Datenbanksysteme

Wintersemester 2016/17

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Größe von S	Größe von T	Equi-Join	Theta-Join
Klein	Klein	Einfach	Einfach
Klein	Groß	Replicated-Join (Reduce-Side-Join)	Replicated-Join
Groß	Klein	Replicated-Join (Reduce-Side-Join)	Replicated-Join
Groß	Groß	Reduce-Side-Join (Probleme bei schiefen Verteilungen, wenigen versch. Werten)	?????

Problem Definition

- **Theta-Join** generalisiert den Equi-Join: Für $S = \{s_1, s_2, \ldots\}$ und $T = \{t_1, t_s, \ldots\}$, finde alle Paare (s_i, t_j), die ein gegebenes **Prädikat** $\theta(s_i, t_j)$ erfüllen.
- Typische Beispiele: Ungleicheitsbedingungen oder Distanzen unter Schwellwert
- **Ziel:** Berechnung des Theta-Joins so auf Knoten zu verteilen, dass Antwortzeit (d.h. von Job Start bis Ende) minimiert wird.

Theta-Join-Matrix

- Das Ergebnis eines Joins ist eine Teilmenge des kartesischen Produkts $S \times T$
- Kann also als Matrix M mit $|S|$ Zeilen und $|T|$ Spalten dargestellt werden, wobei
- Matrixeintrag ist "true" (hier farbig markiert), falls die entsprechenden Tupel das Prädikat erfüllen, sonst "false".
- Matrixeintrag $M(i, j)$ entspricht Tupel i aus S und Tupel j aus T.

T: 5 7 7 7 8 9
S: 5 7 7 8 9 9

$S.A = T.A$

T: 5 7 7 7 8 9
S: 5 7 7 8 9 9

$|S.A - T.A| < 2$

T: 5 7 7 7 8 9
S: 5 7 7 8 9 9

$S.A \geq T.A$

Join-Matrix für Equi-Join: Matrix-To-Reducer Mapping

Standard-Equi-Join

S \ T	5	7	7	7	8	9
5	5					
7		7	7	7		
7		7	7	7		
8					8	
9						9
9						9

R1: Key 5, 8
Input: S1, S4
T1, T5
Output: 2 Tupel

R2: Key 7
Input: S2, S3
T2, T3, T4
Output: 6 Tupel

R3: Key 9
Input: S5, S6
T6
Output: 2 Tupel

Zufällig

S \ T	5	7	7	7	8	9
5	3					
7		2	3	1		
7		3	1	2		
8					1	
9						2
9						1

R1: Key 1
Input: S2, S3, S4, S6
T3, T4, T5, T6
Output: 4 Tupel

R2: Key 2
Input: S2, S3, S5
T2, T4, T6
Output: 3 Tupel

R3: Key 3
Input: S1, S2, S3
T1, T2, T6
Output: 3 Tupel

Balanciert

S \ T	5	7	7	7	8	9
5	1					
7	1	2	2			
7	1	2	2			
8					3	3
9					3	3
9					3	3

R1: Key 1
Input: S1, S2, S3
T1, T2
Output: 3 Tupel

R2: Key 2
Input: S2, S3
T3, T4
Output: 4 Tupel

R3: Key 3
Input: S4, S5, S6
T5, T6
Output: 3 Tupel

Ohne Annahmen über Daten und Join-Bedingung zu machen muss die gesamte Join-Matrix (die ja unbekannt ist in diesem Fall) abgedeckt werden (Kreuzprodukt).
Das bedeutet, dass jede Zelle (also der ihr entsprechenden Tupel) der Matrix auf einem Reducer landen sollen.

Z.B. für Aufteilung auf 6 Reducer:

S \ T	5	7	7	7	8	9
5						
7						
7						
8						
9						
9						

S \ T	5	7	7	7	8	9
5	0		1		2	
7						
7						
8	3		4		5	
9						
9						

Man könnte sich also überlegen, das i-te Tupel aus S an Reducer zu schicken, die für Zeile i zuständig sind. Entsprechend für Spalten für T. Map-Tasks sehen aber nur Tupel unabhängig von anderen. Einfacher: Wähle "Zeile" bzw. "Spalte" zufällig aus für jedes S bzw. T Tupel.

