



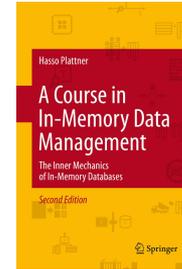
Datenbanken: Indexstrukturen

Dr. Matthias Uflacker, Stefan Halfpap, Dr. Werner Sinzig

12. Juni 2019

- Einführung zu Unternehmensanwendungen (2 Vorlesungen)
- Einführung zu relationalen Datenbanken und Anfrageverarbeitung (2 Vorlesungen)
- Grundlagen des IT-gestützten Rechnungswesens und der Planung (3 Vorlesungen)
- Grundlagen von (spaltenorientierten) Hauptspeicherdatenbanken (5 Vorlesungen)
- **Trends in Hauptspeicherdatenbanken (4 Vorlesungen)**
- Klausur

- Hasso Plattner „A Course in In-Memory Data Management“



- Andy Pavlo „Database Systems“ - Hash Tables & Tree Indexes

<https://15445.courses.cs.cmu.edu/fall2018/slides/06-hashtables.pdf>

<https://15445.courses.cs.cmu.edu/fall2018/slides/07-trees1.pdf>

<https://15445.courses.cs.cmu.edu/fall2018/slides/08-trees2.pdf>

- Felix Naumann: Datenbanksysteme II Indexstrukturen (Klassische Indexstrukturen)

Anwendungen arbeiten nur mit einer kleinen Untermenge aller Daten

- Spaltenorientierung gut, da meist wenige Attribute verwendet werden
- Einige Anfragen beziehen sich nur auf wenige Tupel;
Spaltenorientierung gut, da auf jeder Spalte effizient gefiltert werden kann
- Aber: Idealerweise können Tupel direkt gefunden werden ohne alle Zeilen der Tabelle zu durchsuchen

Idee: Nutzung von Hilfsstrukturen (Effizienz vs. Speicherverbrauch)

Indexstrukturen

Keine Struktur ist die Beste (für alle Anwendungen)

- Zugriffstypen: Gleichheit vs. Bereichsabfragen
- Zugriffszeit - Zeit ein Datenelement /eine Menge von Elementen zu finden
- Einfügezeit - Such-/Zugriffszeit + Aktualisierung der Indexstruktur
- Löschezit - Such-/Zugriffszeit + Aktualisierung der Indexstruktur
- Speicherverbrauch

- Hilfsstruktur, die Zugriff auf Tupel beschleunigen
 - Suchschlüssel: ein oder mehrere Attribute
(Partieller Index: Untermenge der Attribut-Werte) <https://www.postgresql.org/docs/current/indexes-partial.html>
 - Nachteile: Speicherverbrauch und Aktualisierungskosten
- Klassifikation
 - Sortierte Indexe: Basieren auf sortierte Reihenfolge der Schlüsselattribute
 - Hash-Indexe: Basieren auf (Hash-)Funktion, die Schlüssel gleichmäßig auf Behälter aufteilt

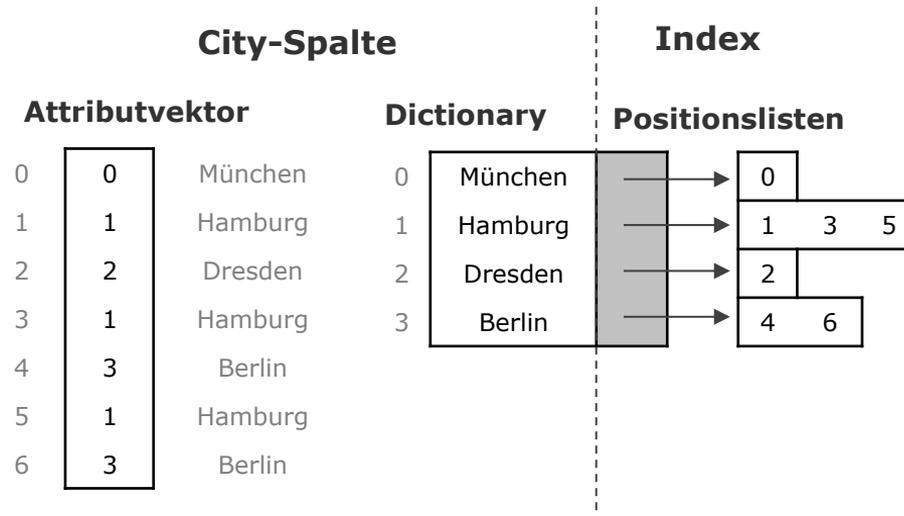
- **Bedeutung von Indexstrukturen für spaltenbasierte DBs**
 - **Indexstrukturen für Dictionary-kodierte Spalten**
 - Indexstrukturen
 - Überblick
 - Kuckucks-Hashing
 - Skip-Liste
 - Bloomfilter
- } Kurzvorstellung von Datenstrukturen/Konzepten,
die über "Standard-DB-Vorlesungen" hinausgehen

