Datenbanken: Kompression von Unternehmensdaten

Stefan Halfpap, Ralf Teusner, Werner Sinzig

5. Juni 2020

Vorlesungsinhalte/-aufbau

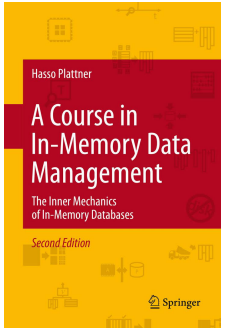
- Einführung zu Unternehmensanwendungen
- Grundlagen des IT-gestützten Rechnungswesens und der Planung
- Einführung zu relationalen Datenbanken und Anfrageverarbeitung
- **Grundlagen von (spaltenorientierten) Hauptspeicherdatenbanken**
- Trends in Hauptspeicherdatenbanken
- Klausur

Überblick

- Motivation
- Dictionary-Kodierung
  - Allgemeines
  - Komprimierung für den Attributvektor
  - Komprimierung für das Dictionary
- Auswahl der Komprimierungskonfiguration
- Zusammenfassung

Literatur und Empfehlung

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



- Daniel Abadi et al. „Integrating Compression and Execution in Column-Oriented Database Systems“ (2006)

  http://db.csail.mit.edu/projects/cstore/abadisigmod06.pdf

- Andy Pavlo „Advanced Database Systems“ - Database Compression

  https://15721.courses.cs.cmu.edu/spring2019/slides/10-compression.pdf

Motivation

- Speicherzugriff ist oft der entscheidende Performanzfaktor bei Datenbanksystemen (CPU vs. Hauptspeichergeschwindigkeit)

Idee: Tausche geringere Speicherzugriffskosten gegen Dekomprimierungs(mehr)aufwand

- Zusätzlich ist Hauptspeicher trotz wachsender Kapazität eine begrenzte (und im Vergleich zu Festplatten kostspielige) Ressource

Kompromiss: Query-Performanz vs. Speicherplatz/Kosten

Motivation

Eigenschaften von Unternehmensdaten ermöglichen sehr gute Kompressionsraten
(Verhältnis von unkomprimierten zu komprimierten Daten)

- Viele Attribute werden in der Regel NICHT benutzt
- Für viele Attribute dominieren NULL oder DEFAULT-Werte
- Viele Attribute haben eine geringe Kardinalität (Anzahl verschiedener Attributwerte)
- Attributwerte sind häufig ungleichmäßig verteilt

Klassifizierung von Kompressionsverfahren

- **Leichtgewichtige** (vs. schwergewichtige) Kompression
  - (fließender Übergang)
  - Datenbanksystem soll Daten trotz Kompression ohne größeren Aufwand verarbeiten können (idealer Weise funktionieren Operationen auf komprimierten Daten)
  - Abhängig vom Zugriffsmuster können auch schwergewichtige Verfahren bei Hauptspeicherdatenbanken verwendet werden (Hyrise unterstützt LZ4)
  - Bei Festplatten-IO kommen eher schwergewichtige Verfahren in Frage (da der Geschwindigkeitsunterschied von CPU und Festplatten-IO größer ist)
- **Verlustfreie** (vs. verlustbehaftete) Kompression
  - Reduktion der Bitanzahl bei gleichem Informationsgehalt

Dictionary-Kodierung

- Dictionary(Wörterbuch)-Kodierung ist eine verlustfreie Kompressionsmethode und Grundlage für weitere Kompressionsverfahren

- Idee: Kodiere jeden unterschiedlichen Wert eines Vektors (mit potenziell sehr großen und häufig vorkommenden Werten) mit einer unterschiedlichen Wert-ID (die klein ist)

Dictionary-Kodierung
Beispiel

- **Zu komprimierender Vektor** V = [Australia, USA, Germany, USA]

