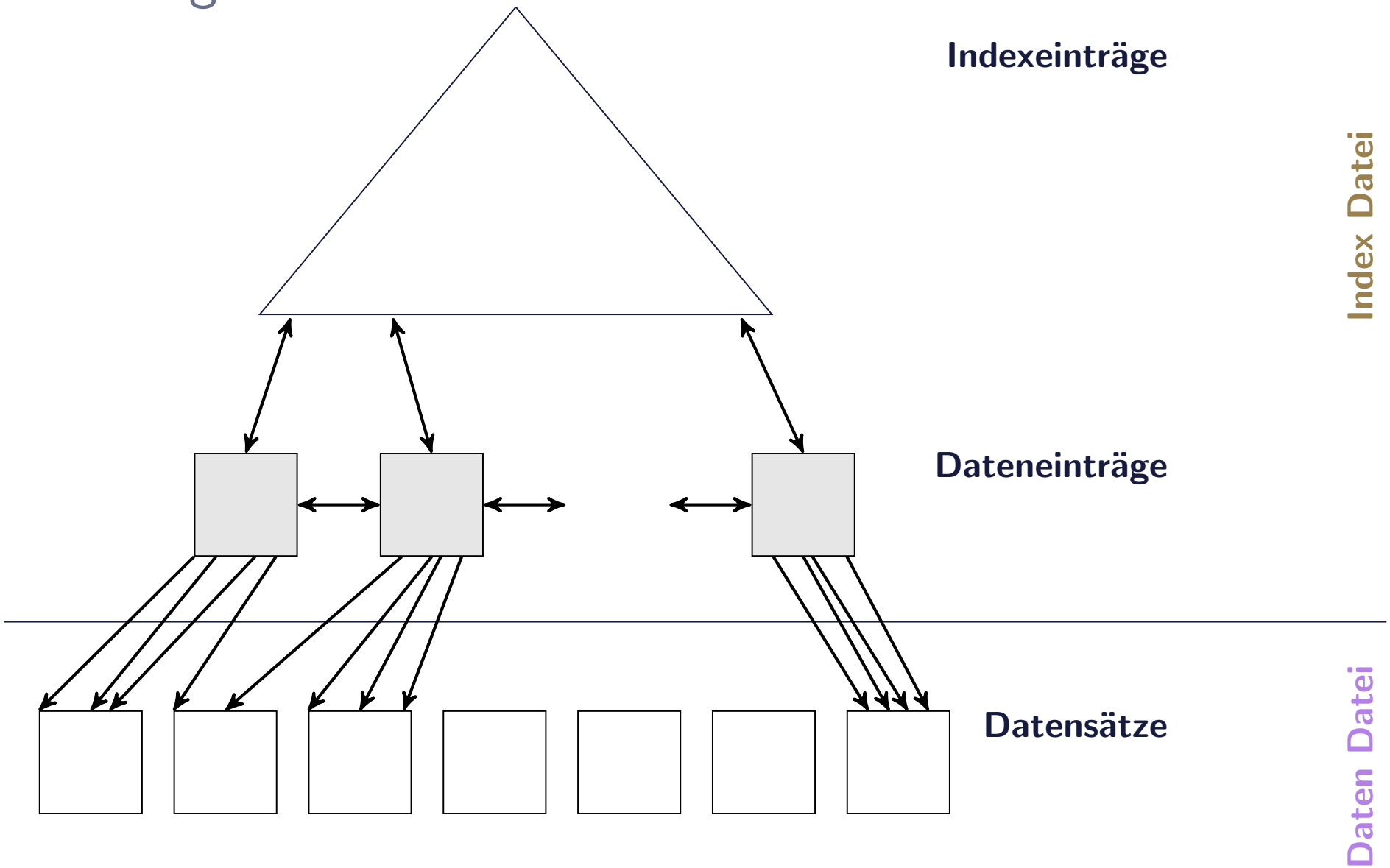
In Postgresql

```
create index myindex on something(mycolumnname);
cluster something using myindex;
```

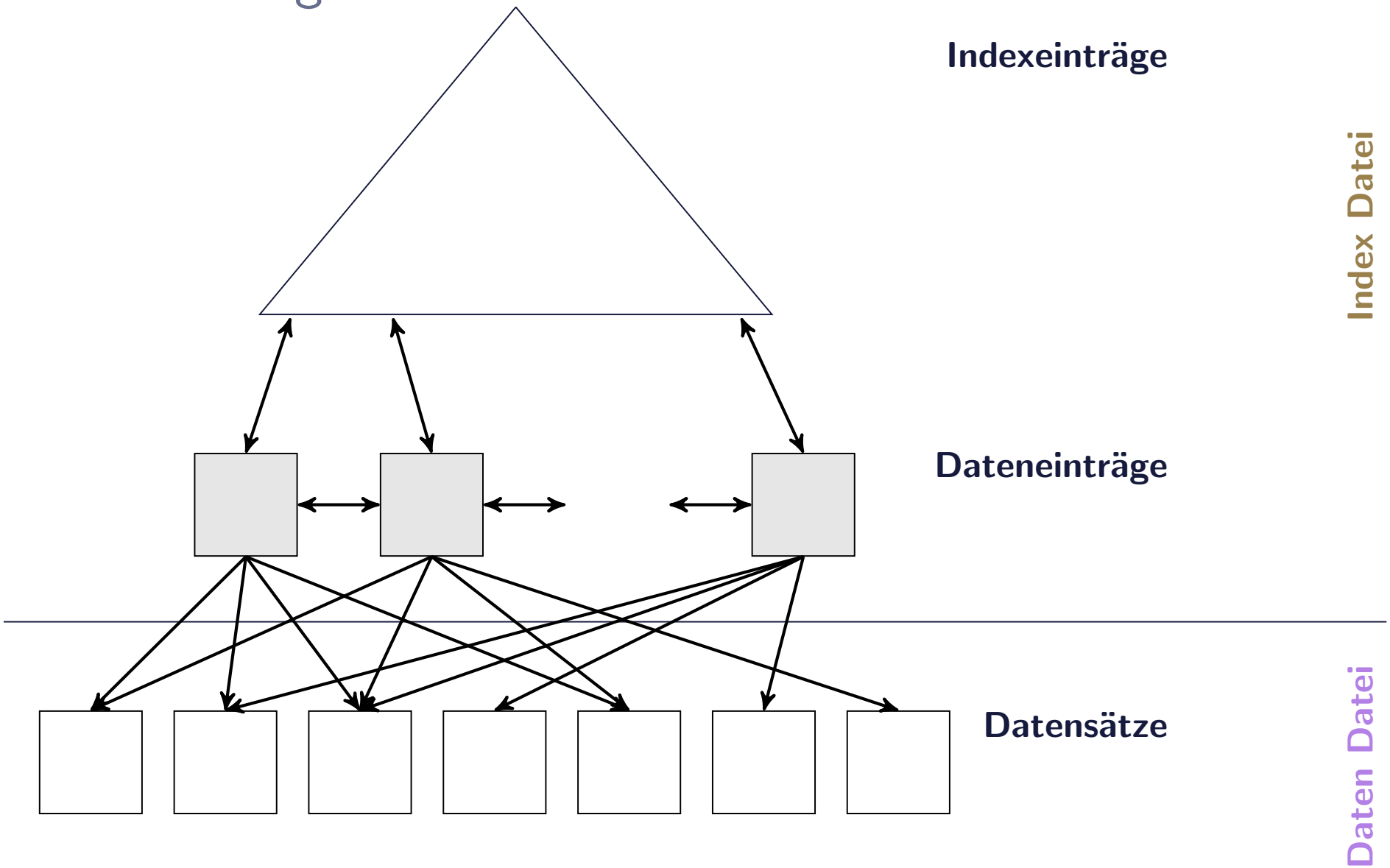
Non-Clustering-Indexe

Es kann natürlich mehrere Non-Clustering-Indexe geben.