- Anwendungen arbeiten nur mit einer kleinen Untermenge aller Daten
 - Spaltenorientierung gut, da meist nur 10% der Attribute verwendet werden

Hasso Plattner: A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. (2009)
 - Scan auf jeden Spalte effizient um Tupel zu filtern
(Indexstrukturen bei zeilenorientieren wichtiger)
- Aber: Komplexität des Scans ist $O(n)$
 - Indexstrukturen können insbesondere selektive Scans (mit kleiner Ergebnismenge) auf Tabellen mit vielen Tupeln beschleunigen

Kester et al.: Access Path Selection in Main-Memory OptimizedData Systems: Should I Scan or Should I Probe? (2017)
<https://stratos.seas.harvard.edu/files/stratos/files/accespathselection.pdf>
 - Schlüsselbedingungen beim INSERT überprüfen
 - Unsortiertes Dictionary benötigt Index um effizient Wert-IDs für Werte zu finden
- (Temporäre Indexstrukturen zur Anfragebearbeitung, z.B. beim Hash-Join)

- Dictionary speichert bereits alle verschiedenen Werte → Schlüsselattribute des Index
 - Idee: speichere pro Schlüsselattribut eine Positionsliste die alle Offsets in den Attributvektor pro Attributwert speichert

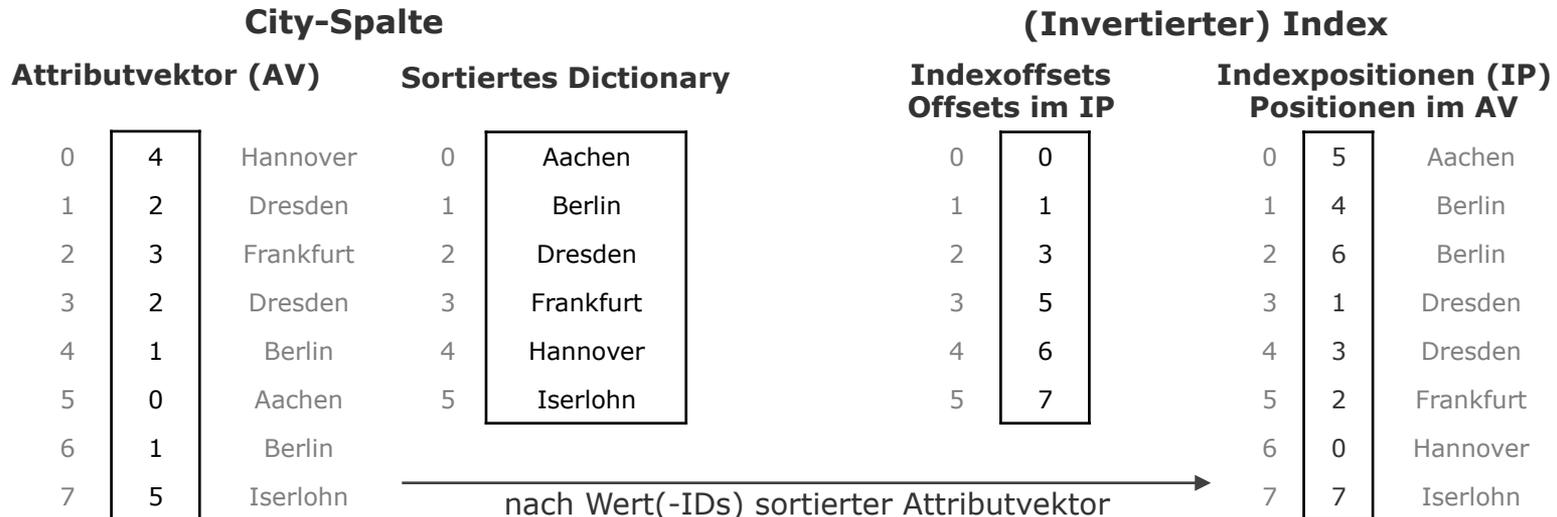


Indexstrukturen für Dictionary-kodierte Spalten

Sortiertes Dictionary (Main Partition oder komprimierter Chunk)

Faust: Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance. (2012)

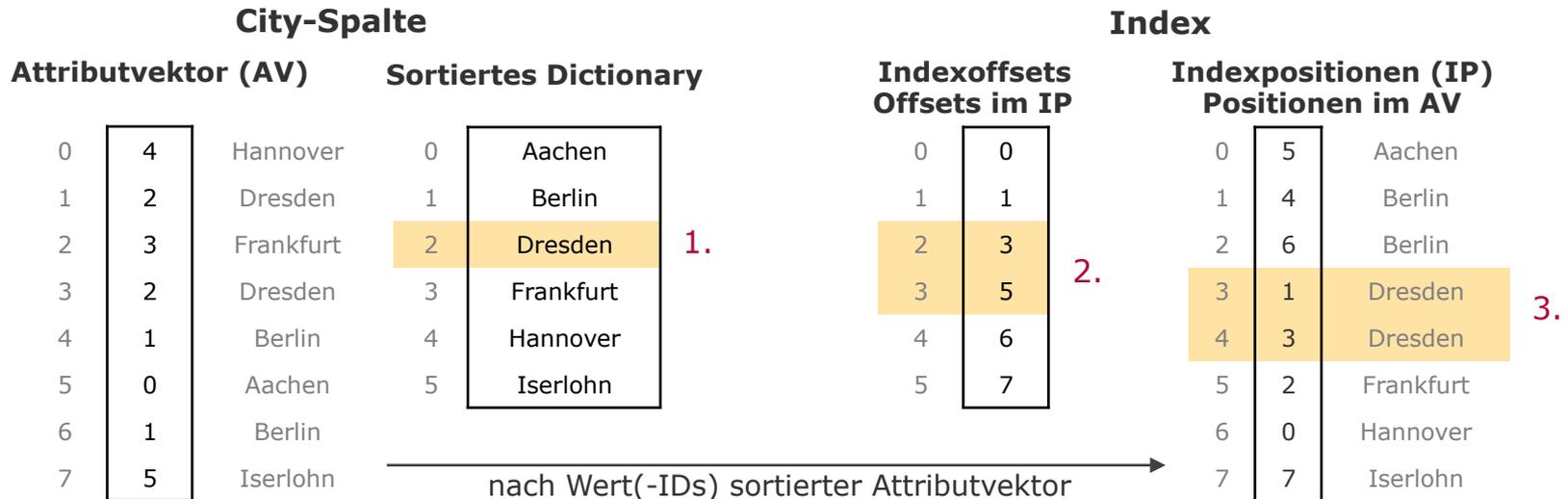
- Partition/Chunk ist unveränderbar (bis auf Invalidierungen)
- Index muss nicht aktualisiert werden
- Attributvektor ordnet (impliziten) Positionen Werte (Wert-IDs) zu
- (Invertierter) Index ordnet Werten Positionen zu



Indexstrukturen für Dictionary-kodierte Spalten

Filter Spalte mit sortiertem Dictionary nach 'Dresden'

1. Suche Wert-ID für 'Dresden' im sortieren Dictionary → 2
2. Lese Indexoffsets für gefundene Wert-ID → [3; 5)
3. Lese Positionen zwischen den Offsets → [1, 3]



Indexstrukturen für Dictionary-kodierte Spalten

Unsortiertes Dictionary (Delta Partition)

Karnagel et al.: Improving In-Memory Database Index Performance with Intel Transactional Synchronization Extensions. (2014)