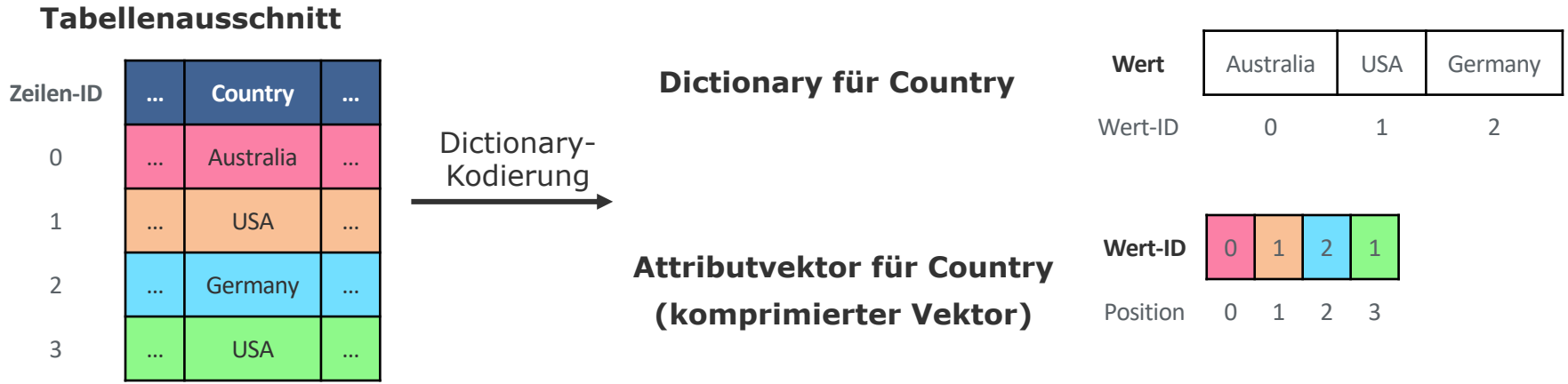
Dictionary-Kodierung 

- Kodierung k (**Dictionary**):
  - Australia → 0
  - USA → 1
  - Germany → 2

- **Kodierter/komprimierter Vektor** k(V) = [0, 1, 2, 1]

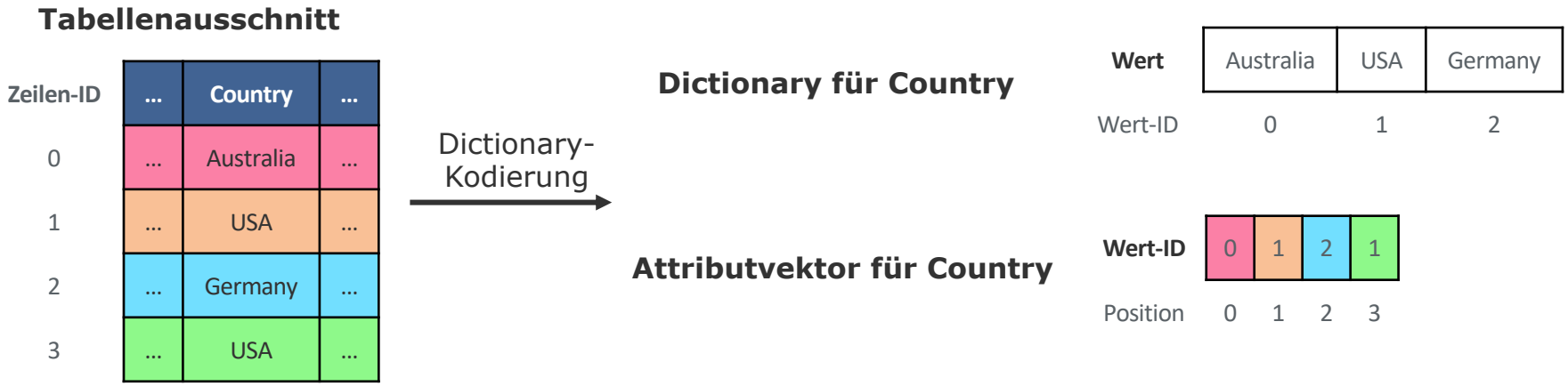
Dictionary-Kodierung
Umsetzung in Datenbanken

- **Direkter Zugriff auf komprimierte Werte** (vs. größte Kompression)
  - **Feste Breite von Wert-IDs** (minimale Anzahl von Bits vs. Byte-Alignment)
  - (Huffman-Kodierung erlaubt keinen direkten Zugriff)
- Datenstruktur für das Dictionary (Kodierung k)
  - Hier: Vektor, wobei das Offset (Position) die Wert-ID implizit bestimmt



Dictionary-Kodierung
Umsetzung in Datenbanken

- **Dictionary** speichert alle verschiedenen Werte mit einer impliziten Wert-ID
- **Attributvektor (komprimierter Vektor)** speichert Wert-IDs für alle Einträge der Spalte



Dictionary-Kodierung
Kompressionsrate der Spalte Country

- Weltbevölkerung: 8 Milliarden Tupel

Zeilen-ID	First Name	Last Name	Country	Year of Birth
0	Paul	Smith	Australia	1986
1	Lena	Jones	USA	1990
2	Hanna	Schulze	Germany	1942
3	Hanna	Schulze	USA	2000
...	...	...	...	...