Clustering-Index



Non-clustering-Index



Index-Only Queries

Was passiert wenn wir einen **non-clustered B+ Index auf Attribut X** haben und die folgende Anfrage ausführen?

```
SELECT X,Y
FROM meineTabelle
WHERE X<1000
```

- Um an den Wert des Attributs Y zu gelangen muss im Datensatz des sich qualifizierten Tupels nachgeschaut lesen.
- Diese **Indirektion** verursacht Kosten.
- Wenn wir in den Index noch Attribut Y hinzunehmen, also als Composite key (X,Y) angeben, **kann die Anfrage nur mithilfe des Indexes beantwortet werden.** Eine sogenannte Index-Only Query.

Sparse vs. Dense

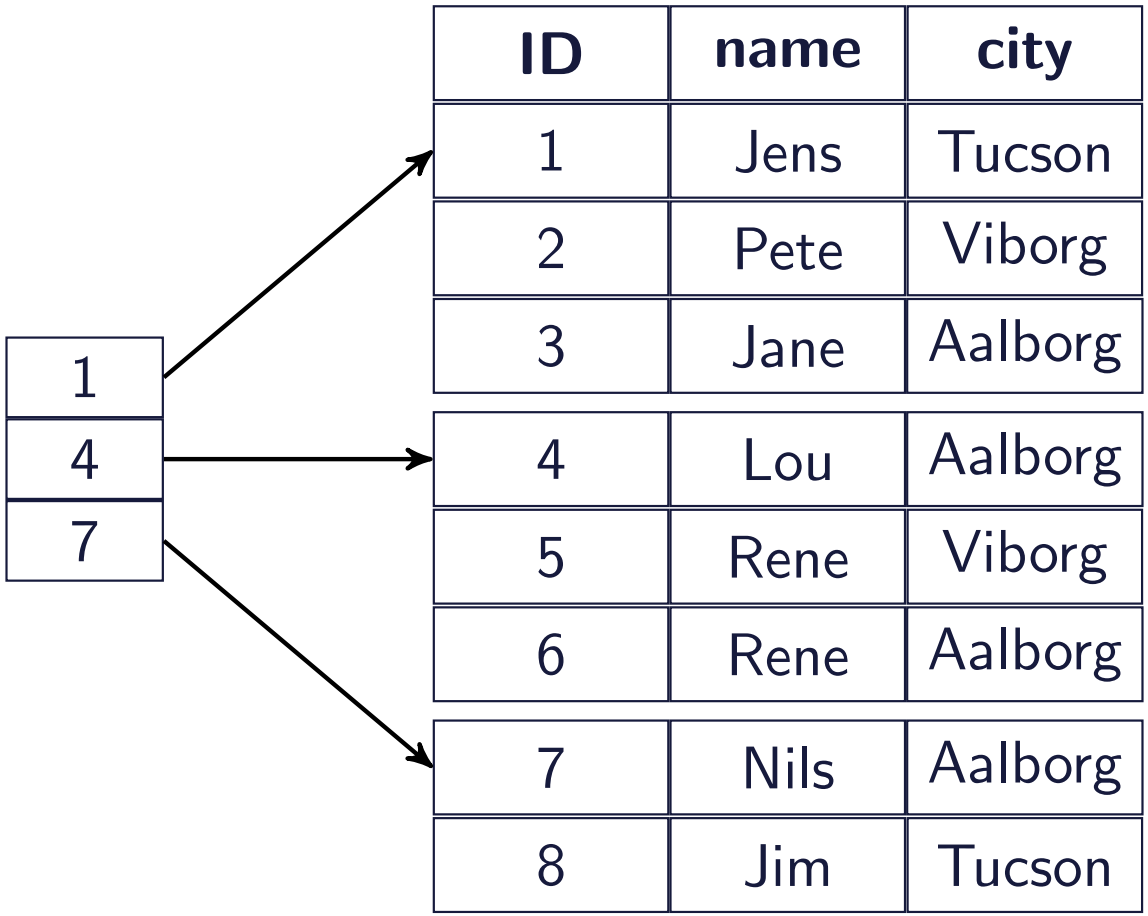
- **Dense(=Dicht)**: Index hat einen Eintrag für jeden (Daten)satz
- **Sparse (=Dünnbesetzt)**: Indexeinträge nur für einen Teil der Suchschlüssel. Auf Anfang Block oder ersten Datensatz mit diesem Schlüssel. Setzt Sortierung nach Schlüssel voraus. Primärindex ist z.B. sparse.

Beispiel

- Gegeben ein sekundärer Index auf einem Attribut, das für jeden Satz einen eigenen (distinct) Wert hat.
- Ist dieser Index sparse oder dense?