1-Bucket-Random Algorithmus

- Gegeben zwei ganze Zahlen A und B: Erzeuge $A \times B$ Tasks, von denen jeder etwa $1/A$ Tupel von S und $1/B$ Tupel von T erhält.
  - Diese Partitionen enthalten zufällig ausgewählte Daten, was für eine sehr ausgeglichene Lastverteilung sorgt.
- Das Endresultat ist die Vereinigung der Einzelresultate.

- **Hauptidee:** Weise jedes S Tupel zufällig an eine von A Zeilen zu; und jedes T Tupel an eine von B Spalten.

0	1	2
3	4	5

Abbildung zeigt Partitionierung für $A = 2$ und $B = 3$. Nummern zeigen IDs der einzelnen Regionen.

Beispiel

	Spalte 0	Spalte 1	**Spalte 2**
Zeile 0	0	1	2
Zeile 1	3	4	**5**

Für Tupel s_i aus S wird Zeile 1 ausgewählt. Das Tupel wird also zu Knoten 3, 4 und 5 gesendet.
Für Tupel t_j aus T wird Spalte 2 ausgewählt. D.h. das Tupel wird zu Knoten 2 und 5 gesendet.

Dann befinden sich Kopien beider Tupel auf Knoten 5 und können dort (wenn Prädikat erfüllt ist) gejoined werden.

1-Bucket-Random: Map

```
map(..., tuple x]) {
    if (x is from S) {
        //select random integer from range [0,..., A-1]
        row = random(0, A-1)

        //emit the tuple for all keys in the selected row
        for key = (row*B) to (row * B+B-1)
            emit(key, (x,'S'))
    }
    else { //x is from T
        //select random integer from range [0,..., B-1]
        col=random(0, B-1)
         //emit the tuple for all keys in selected column
        //requires skipping B region numbers forward from
        //start region key equal to col
        for key=col to ((A-1)*B+col) step B
            emit(key, (x, 'T'))
        }
}
```

	Spalte 0	...	Spalte B-1
Zeile 0	0	1	2
Zeile A-1	3	4	5

1-Bucket-Random: Reduce

```
reduce(regionID, [(x1, flag1), (x2, flag2), ...]) {
    initialize S_list and T_list

    //separate the input list by the data set the tuples came from
    for all (x, flag) in input list do
        if (flag=='S')
            S_list.add(x)
        else
            T_list.add(x)

    joinResult = myFavoriteJoinAlgorithm(S_list, T_list)
    for each tuple t in joinResult do
        emit(t)
}
```

Korrektheit von 1-Bucket-Random

- Nehme ein beliebiges Paar (s, t), $s \in S$ und $t \in T$.
- Tupel s wird allen Schlüsseln in genau einer Zeile zugewiesen; t allen Schlüsseln in genau einer Spalte.
- Darum gibt es genau einen Schlüssel der beide enthält - dort wo Zeile von s und Spalte von t sich überschneiden.
- Die Reduce Ausführung für diesen Schlüssel generiert Ausgabe (s, t), falls das Joinprädikat erfüllt ist.
- Keine andere Reduce Ausführung kann (s, t) generieren.

Bestimmen von A und B

- Sei r die **gewünschte Anzahl von Teilproblemen.**
- Um A und B zu bestimmen benötigt man nur **minimale Informationen über die Daten**, nämlich das Verhältnis $|S|/|T|$.

Sei o.B.d.A. $|S| \leq |T|$ und $C = |S|/|T|$.

- Falls $C < 1/r$, dann setze $A = 1$ und $B = r$
- Sonst, d.h. für $C \geq 1/r$, setze $A = \lfloor\sqrt{C \times r}\rfloor$ und $B = \lfloor\sqrt{C^{-1} \times r}\rfloor$.

Untere Schranken:

- $\frac{|S| \times |T|}{r}$ ist untere Schranke für max-reducer-output
- $2\sqrt{\frac{|S| \times |T|}{r}}$ ist untere Schranke für max-reducer input

Effektivität: Ein- und Ausgabeverteilung

Mit den obigen Werten für A und B kann folgendes gezeigt werden (bzgl. Erwartungswert):

Für Eingabeanteil: Kein Task erhält mehr als $(2 + 1/A + 1/B)/2$ mal die untere Schranke des idealen Eingabeanteils.

- Schlechtester Fall: Für $A = B = 1$ ist der Faktor 2.
- Für realistischere $A = B = 10$ ist der Faktor nur $1{,}1$.