Dictionary-Kodierung
Kompressionsrate der Spalte Country

- Weltbevölkerung: 8 Milliarden Tupel
- Annahme: 200 verschiedene Länder, alle werden mit 50 Bytes gespeichert
  - Unkomprimiert: $8 * 10^9 * 50\ B = 400 * 10^9\ B = 400\ GB$
  - Dictionary-Größe: $200 * 50\ B = 10.000\ B = 10\ KB = 0{,}00001\ GB$
  - Benötigte Bits pro Wert-ID: $\text{ceil}(\log_2(200))\ b = 8\ b$
  - Größe des Attributvektors: $8 * 10^9 * 8b = 8 * 10^9\ B = 8\ GB$
  - Kompressionsrate = unkomprimierte Größe / komprimierte Größe
    $= 400\ GB / (8{,}00001\ GB) \approx 50$

Dictionary-Kodierung
Daten anfragen

- Suche alle Personen aus den USA
  - Suche die Wert-ID für den gesuchten Wert (USA) O(n)
  - Falls der Wert nicht im Dictionary existiert, muss der Attributvektor nicht durchsucht werden
  - Suche die gefundene Wert-ID (1) im Attributvektor O(n)

Tabellenausschnitt

Zeilen-ID	...	Country	...
0	...	Australia	...
1	...	USA	...
2	...	Germany	...
3	...	USA	...

Dictionary-Kodierung

Dictionary für Country

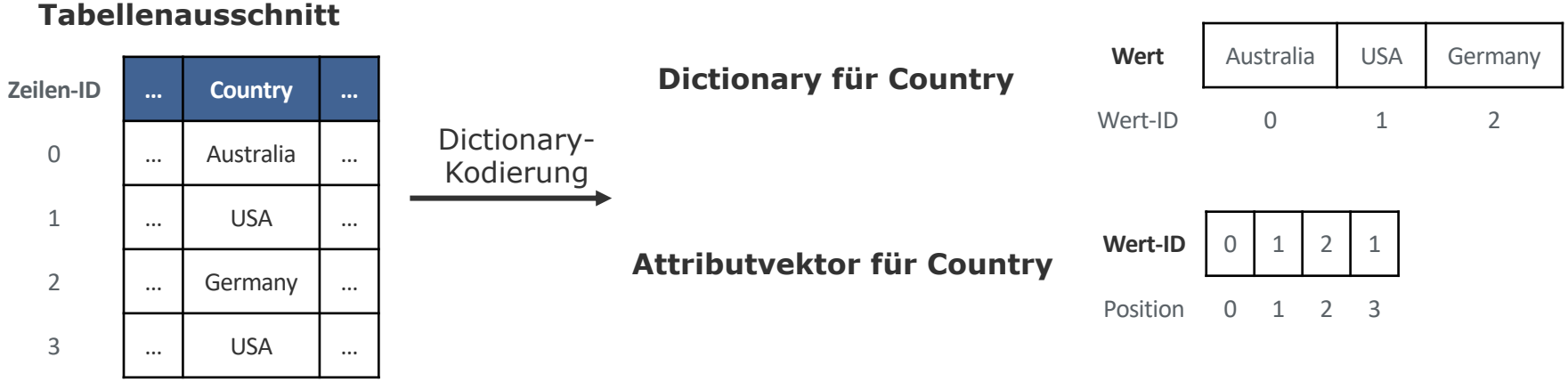
Wert	Australia	USA	Germany
Wert-ID	0	1	2

Attributvektor für Country

Wert-ID	0	1	2	1
Position	0	1	2	3

Dictionary-Kodierung
Bereichsanfragen

- Suche alle Personen aus Ländern mit den Anfangsbuchstaben A - G
  1. Suche alle Wert-IDs für die gesuchten Werte und anschließend im Attributvektor
  2. Lese für jede Wert-ID im Attributvektor den Wert aus dem Dictionary während des Scans



Dictionary-Kodierung
Daten einfügen

- Füge Person aus Deutschland ein
  - Suche die Wert-ID für den gesuchten Wert (Germany) O(n)
    (Füge gegebenenfalls neuen Wert in das Dictionary ein) O(1)
  - Füge die gefundene Wert-ID (2) im Attributvektor ein O(1)

Tabellenausschnitt

Zeilen-ID	...	Country	...
0	...	Australia	...
1	...	USA	...
2	...	Germany	...
3	...	USA	...
4	...	Germany	...