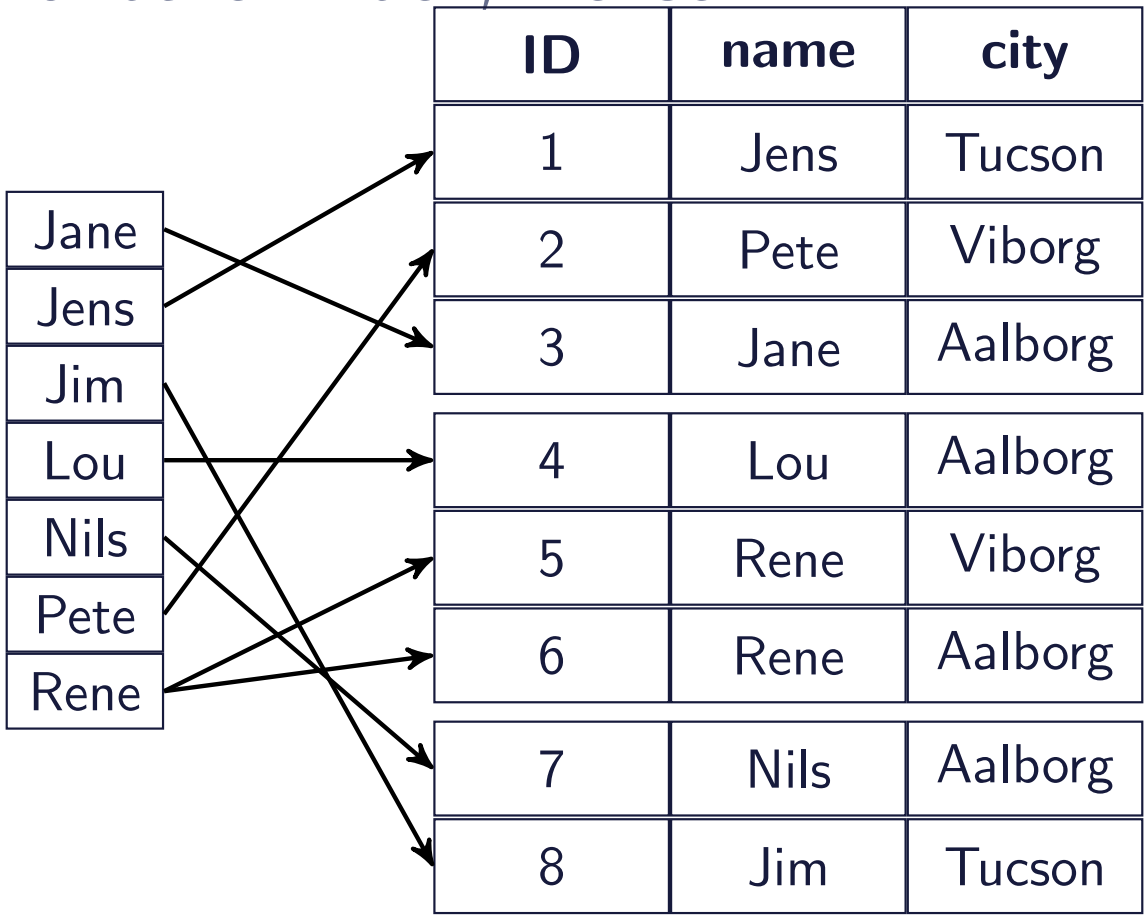
Literatur: Das Buch von Elmasri & Navathe: "Fundamentals of Database Systems" (Addison Wesley) ist hier sehr ausführlich!
Generell Vorsicht bei nicht einheitlicher Verwendung der Bezeichnungen Primärindex oder Sekundärindex in Literatur.

Beispiel: Primärer Index, Sparse



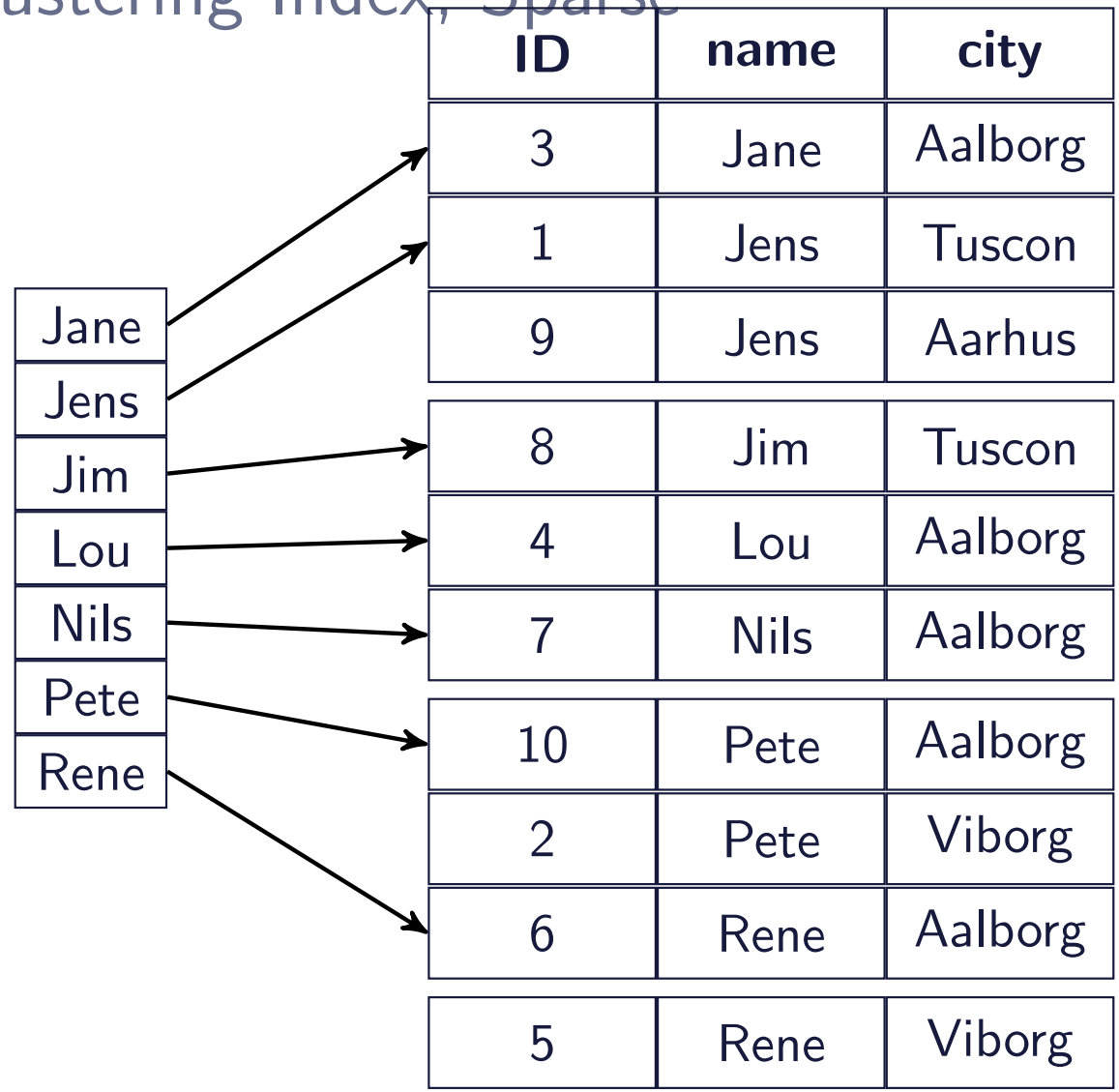
- Angelegt über einer Datei, die nach dem Such-Schlüssel sortiert ist
- Ein Index-Eintrag für jeden Block

Beispiel: Sekundärer Index, Dense



- Angelegt über einer Datei, die nicht nach dem Such-Schlüssel geordnet ist
- Es gibt Duplikate in den Such-Schlüsseln
- Ein Index-Eintrag pro Tupel

Beispiel: Clustering-Index, Sparse

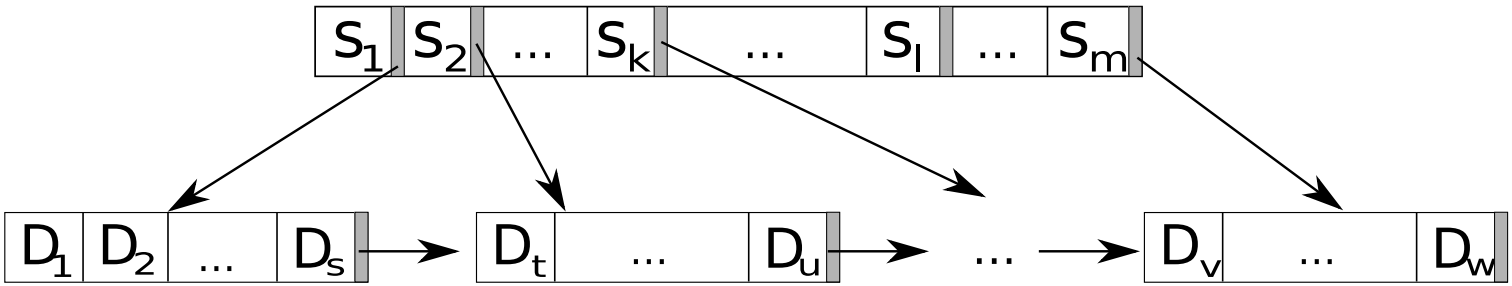


- Angelegt über einer Datei, die nach dem Such-Schlüssel geordnet ist
- Ein Index-Eintrag pro distinct Wert auf ersten Datensatz mit Wert.