Für Ausgabeanteil: Kein Task generiert mehr als $\frac{(A+1)(B+1)}{A \times B}$ mal die untere Schranke des idealen Eingabeanteils.

- Schlechtester Fall: Für $A = B = 1$ ist der Faktor 4.
- Für realistischere $A = B = 10$ ist der Faktor nur $1{,}21$.

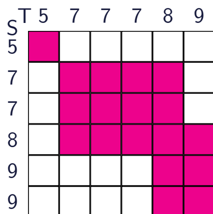
1-Bucket-Random: Eigenschaften

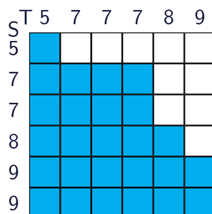
- Fast perfekte Eingabelastverteilung praktisch garantiert.
- Erwartungswert der Ein- und Ausgabelast ist perfekt verteilt.
- Für eine gewünschte Anzahl Tasks können wir immer gute Werte für A und B finden (siehe Folie zu Effektivität)

1-Bucket-Random: Eigenschaften

- Jedes Eingabetupel von S wird B mal kopiert, jedes von T wird A mal kopiert.
- Ausgabetupel werden nicht kopiert.
- Da die Last fast perfekt verteilt wird, **kann 1-Bucket-Random nur verbessert werden, indem man weniger Eingabekopien erzeugt.**
- **Dazu müssen allerdings Eigenschaften des Joinprädikats ausgenutzt werden.**

Matrix Überdeckung - Matrix-To-Reducer Mapping





Jede true-wertige (farbige) Zelle muss von einem Reduce-Task berechnet werden.

Matrix Überdeckung: Anforderungen

- **Jede true-wertige Zelle muss von einem Reduce-Task berechnet werden.**
- Dies führt zu einem Überdeckungsproblem, wo alle true-wertigen Zellen von einem Schlüssel überdeckt werden müssen.

- False-wertige Zellen brauchen nicht überdeckt zu werden. Ihre Überdeckung führt nicht zu falschen Resultaten, da Joinprädikat überprüft wird.
- **Keine Zelle soll von mehr als einem Schlüssel überdeckt werden**, um Duplikate zu vermeiden. Denn: Duplikateliminierung ist teuer.

Reduzierte Überdeckung

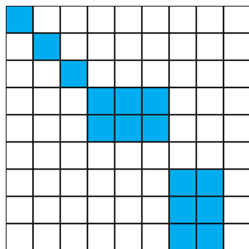
- 1-Bucket-Random ist praktisch optimal in der Klasse der Theta-Join-Algorithmen, die die gesamte Joinmatrix überdecken.
- Um besser zu sein, muss man den zu überdeckenden Teil der Matrix erheblich verringern.
- Da alle true-wertigen Zellen überdeckt werden müssen, kann dies nur über die Identifikation von false-wertigen Zellen geschehen. Aber wie findet man diese, ohne den Join selbst zu berechnen?

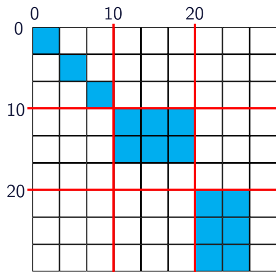
Ausnutzung von Daten- und Joineigenschaften

- Für eine Region der Matrix muss bewiesen werden, dass dort keine true-wertigen Zellen sein können.
- Natürlich muss dies unter geringem Kostenaufwand geschehen.

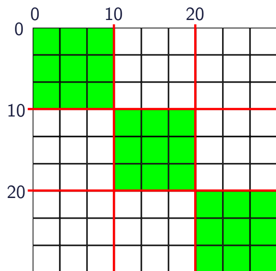
- Beispiel: Für einen Equi-Join $S.A = T.A$ kann dies folgendermaßen erreicht werden.
  - Angenommen A hat Werte zwischen 0 und 30, die in Intervalle $[0{,}10)$, $[10{,}20)$ und $[20{,}30)$ aufgeteilt sind.
  - Ein S-Tupel in Intervall $[10{,}20)$ kann nicht mit T-Tupeln im Interval $[20{,}30)$ verbunden werden usw.
  - Auf diese Weise können die viele Intervall-Kombinationen verworfen werden.