Dictionary-Kodierung

Dictionary für Country

Wert	Australia	USA	Germany
Wert-ID	0	1	2

Attributvektor für Country

Wert-ID	0	1	2	1	2
Position	0	1	2	3	4

Dictionary-Kodierung
Daten einfügen

- Suche der Wert-ID im unsortierten Dictionary in O(n) für die Praxis meist zu langsam
  - Suche beim Einfügen für jedes Attribut notwendig
  - Kosten steigen je mehr verschiedene Werte eingefügt werden

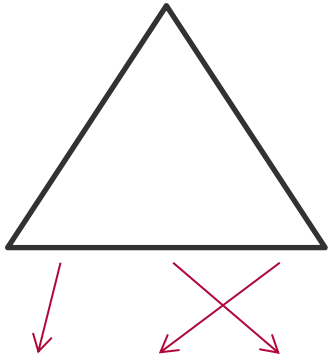
  → Hilfsstruktur, die effizientes Suchen und Einfügen erlaubt

Tabellenausschnitt

Zeilen-ID	...	Country	...
0	...	Australia	...
1	...	USA	...
2	...	Germany	...
3	...	USA	...

Dictionary-Kodierung →

Suchbaum für Country



Dictionary für Country

Wert	Australia	USA	Germany
Wert-ID	0	1	2

Attributvektor für Country

Wert-ID	0	1	2	1
Position	0	1	2	3

Dictionary-Kodierung
Positionslisten materialisieren

- Finde das Land für die an Position 3 gespeicherte Person
  - Lese die Wert-ID im Attributvektor an Position 3 O(1)
  - Lese für die gefundene Wert-ID (1) den Wert im Dictionary O(1)

Tabellenausschnitt

Zeilen-ID	...	Country	...
0	...	Australia	...
1	...	USA	...
2	...	Germany	...
3	...	USA	...

Dictionary-Kodierung

Dictionary für Country

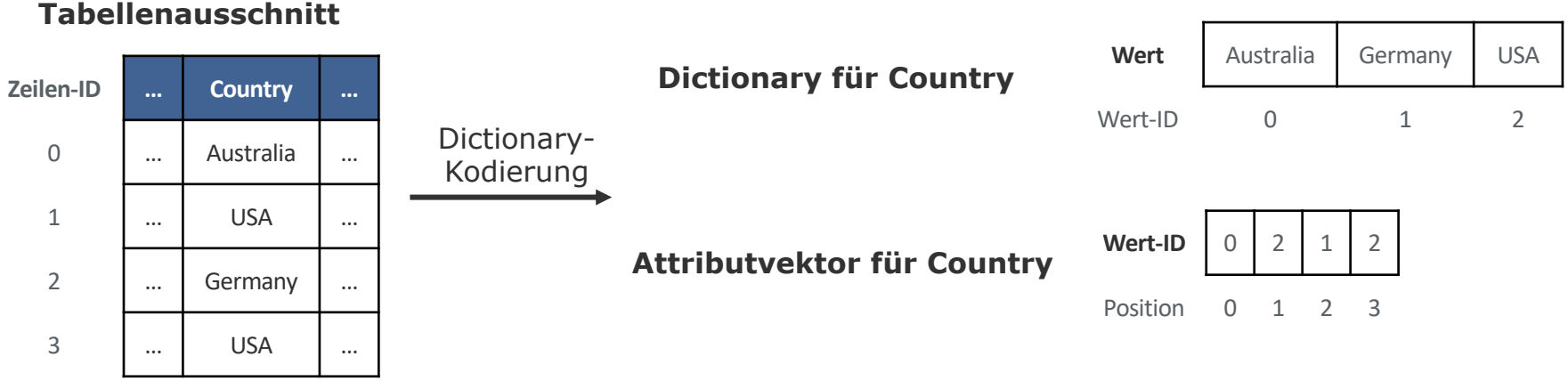
Wert	Australia	USA	Germany
Wert-ID	0	1	2

Attributvektor für Country

Wert-ID	0	1	2	1
Position	0	1	2	3

Dictionary-Kodierung
Sortiertes Dictionary: Vorteile

- Dictionary-Einträge sind nach Wert sortiert
  - Suche der Wert-ID im Dictionary ohne Hilfsstruktur hat Komplexität O(log(n)) statt O(n)
  - Bereichsabfragen (range queries) beschleunigen
  - Dictionary kann besser komprimiert werden



Dictionary-Kodierung
Nachteile

- Sortiertes Dictionary
  - „Umsortierung“ für jeden neuen Wert, der nicht ans Ende des Dictionary gehört (vergleichsweise günstig)
  - Attributvektor aktualisierten (teuer, da dieser (meist) größer ist)
  - Alternative: Dictionary vorher füllen (z.B. bei bekanntem Wertebereich) oder Lücken lassen

- Allgemein
  - Zusätzliche Indirektion für Dictionary-Lookup (z.B. bei der Materialisierung von Anfrage(zwischen)ergebnissen aus Positionslisten oder beim Einfügen neuer Werte)