ISAM

Index-Sequential Access Method (ISAM). Statisch.

Besteht aus Schlüsseln S_j und Datensätzen D_i.



- Sowohl Schlüssel als auch Daten werden geordnet abgespeichert.
- **Suche:**
  - Binäre Suche in Schlüsseln zur gewünschten Position
  - Sequentielles Lesen in Datenseiten
- **Einfügen:**
  - Auffinden der Einfügeposition (wie bei Suche)
  - Was passiert wenn die Seite, in die eingefügt werden soll, voll ist? Nicht gut ...
- **Löschen:**
  - Auffinden der Löschposition (wie bei Suche und Einfügen)
  - Löschen des Datensatzes. Was passiert wenn die Seite leer wird?

Bäume

Beobachtung

- Binäre Bäume nicht optimal für Festplatten
- Wichtig: Anpassung der Kapazität der Knoten an Größe der Seiten.
- Warum?

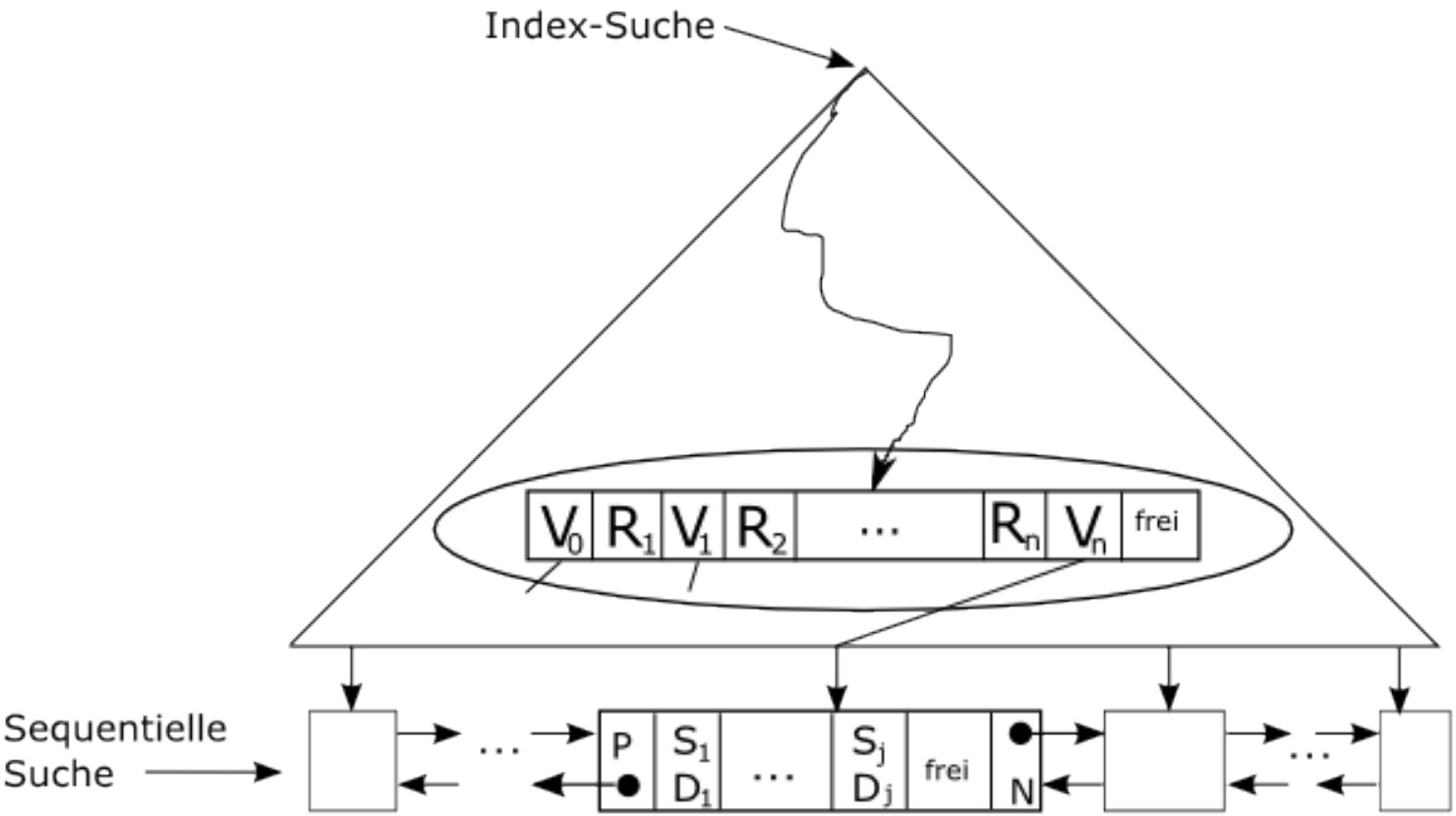
Fanout

- Je breiter der Baum desto weniger tief.
- Je flacher desto weniger "Sprünge" zwischen Knoten

$\Rightarrow$ Weniger Zugriffe auf Seiten auf der Festplatte.

B+ Baum (Bekannt aus VL Informationssysteme)

- "Hohler" Baum: Daten nur in den Blättern
- Suche muss also immer bis zu den Blättern laufen
- Aufbau: Referenzschlüssel R_j, Schlüssel S_k, Daten D_i, Zeiger V_m
- Blattknoten sequentiell verbunden!