- Unsortiertes Dictionary benötigt Index um effizient Wert-IDs für Werte zu finden
- Beispiel: INSERT von 'Potsdam' in den Attributvektor benötigt Wert-ID für 'Potsdam'

City-Spalte

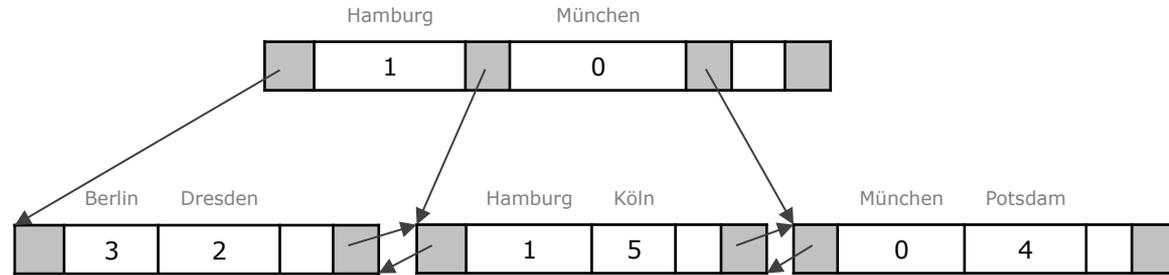
Attributvektor (AV)

| | | |
|---|---|---------|
| 0 | 0 | München |
| 1 | 1 | Hamburg |
| 2 | 2 | Dresden |
| 3 | 1 | Hamburg |
| 4 | 3 | Berlin |
| 5 | 4 | Potsdam |
| 6 | 5 | Köln |

Unsortiertes Dictionary

| | |
|---|---------|
| 0 | München |
| 1 | Hamburg |
| 2 | Dresden |
| 3 | Berlin |
| 4 | Potsdam |
| 5 | Köln |

Index für unsortiertes Dictionary



- Baum ist wichtigste Indexstruktur für Datenbanken
 - Effiziente Indexerstellung
 - Unterstützt Bereichsabfragen und sortiertes Lesen
 - Effiziente Nebenläufigkeitskontrolle

Viele Varianten und Optimierungen ...

Goetz Graefe: Modern B-Tree Techniques. (2011)

- Hash-Index als Alternative mit ebenfalls vielen Varianten - hier: **Kuckucks-Hashing**
- **Skip-Liste** als weitere Indexalternative
- **Bloomfilter** als Struktur um unnötige Zugriffe auf Partitionen zu vermeiden
„Partition Pruning“ (siehe 08_DB_Weitere_Optimierungen)

Indexstrukturen

Hash-Index - Kuckucks-Hashing

- Problem beim Hash-Index - Kollisionen
 - Idee: nutze mehrere Hash-Funktionen/-Tabellen
 - Überprüfe beim Einfügen alle Hash-Tabellen und wähle eine mit einem freien Platz
 - Falls keine Tabelle einen freien Platz hat: vertreibe das Element von einer Tabelle und suche für dieses einen Platz in einer anderen Tabelle
(Erstelle die Tabellen neu, wenn ein Zyklus (Endlosschleife) entsteht)
- Suche und Löschen sind immer $O(1)$,
weil nur eine Position pro Hash-Tabelle überprüft werden muss

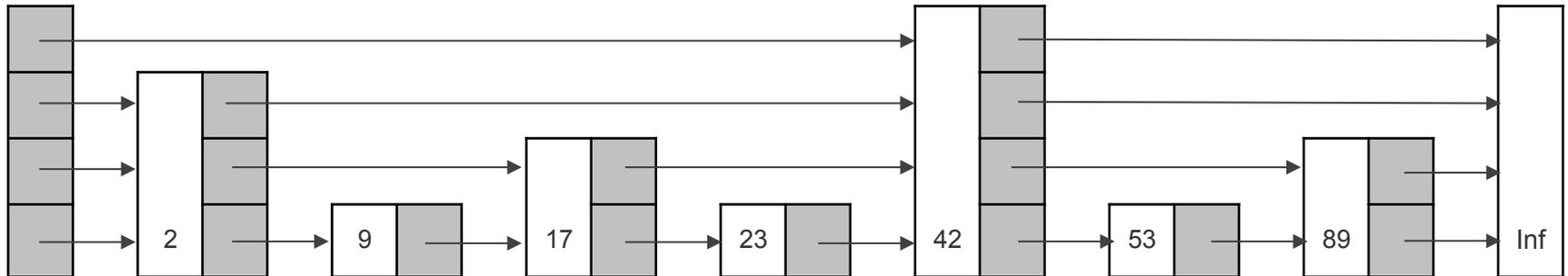
<http://www.cs.tau.ac.il/~shanir/advanced-seminar-data-structures-2007/bib/pagh01cuckoo.pdf>