Dictionary-Kodierung
Gültigkeitsbereich

- (bis jetzt) Einzelne Spalte: aufwändig zu aktualisieren

- Einzelner Block innerhalb der Spalte (Chunk siehe 07_DB_Weitere_Optimierungen)
  - Benötigt keine Aktualisierung
  - Potenziell geringere Kompressionsrate

- Über Spalten/Tabellen hinweg
  - Aufwändig zu aktualisieren
  - Ermöglicht effizienten Join mit Wert-IDs

Komprimierung für den Attributvektor und für das Dictionary

- Komprimierung für den Attributvektor

  (diese Verfahren funktionieren (z.T. mit Anpassungen) auch für unkomprimierte Vektoren)

  - Run-Length-Encoding (Lauflängenkodierung)
  - Präfix-Kodierung
  - Cluster-Kodierung
  - Sparse-Kodierung
  - Bitvektor/Bitmap-Kodierung
  - Indirekte Kodierung

- (Komprimierung für das Dictionary)
  - Delta-Kodierung

Komprimierung für den Attributvektor und für das Dictionary

- Operationen auf komprimierte Daten:
  - Scan des gesamten Vektors, z.B. Selektion nach bestimmten Wert oder Wertebereich
  - Punktzugriff: lese Wert aus Attributvektor/Dictionary an bestimmter Position, z.B. Materialisierung von Positionslisten
  - (Einfügen neuer Werte)

- Punktzugriff nur bei Werten fester Länge (Byte-Offset) oder mit Hilfsstrukturen möglich, aber beides verbraucht Speicherplatz

→ Allgemeine Idee: unterteile den Vektor in Blöcke, sodass der Block für den Punktzugriff effizient gefunden werden kann

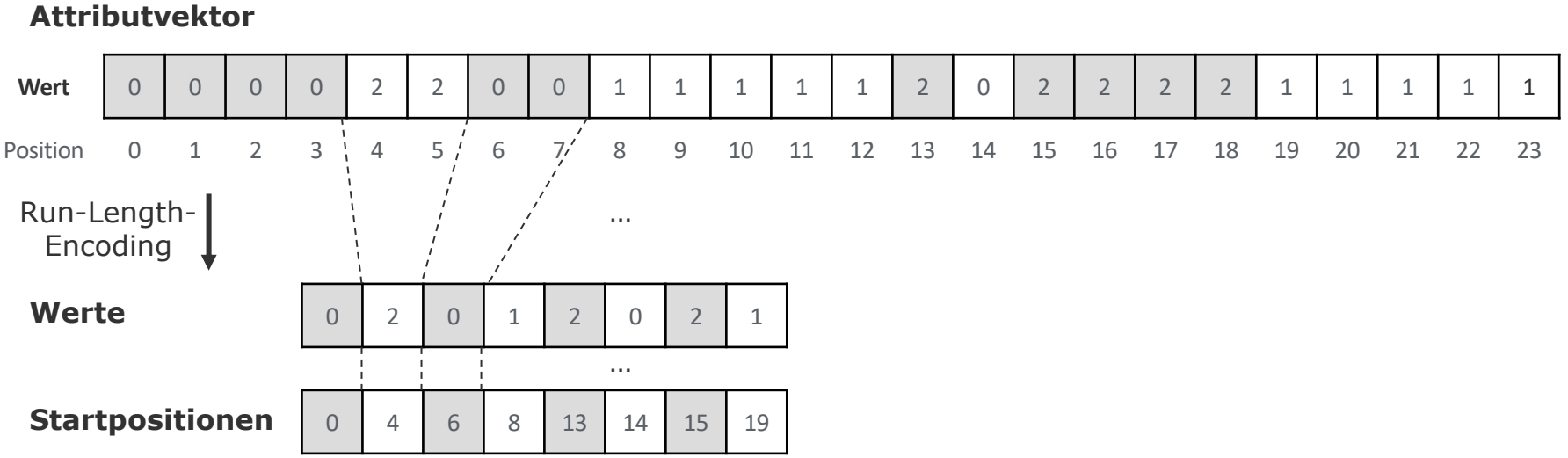
Komprimierung für den Attributvektor
Run-Length-Encoding (Lauflängenkodierung)

- Komprimierungsvervahren für Sequenz von sich wiederholenden Werten
- Grundidee: Speicherung von Wert und dessen Anzahl
- Verschiedene Verfahren/Implementierungen

- Beispiel im Datenbankkontext:
  - Wiederholung von Werten als Tupel (Wert, Startposition, Wiederholungen)
  - Optimierung: Für Vektoren kann die Anzahl der Wiederholungen aus Startpositionen berechnet werden (andersrum kann die Startposition auch aus allen vorherigen Wiederholungen berechnet werden)