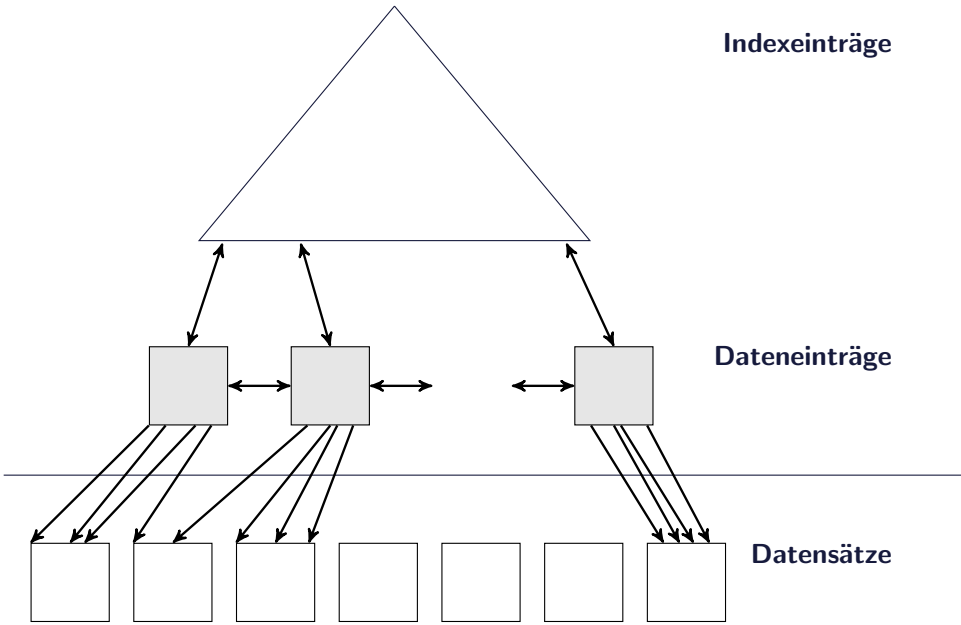


B+ Baum Übersicht

- Jeder Knoten eines B+ Baums hat die Größe eines Blocks (Seite).
- Jeder Knoten ist mindestens 50% gefüllt
- Ein B+ Baum hat eine relativ geringe Anzahl von Stufen (Levels)
- Die ersten Ebenen (ein oder zwei) des Baumes werden im Hauptspeicher gehalten!
- "Logisch" nahe Knoten im Baum bedeutet nicht unbedingt auch physisch nahe. D.h. Zugriff auf einen Knoten kostet einen wahlfreien Zugriff.
- Erwarteter Füllgrad 69%
- Die inneren Knoten sind eine Hierarchie von sparse Indexen.

Zugriff auf Sortierte Daten: **Clustering**-Index

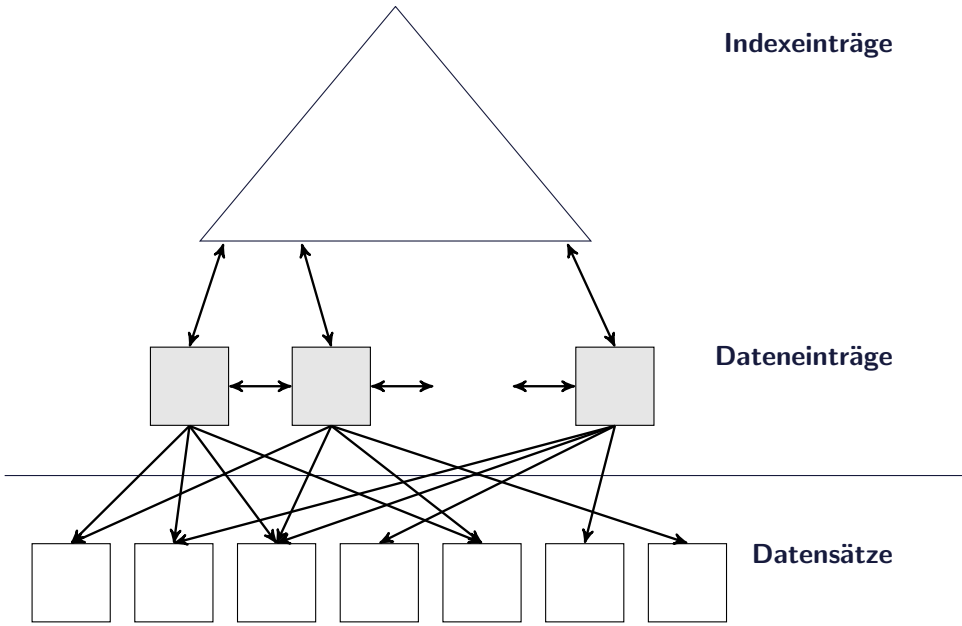
- Ist der B+ Baum clustered bzw. primär, dann ist die Traversierung der Datensätze in sortierter Reihenfolge sehr effizient.
- Kosten: Suche am weitesten links stehenden Dateneintrag und lese dazugehörige Datenseiten.
- **Keine Datenseite wird mehrfach gelesen.**



- **Noch besser: Siehe Möglichkeit 1, d.h. Datensätze befinden sich direkt in Blättern des Indexes!** Anmerkung: B+ Baum Speicherplatzausnutzung von 69%, wobei im Prinzip bei einfacher sortierter Datei 100% Ausnutzung möglich sind.

Zugriff auf Sortierte Daten: **Non-Clustering**-Index

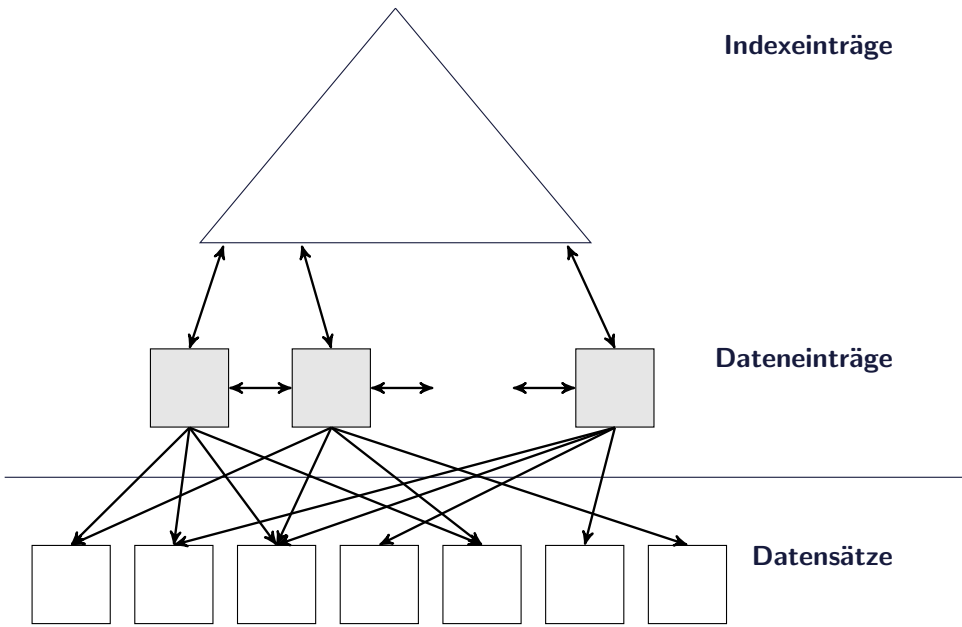
- Dateneinträge haben Form $\langle k, rid \rangle$
- **Worst Case: Jeder** rid **Eintrag verweist auf eine andere Datenseite.**
- Kosten: Für jeden Datensatz muss ein Block gelesen werden, plus Lesen der Indexeinträge.