Join-Matrix für Equi-Join $S.A = T.A$





Intervall-partitionierte Matrix. Jedes S Tupel gehört in genau eine der drei Zeilenpartitionen; jedes T Tupel in genau eine der drei Spaltenpartitionen.



Zellen die überdeckt werden müssen sind schwarz markiert. Die Gleichheitsbedingung des Equi-Joins eliminiert 6 von 9 Blöcken.

Algorithmen

- Nachdem Regionen mit zu überdeckenden Zellen identifiziert wurden, müssen sie mit r Schlüsseln überdeckt werden. Aufgrund der komplexeren Struktur ist dies schwieriger als für die gesamte Matrix.
- In Literatur (siehe unten) werden Heuristiken dafür vorgeschlagen, welche sich auf Eingabe- bzw. Ausgabeverteilung konzentrieren.

Alper Okcan, Mirek Riedewald: Processing theta-joins using MapReduce. SIGMOD Conference 2011: 949-960

Anmerkungen

- Intelligente Datenverteilung ist essenziell für effiziente/skalierbare Verarbeitung von großen Datenmengen.
  - Jeder Knoten ist nur für einen kleinen Teil der Daten zuständig.
  - Jeder Knoten kann Arbeit unabhängig von anderen Knoten ausführen, mit geringem Datenaustausch.

Column-Stores, OLTP/OLAP, In-Memory DBS

Data Layouts: Zeilenorientierte Speicherung (Row Store)

- Bislang davon ausgegangen, dass eine Tabelle in einem Datenbanksystem Zeile für Zeile in einer Datei abgespeichert wird.

PersNr	Name	Gehalt	Rang
2125	Sokrates	70000	C4
2127	Kopernikus	20000	C3
2133	Popper	60000	C3
2134	Augustinus	50000	C3
2136	Curie	80000	C4
2137	Kant	50000	C4
2126	Rusel	60000	C4

2125; Sokrates; 70000; C4; 2127; Kopernikus; 20000; C3; 2133; Popper; 60000; C3; 2134; Augustinus; 50000; C3; 2136; Curie; 80000; C4; 2137; Kant; 50000; C4; 2126; Rusel; 60000; C4

Ein Datenbanksystem, das Tupel zeilenweise ablegt, nennt man Row Store.

Überlegungen

- Bei **zeilenorientierter Speicherung** wird bei Datenzugriff also ein Block gelesen, in dem komplette Tupel der Tabelle enthalten sind.
- Es sind also sowohl PersNr, Name, als auch Gehalt verfügbar.

Betrachten wir folgende Anfragen/Anweisungen an die Datenbank:

- Ändere Gehalt von Professor mit PersNr 2136 auf 85000.
- Wie viel verdient Professor Sokrates?
- Wie hoch ist das durchschnittliche Gehalt der Professoren?
- Wie hoch ist das durchschnittliche Gehalt der C4 Professoren?
- Wie viele Professoren gibt es?

Eine einfache Rechnung

Die Tabelle "Schufa" habe 5 Millionen Einträge. Ein Tupel benötigt 500 Bytes (SSN: int, FirstName: char(50), LastName: char(50), Telefon: char(50), Stadt: char(50), Schulden: int, Risiko: float, ...). Insgesamt müssen also 2,5 GB gespeichert werden.

Mit 10ms für Latenz und Lesegeschwindigkeit von 100MB/s, ergibt circa 25 Sekunden für das Lesen dieser Daten.

Was ist mit einer Anfrage "SELECT avg(schulden) FROM Schufa"?

Auch hier müssen 2,5GB Daten von der Festplatte gelesen werden, aber wir brauchen eigentlich nur 5 Millionen * sizeof(int) = 5 000 000 * 4 Byte = 20MB.
20MB könnten in ungefähr 0,2 Sekunden gelesen werden.

OLTP vs. OLAP

Es gibt im wesentlichen die folgenden beiden Unterteilungen von Datenbank-Workloads (d.h. Operationen, die die Datenbank ausführen muss):

Online Transaction Processing (OLTP):

- Änderungs- Einfügeoperationen und (einfache) Anfragen.
- Operatives "Tagesgeschäft", z.B. Bestellannahme, Flugbuchungen.

Online Analytical Processing (OLAP):

- Ist im Gegensatz zu OLTP anfragelastig.
- Aggregation, Aufbereitung und Auswertung (Decision Support)

Je nach Art des Workloads können verschiedene Realisierungen von Datenbanksystemen Vor- oder Nachteile haben.