- Besonders gut für sortierte Spalten (viele Wiederholungen)

Komprimierung für den Attributvektor
Run-Length-Encoding - Beispiel



Komprimierung für den Attributvektor
Run-Length-Encoding - Effizienz

- Effizienter Scan des Werte-Vektors
  - Positionsliste kann direkt aus Startpositionen-Vektor erstellt werden
  - (Positionsliste kann auch komprimiert werden)
- Kein direkter Punktzugriff: Binäre Suche (oder Interpolationssuche) im Startpositionen-Vektor notwendig

Komprimierung für den Attributvektor
Präfix-Kodierung

- „Run-Length-Encoding des ersten Wertes“
- Direkter Zugriff auf Attributvektor

Attributvektor

Werte	0	0	0	0	2	2	0	0	1	1	1	1	1	2	0	2	2	2	2	1	1	1	1	1

Präfix-Kodierung ↓ ...

Attributvektor

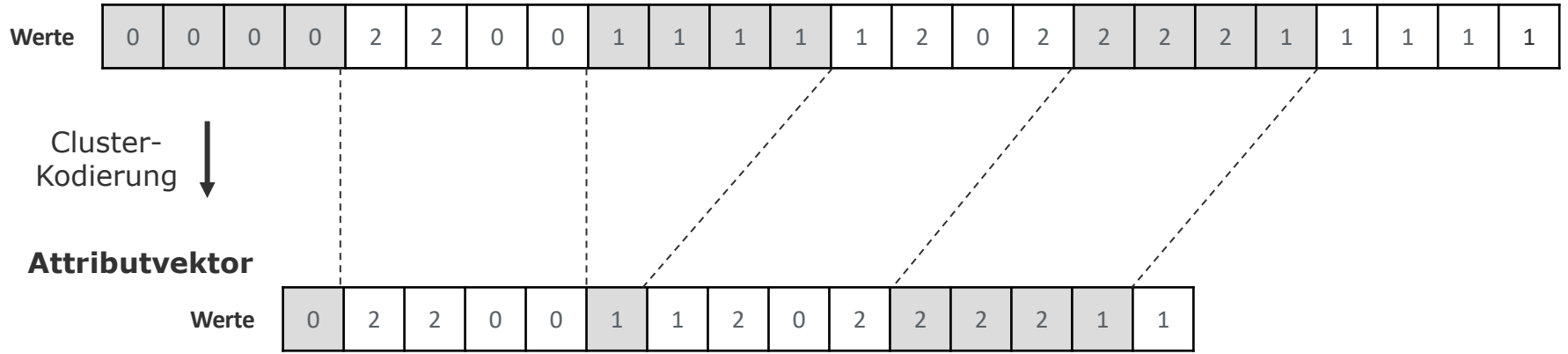
Werte	0	2	2	0	0	1	1	1	1	1	2	0	2	2	2	2	1	1	1	1	1

Anzahl des ersten Wertes: 4

Komprimierung für den Attributvektor
Cluster-Kodierung

- Vektor ist in Blöcke/Cluster fester Größe zerlegt
- Falls ein Cluster nur gleiche Werte beinhaltet, wird das Cluster durch diesen Wert kodiert
- Ein Bitvektor kennzeichnet, welche Cluster kodiert sind

Attributvektor (Blockgröße: 4)



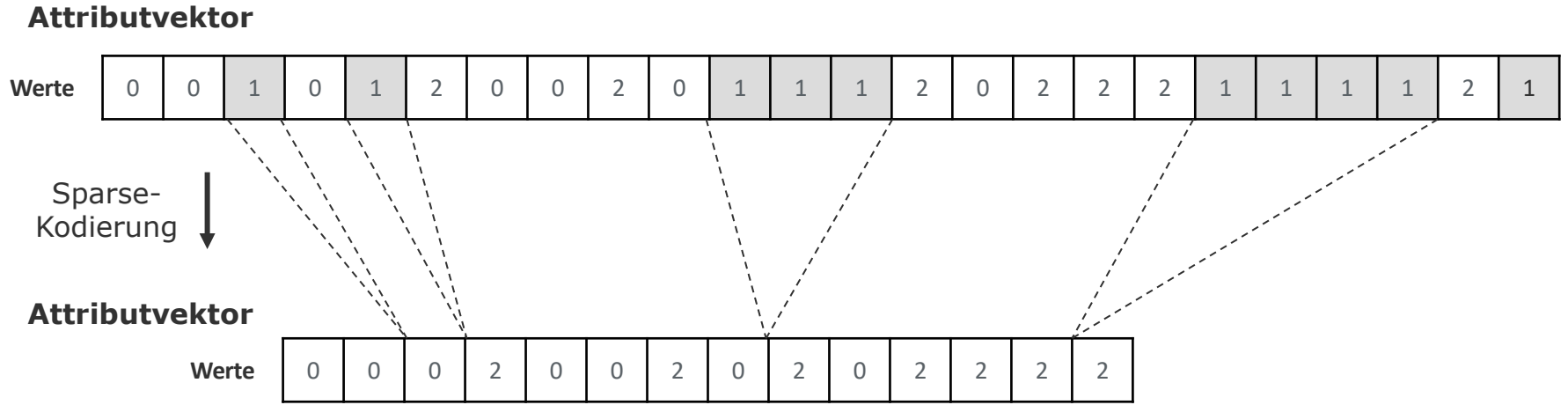
Bitvektor **10 10 01** (1 falls Cluster-kodierter Block)