In Praxis sind Kosten niedriger, da z.B. einige Seiten im Datenbankpuffer gefunden werden können. **Nutzen von Unclustered-Index für sortierten Zugriff auf Daten hängt von Grad der Übereinstimmung von Ordnung der Dateneinträge und physischer Ordnung der Datensätze ab.**

Zugriff auf Sortierte Daten: **Non-Clustering**-Index (2)

- Annahme: N Datenseiten und p Datensätze pro Datenseite, also $p \times N$ Datensätze.
- Weiterhin sei f das Verhältnis von Dateneintrag zu Datensatz, dann haben wir $f \times N$ Blattknoten.
- Kosten die Datensätze in sortierter Reihenfolge zu lesen ist also $(f + p) \times N \approx p \times N$ für $f = 0.1$ oder kleiner und p typischerweise größer als 10.



Zugriff auf Sortierte Daten: Vergleich Non-Clustering-Index zu Sortieren

Folgende Tabelle vergleicht Kosten von sortierem Zugriff via Unclustered-Index und Kosten durch externes Sortieren ($B = 1000$, $b = 32$, Details dazu später).

N	**Sortieren**	$\mathbf{p = 1}$	$\mathbf{p = 10}$	$\mathbf{p = 100}$
100	200	100	1000	10.000
1000	2000	1000	10.000	100.000
10.000	400.00	10.000	100.000	1.000.000
100.000	600.000	100.000	1.000.000	10000000
1.000.000	8.000.000	1.000.000	10.000.000	100.000.000
10.000.000	80.000.000	10.000.000	100.000.000	1.000.000.000

Zum Vergleich:

Für Clustering-Index sind die Kosten ungefähr gleich N.

Zugriff auf Sortierte Daten: Vergleich Non-Clustering-Index zu Sortieren (2)

Auch für den Fall, dass gar nicht alle Datensätze gelesen werden sollen, z.B. bei Bereichsanfrage nach allen Professoren mit Personalnummer kleiner 31415, kann es günstiger sein die komplette Relation (=Datei) zuerst zu sortieren und dann Tupel in Datenseiten sequentiell lesen als den Non-Clustering-Index zu benutzen.

```
SELECT *
FROM Professoren
WHERE PersNr<31415
```

Statischer Hash-Index

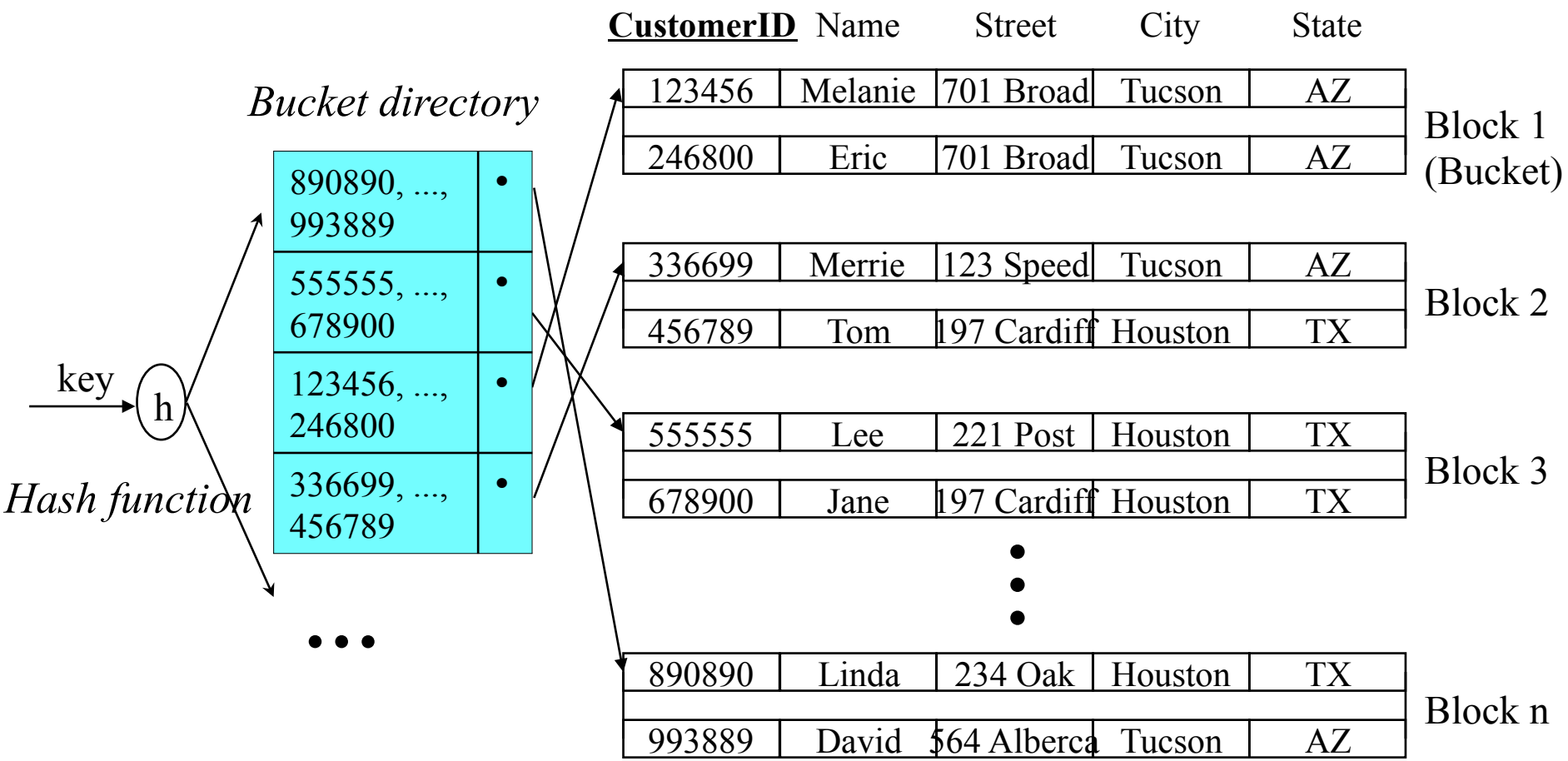
Idee

Erzeuge Index basierend auf Gruppierung von Tupel anhand Hash-Funktion.