Indexstrukturen

Skip-Liste

William Pugh. Skip lists: **a probabilistic alternative to balanced trees.** (1990)

- Einfache Variante einer dynamischen Datenstruktur, die Sortierung bewahrt ist, ist eine sortierte verkettete Liste
- Aber alle Operationen benötigen lineare Suche
- Idee: Verwende zusätzliche Pointer um über Listenbereiche zu springen um effizienter zu suchen (Speichere zusätzliche Pointer um binäre Suche zu imitieren)



- Bisher: Effizientes Suchen von vorhandenen Werten
- Häufig hilft es auch zu Wissen, dass Werte nicht vorkommen, z.B. zum Partition Pruning, das ist die Vermeidung unnötiger Scans von Partition, wenn keine relevanten Daten in der Partition gespeichert sind
 - Indexstrukturen können dazu auch verwendet werden, benötigen in der Regel viel Platz
 - Idee: Verwende Speicher-effiziente probabilistische Strukturen z.B. min/max-Aggregate oder **Bloomfilter**

Bloomfilter: Verwende eine kleine Bitmap um alle gespeicherten Werte zu repräsentieren

- Leere Partition entspricht Bitmap mit nur Nullen
- Einfügen: Jeder gespeicherte Wert erzeugt ein Bitmuster, der sich auf die Bitmap legt:
Bitmuster wird durch eine Menge von Hash-Funktionen bestimmt
- Suchen: Bitmuster für gesuchten Wert mit Bitmap abgleichen:
 - Wert ist mit Sicherheit nicht vorhanden, wenn ein gesetztes Bit des Bitmusters in der Bitmap Null ist
 - Ansonsten: ist der Wert *wahrscheinlich* vorhanden
- Löschen: beim normalen Bloomfilter nicht unterstützt (Zählende Bloomfilter als Erweiterung)

Pruning Filter

Bloomfilter - Beispiel

| Gespeicherte Werte | key | Hash1(key) | Hash2(key) | Hash3(key) | Bitmuster |
|--------------------|----------|------------|------------|------------|----------------|
| | Hannover | 0 | 6 | 1 | 1100 0010 0000 |
| | Berlin | 7 | 10 | 8 | 0000 0001 1010 |
| | Köln | 6 | 0 | 8 | 1000 0010 1000 |
| | Hamburg | 0 | 11 | 0 | 1000 0000 0001 |

| Bloomfilter | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
|-------------|---|---|---|---|---|---|---|---|---|---|----|----|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

- Suche 'Berlin': 0000 0001 1010
- Suche 'Potsdam': 0001 0100 0000
- Suche 'Bielefeld': 0100 0010 1000 ← Falsch positive (Wert ist nicht vorhanden, aber mit Bloomfilter nicht feststellbar)

Wahrscheinlichkeit bei 4 Werten, 12 Bits und 3 Funktionen: 0.253

<http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html>

- Indexstrukturen sind für effiziente Scans in spaltenbasierten DBs nicht so wichtig wie in zeilenorientierten
- Dennoch sind Indexstrukturen für Überprüfung von Schlüsselbedingungen, INSERTS bei Spalten mit unsortiertem Dictionary, häufigen selektiven Scans und zur effizienten Anfragebearbeitung auch für spaltenbasierte DBs wichtig
- Bäume und Hash-Tabellen sind die wichtigen Indexstrukturen für DBs
- Kuckucks-Hashing verwendete mehrere Hash-Funktionen/-Tabellen für Kollisionen
- Skip-Listen sind Erweiterung von verketteter Liste mit effizienter Suche
- Bloomfilter ist eine probabilistische Datenstruktur, z.B. zum Partition Pruning