Komprimierung für den Attributvektor
Cluster-Kodierung - Effizienz

- Effizienter Scan des Werte- und Bitvektors: Aktuelle Position muss gewartet werden (Blockgröße sollte daher nicht zu klein sein)
- Kein direkter Punktzugriff:
  - Scan des Bitvektors nötig
  - Bitvektor kann in Blöcke unterteilt werden

Komprimierung für den Attributvektor
Sparse-Kodierung

- Entferne den Wert, der am häufigsten im Vektor vorkommt
- Ein Bitvektor kennzeichnet, an welchen Positionen ein Wert entfernt wurde



Bitvektor **00101000 00111000 00111101** (1 falls Wert entfernt)

Komprimierung für den Attributvektor
Sparse-Kodierung - Effizienz

- Effizienter Scan des Werte- und Bitvektors: Aktuelle Position muss gewartet werden (Bitvektor ist größer als bei Cluster-Kodierung)
- Kein direkter Punktzugriff:
  - Scan des Bitvektors nötig (Bitvektor ist größer als bei Cluster-Kodierung)
  - Bitvektor kann in Blöcke unterteilt werden

Komprimierung für den Attributvektor
Bitvektor/Bitmap-Kodierung

- Sparse-Kodierung speichert die Positionsliste des häufigsten Wertes als Bitvektor

Bitvektor/Bitmap-Kodierung

- Speichere einen Bitvektor für jeden verschiedenen Wert des Vektors

Attributvektor

Werte	0	0	1	0	1	2	0	0	2	0	1	1	1	2	0	2	2	2	1	1	1	1	2	1

Bitvektor Kodierung ↓

Attributvektor

Bitvektor für 0 **11010011 01000010 00000000** (1 falls Wert übereinstimmt, sonst 0)
Bitvektor für 1 **00101000 00111000 00111101**
Bitvektor für 2 **00000100 10000101 11000010**

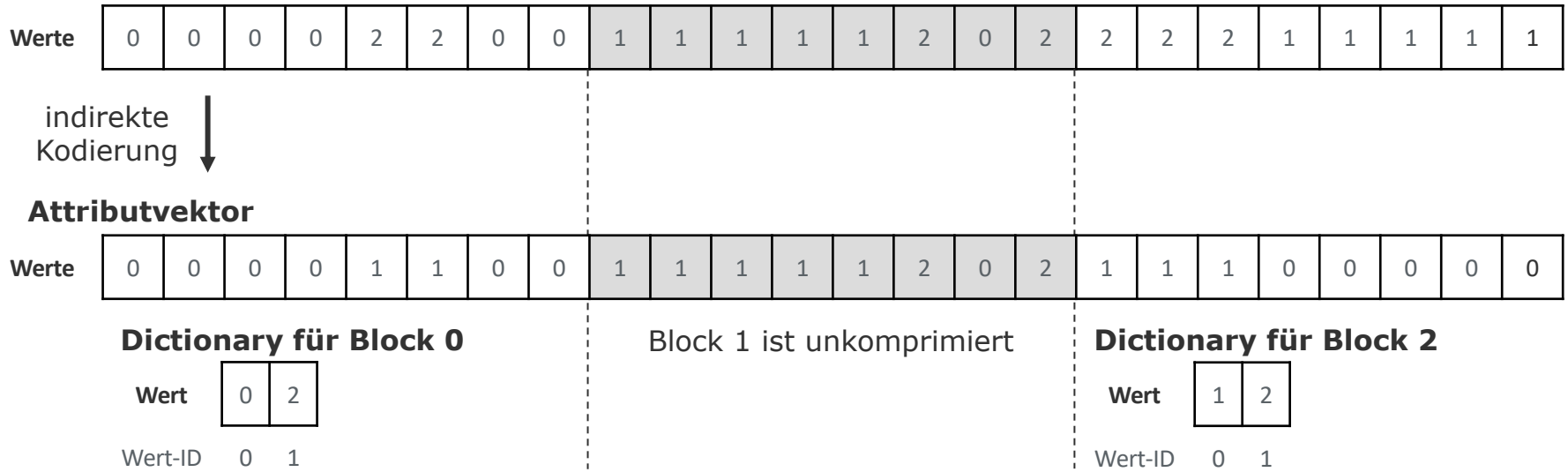
Komprimierung für den Attributvektor
Bitvektor/Bitmap-Kodierung - Effizienz