Vorgehensweise

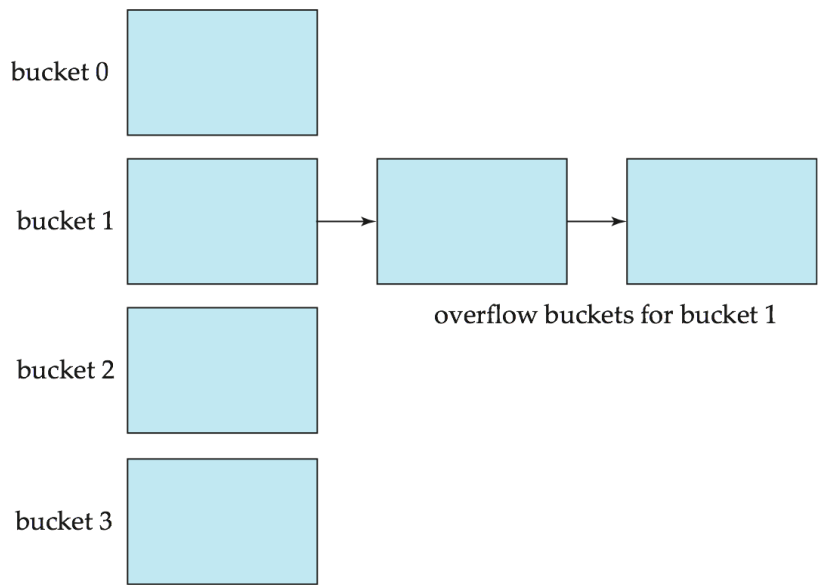
- Suche geeignete **Hash-Function** h
- Wende Hash-Funktion h an auf Wert k des Such-Schlüssel eines Tupels $\rightarrow h(k)$
- Erzeuge ein **Bucket** (Eimer) für jeden Wert von $h(k)$
- Hier, ein Block für jeden Bucket
- Einfaches Beispiel: $h(x) := x$ **mod** 5. Bildet z.B. Matrikelnummern ab auf Buckets 0 bis 4.

Statischer Hash-Index: Beispiel



Statischer Hash Index

- Suche (Lookup)
  - Ein Zugriff auf auf das Verzeichnis (directory)
  - Ein Zugriff auf die eigentliche Datei
- Performance hängt von Wahl der Hash-Funktion ab
- Überlauf von Buckets
  - Zu viele verschiedene Schlüssel-Werte werden auf gleichen Bucket abgebildet
  - Lösung: **Überlaufbehandlung** mittels Verkettung von Überlauf-Buckets



Statischer Hash Index: Probleme

Hash-Funktion und Anzahl der Buckets ist gegeben bei Initialisierung

Aber Daten verändern sich im Laufe der Zeit:

- Initiale Anzahl Buckets wird zu klein
  $\rightarrow$ viele Überlaufe, schlechte Performance
- Initiale Anzahl Buckets zu groß gewählt
  $\rightarrow$ Verschwendung von Speicherplatz (leere Blöcke)

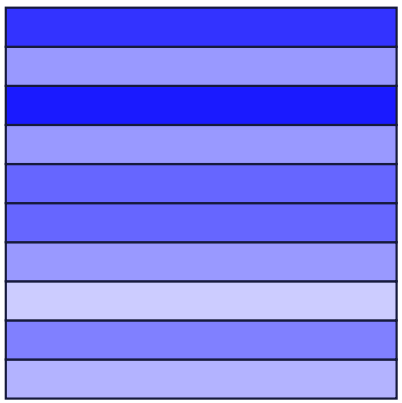
Lösung:

- Periodische Neuorganisation
- Dynamische Hashverfahren:
  erlauben dynamische Anpassung der Anzahl der Buckets

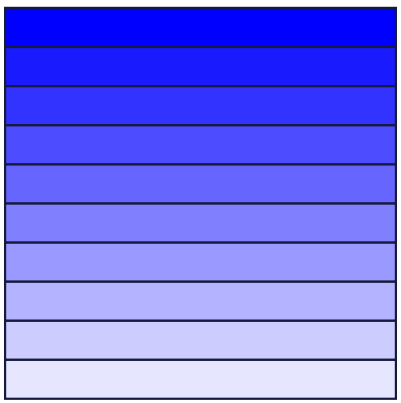
Wir kommen im Kapitel über Indexstrukturen ausführlich darauf zurück.

Überblick: Anordnung von Tupeln in Dateien

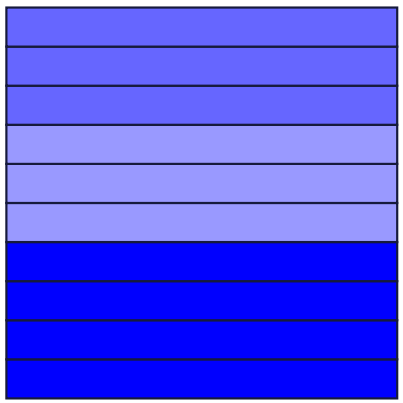
Haufen



Sequenziell (Geordnet)



Hashing



Indexe in Postgres: B+ Baum

```
create table MeineTabelle (
        ID int,
        major int,
        minor int,
        name varchar
);
```

B+ Baum mit Schlüssel ID

```
create index index1 on MeineTabelle ID;
```

B+ Baum mit Schlüssel (major, minor)

```
create index index2 on MeineTabelle (major, minor);
```

Indexe über mehrere Spalten

Sogenannte **Composite-Key Indexe**.
Angabe einer Reihe (**Achtung! Reihenfolge ist wichtig!**) von Spalten.

create index indexName **on** MeineTabelle (att1 , att2, att3);

Tupel werden dann im Index sortiert anhand dieser Attribute, "Lexikographische Ordnung": att1 ist primäres Kriterium, gefolgt von att2, etc.

Optional mit Ordnung ASC oder DESC der einzelnen Attribute. Default ist ASC. Zum Beispiel:

create index indexName **on** MeineTabelle (att1 **DESC** , att2, att3);

Indexe über mehrere Spalten (2)

```
SELECT   E.eid
FROM     Employees E
WHERE    E.age BETWEEN 20 AND 30
         AND E.salary BETWEEN 3000 AND 5000
```

- Index auf (age, salary) würde helfen.
- Wenn beide Kriterien gleich selektiv sind spielt es keine Rolle ob (age, salary) oder (salary,age)
- Im Allgemeinen spielt Reihenfolge aber eine Rolle.

Indexe über mehrere Spalten (3)

```
SELECT  E.eid
FROM    Employees E
WHERE   E.age=25
        AND E.salary BETWEEN 3000 AND 5000
```

- Ein clustered B+ Index auf (age, salary) wird gut funktionieren, da Datensätze nach age sortiert sind (und falls mehrere den gleichen Wert für alt haben, wird nach salary sortiert). D.h. alle Datensätze mit age=25 sind zusammen abgelegt.
- Ist jedoch der Index definiert auf (salary, age), ist dies nicht mehr gegeben.
- Würde die Anfrage nur eine Bereichsanfrage über salary enthalten, so wäre dieser (salary, age) Index OK.

Indexe über mehrere Spalten (4)

```
SELECT   AVG(E.salary)
FROM     Employees E
WHERE    E.age=25
         AND E.salary BETWEEN 3000 AND 5000
```

- Mit einem Index auf (age, salary) können wir die Anfrage direkt aus dem Index beantworten. "Index only"
- Dies ist auch mit einem Index auf (salary, age) möglich, bloß müssen dann voraussichtlich mehr Indexeinträge gelesen werden.