Spaltenorientierte Speicherung (Column Store)

Idee: Wir speichern die einzelnen Attribute (Spalten) einer Tabelle in getrennten Dateien.

	Col0
o_0	2125
o_1	2127
o_2	2133
o_3	2134
o_4	2136
o_5	2137
o_6	2126

	Col1
o_0	Sokrates
o_1	Kopernikus
o_2	Popper
o_3	Augustinus
o_4	Curie
o_5	Kant
o_6	Rusel

	Col2
o_0	70000
o_1	20000
o_2	60000
o_3	50000
o_4	80000
o_5	50000
o_6	60000

	Col3
o_0	C4
o_1	C3
o_2	C3
o_3	C3
o_4	C4
o_5	C4
o_6	C4

Ein Datenbanksystem, das die Daten nach Spalten anordnet, nennt man Column Store.

Natürlich ist es essentiell, dass **Tupel korrekt rekonstruiert werden können.** Dies geschieht anhand von implizit anhand der Positionen innerhalb der Zeilen, oder explizit durch Schlüssel.

Korrelierte Spalten

Bei korrelierten Spalten ist die Reihenfolge der Daten in den einzelnen Spalten bzgl. der Tupel-Id / Object-Id (oid) identisch.

	Col0		Col1		Col2		Col3
o_0	2125	o_0	Sokrates	o_0	70000	o_0	C4
o_1	2127	o_1	Kopernikus	o_1	20000	o_1	C3
o_2	2133	o_2	Popper	o_2	60000	o_2	C3
o_3	2134	o_3	Augustinus	o_3	50000	o_3	C3
o_4	2136	o_4	Curie	o_4	80000	o_4	C4
o_5	2137	o_5	Kant	o_5	50000	o_5	C4
o_6	2126	o_6	Rusel	o_6	60000	o_6	C4

Korrelierte Spalten

D.h. man braucht die oid nicht zu speichern, da sie implizit vorhanden ist.

Col0	Col1	Col2	Col3
2125	Sokrates	70000	C4
2127	Kopernikus	20000	C3
2133	Popper	60000	C3
2134	Augustinus	50000	C3
2136	Curie	80000	C4
2137	Kant	50000	C4
2126	Rusel	60000	C4

Unkorrelierte Spalten

Bei unkorrelierten Spalten ist die Reihenfolge der Daten in den einzelnen Spalten unabhängig (o_i). D.h. man braucht einen **expliziten Schlüssel**, um die ursprünglichen Tupel rekonstruieren zu können.

Col0

OID	PersNr
5	2137
0	2125
2	2133
1	2127
4	2136
3	2134
6	2126

Col1

OID	Name
1	Kopernikus
0	Sokrates
6	Rusel
2	Popper
4	Curie
5	Kant
3	Augustinus

Col2

OID	Gehalt
3	50000
4	80000
1	20000
0	70000
6	60000
2	60000
5	50000

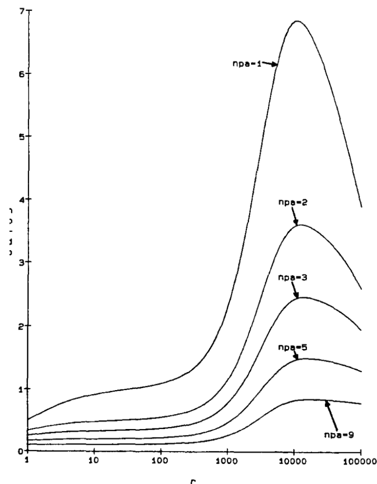
Col3

OID	Rang
1	C3
4	C4
2	C3
6	C4
0	C4
3	C3
5	C4

Tupel Rekonstruktion

Kosten für Rekonstruktion ist abhängig von Anzahl der Attribute, die für die Anfrage relevant sind.

- Entscheidend für Analyse ob für bekannten Workload Column-Store besser ist als Row-Store.



Quelle: G. Copeland and S. Khoshafian. A Decompositional Storage Model. 1985.

Early vs. Late Materialization

Early Materialization

- Lese von Festplatte (oder aus Hauptspeicher) die Spalten, die für die Anfragen benötigt werden.
- Rekonstruiere Tupel.
- Verarbeite Tupel, wie in einem normalen Row-Store.