- Gute Kompressionsrate nur bei wenig verschiedenen Werten
- Effizienter Scan
  - Bitvektor/Bitmap-Kodierung speichert Positionsliste aller Werte als Bitvektor
  - Bitvektoren gleicher Spalten und Tabellen können mit Bitoperationen kombiniert werden
- Punktzugriff: Lese alle Bitvektoren der Spalte an gesuchter Position

Komprimierung des Attributvektors
Indirekte Kodierung

- Vektor ist in Blöcke fester Größe zerlegt
- Falls ein Block wenige verschiede Wert enthält, wird er über ein zusätzliches Dictionary kodiert; die Wert-IDs werden dann mit weniger Bits dargestellt

Attributvektor (Blockgröße: 8)



Komprimierung für den Attributvektor
Indirekte Kodierung - Effizienz

- Effizienter Scan pro Block; Wert-ID muss pro Block im Dictionary gesucht werden
- Direkter Punktzugriff

Komprimierung für das Dictionary
Delta-Kodierung

- Vektor **vom Typ String** ist in Blöcke mit einer festen Anzahl von Werten zerlegt
- Blöcke werden einzeln komprimiert und speichern für jeden Wert:
  - Länge des Präfix, welchen der Wert mit dem **unmittelbaren Vorgänger** gemeinsam hat
  - Anzahl der zusätzlichen Zeichen ohne den gemeinsamen Präfix
  - Zusätzlichen Zeichen

Dictionary

Werte	Bahamas	Bahrain	Bangladesh	Barbados	Belarus	Belgium

Delta-Kodierung ↓

Werte	7	Bahamas	3	4	rain	2	8	ngladesh	2	6	rbados	1	6	elarus	3	4	gium

(Vereinfachte Sicht:
Länge muss auch hier gespeichert werden)

Delta-Kodierung
Frame-of-Referenz-Kodierung

Delta-Kodierung

- Delta-Kodierung funktioniert auch für Vektoren **mit numerischen Werten** (oder beliebigen Typen, deren Werte sich aus einen anderen Wert und einem Delta beschreiben lassen)
- Blöcke werden einzeln komprimiert und speichern für jeden Wert die Differenz (das Delta) zum Vorgänger
- Idee: Differenzen sind kleiner als absolute Werte und benötigen weniger Speicherplatz

Frame-of-Referenz-Kodierung

- Für Punktzugriffe innerhalb eines Blockes müssen bei der Delta-Kodierung alle Vorgängerwerte dekodiert werden
- Idee: Speichere nicht die Differenz zum Vorgänger, sondern zum ersten Wert des Blockes

→ Direkter Punktzugriff

Auswahl der Komprimierungskonfiguration

Unterschiedliche „Teile“ der Datenbank können unterschiedlich komprimiert werden

- Granularität:
  - Spalte: wie bisher vorgestellt
  - Partition/Chunk (siehe 07_DB_Weitere_Optimierungen): Abschnitt einer Spalte
  - (Block, Tupel oder einzelnes Attribut bei Zeilenorientierung)

- Geeignetes Kompressionsverfahren hängt verschiedenen Faktoren ab
  - Query-Performanz vs. Speicherbudget (Kosten)
  - Werteverteilung des Attributes
  - Zugriffsmuster des Attributes

Zusammenfassung

- Neben der Reduktion des Speicherbedarfs ist Komprimierung eine Möglichkeit dem Speichzugriffsengpass heutiger Computer entgegenzuwirken
- Besonders wichtig für Hauptspeicherdatenbanken sind Komprimierungsverfahren, die schnelle Scans und direkten Datenzugriff (ohne teure Dekompression) erlauben
- Spaltenorientierte Datenbanken ermöglichen (im Allgemeinen) höhere Kompressionsraten, da sich Werte eines Attributs (meist) ähnlicher sind als die eines Tupels
- Viele Kompressionsverfahren sind erst für sortierte Listen effizient, aber Tabellen können nur nach einer Spalte oder kaskadierend sortiert werden
- Geeignete Kompressionsverfahren ergeben sich aus Daten- und Anfrageeigenschaften (der Spalte)