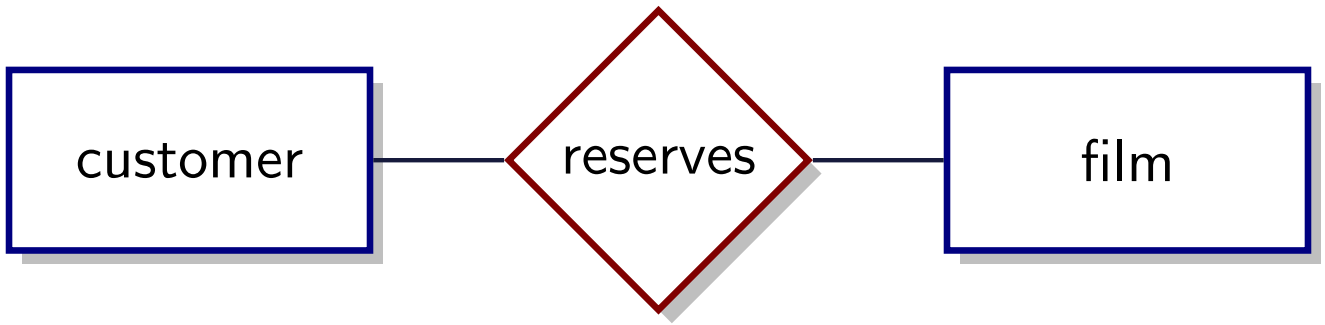
Indexe in Postgres: Hash Index

create index index3 **on** MeineTabelle **using hash** (column);

Eignung von Hash Indexen allgemein

- Verwendung bei Punktanfragen (d.h. vom Typ A=x)
- Hash Index über mehrere Attribute (Spalten) werden von Postgresql nicht unterstützt
- Nicht geeignet für Anfragen mit Operatoren $<$, $\leq$, $\geq$, $>$
- ebenso wie für **Bereichsanfragen**, z.B. $20 <$ Alter < 50

Beispiel Datenbank



- **customer** (customerID, name, street, city, state)
- **reserved** (customerID, filmID, resDate)
- **film** (filmID, title, kind, rentalPrice)

Verschiedene Selektionen

- Primärschlüssel, Punktanfrage

$$\sigma_{filmID=2}(film)$$

- Punktanfrage

$$\sigma_{title='Terminator'}(film)$$

- Bereichsanfrage

$$\sigma_{1<rentalPrice<4}(film)$$

- Konjunktion (d.h. logisches und)

$$\sigma_{kind='F' \ \wedge \ rentalPrice=4}(film)$$

- Disjunktion (d.h. logisches oder)

$$\sigma_{rentalPrice<2 \ \vee \ kind='D'}(film)$$

Ziel

Ersetze die Blätter des Anfrageplans durch spezifische Zugriffs-Methoden, d.h. kann/soll ich einen Index benutzen oder besser einen Sequenziellen-Scan der Datei?

Strategien für konjunktive Anfragen

```
SELECT *
FROM customer
WHERE name = 'Jensen' AND street = 'Elm'
      AND state = 'Arizona'
```

- Können die Indexe auf (name) und (street) benutzt werden?
- Kann der Index auf (name, street, state) benutzt werden?
- Kann der Index auf (name, street) benutzt werden?
- Kann der Index auf (name, street, city) benutzt werden?
- Kann der Index auf (city, name, street) benutzt?

Optimierung für konjunktive Anfragen: Indexe anschauen!

Strategien für konjunktive Anfragen

```
SELECT *
FROM customer
WHERE name = 'Jensen' AND street = 'Elm'
      AND state = 'Arizona'
```

- Können die Indexe auf (name) und (street) benutzt werden? Ja
- Kann der Index auf (name, street, state) benutzt werden? Ja
- Kann der Index auf (name, street) benutzt werden? Ja
- Kann der Index auf (name, street, city) benutzt werden? Ja
- Kann der Index auf (city, name, street) benutzt? Nein

Optimierung für konjunktive Anfragen: Indexe anschauen!

Strategien für Punkt- und Bereichsanfragen

Lineare Suche

- Teuer, aber immer anwendbar

Binäre Suche

- Nur, wenn Daten passend sortiert sind

Suche mit Hash-Index

- Gut für Punktanfragen, aber nicht geeignet für Bereichsanfragen

Suche mit Primär/Clustering

- Für jeden Index Eintrag ein Block/Seite mit mehreren Einträgen (sparse)
- Verweis durch einen einzelnen Verweis (Pointer) auf den ersten Eintrag dort

Suche mit Sekundärindex

- Implementiert mittels Zeigern auf einzelne Einträge (dense)
- Kann teuer werden

Indexierung Guidelines

Indexieren oder nicht indexieren (Guideline 1):

- Ist der Index überhaupt erforderlich, d.h. gibt es eine Anfrage, die ihn benutzt?
- Falls möglich, lege Indexe an, die mehr als eine Anfrage beschleunigen.

Wahl des Such-Schlüssels (Guideline 2):

Attribute in der WHERE Klausel sind Kandidaten für Indexierung.

- Punktanfrage? Index auf ausgewählte Attribute, idealerweise Hashindex.
- Bereichsanfrage? B+ Baum.

aus dem Buch von Ramakrishnan und Gehrke.

Indexierung Guidelines (2)

Index über mehrere Attribute (Guideline 3):

...sollten in den folgenden beiden Szenarien beachtet werden

- In der WHERE Klausel steht eine Bedingung über mehr als ein Attribut der Relation
- Sie erlauben "Index-Only" Bearbeitung einer Anfrage, d.h. der Index ist ausreichend für die Anfrageverarbeitung. Dazu müssen nicht unbedingt alle Index-Schlüssel auch in der WHERE Klausel vorkommen.

Clustering-Index oder nicht (Guideline 4):

Nur ein Index auf einer Relation kann clustered sein. Clustering kann die Performance erheblich verbessern, also ist die richtige Wahl sehr wichtig.

- Daumenregel: Bereichsanfragen profitieren am meisten vom Clustering. Gibt es mehrere Bereichsanfragen auf versch. Attributen, dann betrachte ihre Selektivität und Häufigkeit.

Indexierung Guidelines (3)

Hash- vs. Baum-Index (Guideline 5):

Ein B+ Baum-Index ist normalerweise einem Hash-Index vorzuziehen da er sowohl Punkt- als auch Bereichsanfragen unterstützt. Ein Hash-Index ist besser falls:

- Es gibt eine sehr wichtige Punkt-Anfrage, keine Bereichsanfragen.
- Ähnlicher Fall: Es wird häufig ein "index nested loops join" (sehen wir gleich) benutzt, mit der indexierten Relation als innere.

Kosten für Instandhaltung der Indexe (Guideline 6):

Welchen Effekt haben die nun identifizierten Indexe auf die Updates?

- Sind Updates häufig und der Index verlangsamt diese drastisch, lösche den Index.
- Indexe können dennoch auch die Performance von Updates unterstützen.

Beispiel

SELECT E.ename, D.mgr
FROM Employees E, Departments D
WHERE D.dname = 'Toy' AND E.dno = D.dno

Welche Indexe sollen wir hier anlegen?

Beispiel

SELECT E.ename, D.mgr
FROM Employees E, Departments D
WHERE D.dname = 'Toy' AND E.dno = D.dno

Welche Indexe sollen wir hier anlegen?

- Klar: Hash Index auf D.dname
- Für jedes sich qualifizierende Tupel finden wir dann mit Index auf E.dno die Join-Partner.
- Brauchen wir einen Index auf dno in der Relation Departments? Nein.