Late Materialization

- Verschieben der Rekonstruktion auf einen späteren "Zeitpunkt".
- Vorteile: Z.B. durch Selektion und Aggregation ist die Rekonstruktion einiger Tupel unnötig. Zur Rekonstruktion muss Dekomprimiert werden, sondern auf kompakten Daten (weiter) gearbeitet.

Allgemein: Vertikale Partitionierung

Gegeben eine Relation $R(A_1, A_2, \ldots A_n)$.

Es gibt B_n viele Möglichkeiten, Attribute in Partitionen zu unterteilen (Redundanz ist zugelassen). B_n ist die Bellsche Zahl. Für $n = 5$ gibt es bereits 52 Möglichkeiten.

Ein **Column-Store** wird Relation R attributweise abspeichern, d.h. die Partitionierung besteht aus n Tabellen, eine für jedes Attribut A_i.

Ein **Row-Store** betrachtet nur eine Partition, in der alle Attribute gemeinsam sind.

Allgemein gibt es **weitere Möglichkeiten** zu Partitionieren, u.a. kann auch **Redundanz** zugelassen werden (siehe Abschnitt über Schema-Denormalisierung).

Vertikale Partitionierung: Beispiel

	PersNr	Name
o_0	2125	Sokrates
o_1	2127	Kopernikus
o_2	2133	Popper
o_3	2134	Augustinus
o_4	2136	Curie
o_5	2137	Kant
o_6	2126	Rusel

	Rang
o_0	C4
o_1	C3
o_2	C3
o_3	C3
o_4	C4
o_5	C4
o_6	C4

	Name	Gehalt
o_0	Sokrates	70000
o_1	Kopernikus	20000
o_2	Popper	60000
o_3	Augustinus	50000
o_4	Curie	80000
o_5	Kant	50000
o_6	Rusel	60000

Für welche Anfragen ist diese Partitionierung gut geeignet?

Was ist mit Änderungsoperationen? **Partitionierungen, die nicht frei von Redundanz sind haben Nachteile was Änderungsoperationen betrifft.**

DSM vs. NSM

In Literatur wird auch von NSM und DSM gesprochen.

- NSM = N-ary Storage Model (entspricht Row-Store)
- DSM = Decomposition Storage Model (entspricht vertikaler Partitionierung / Column-Store)

G. Copeland and S. Khoshafian. A Decompositional Storage Model. SIGMOD, 1985.

Komprimierung

Schauen wir uns das Beispiel von zuvor nochmal an:

o_0	C4
o_1	C3
o_2	C3
o_3	C3
o_4	C4
o_5	C4
o_6	C4

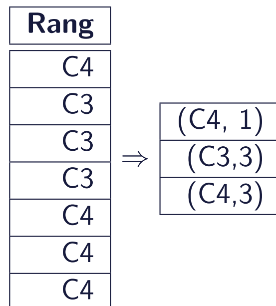
Die Anordnung in Spalten bringt ähnliche (vom Datentyp her aber auch oft lexikographisch/numerisch nah) Attributwerte zusammen.

Idee: Ersetze Daten durch kompaktere Repräsentation.

Potentielle Nachteile: Rechenaufwand bei Dekomprimierung (CPU Kosten)

Komprimierung: Run-Length-Encoding (RLE)

- Fasse Bereiche mit identischen Attributwerten zusammen.
- Ideal geeignet für Column-Stores, im Vergleich zu Row-Stores, da Daten eines einzelnen Attributs zusammen liegen.
- Durch Sortierung kann noch besser komprimiert werden.



Delta Coding

- Speichere anstelle des tatsächlichen Attributwerts die Differenz zum vorherigen Attributwert.
- Wie beim LRE kann Sortierung Effektivität der Komprimierung erhöhen.
- Ebenfalls perfekt geeignet für Column-Stores.

VorlNr		VorlNr
4052		4052
4630		578
5001		371
5022		21
5041	$\Rightarrow$	19
5043		2
5049		6
5052		3
5216		164
5259		43

Dictionary Encoding

Benutze Dictionary, das Daten (Attributwerten) eine kompaktere Stellvertreter-Repräsentation zuordnet.

Fachbereich
Informatik
Wirtschaftswissenschaften
Physik
Informatik
Elektrotechnik und ...
Mathematik
Biologie
Informatik
Wirtschaftswissenschaften
Elektrotechnik und ...

$\Rightarrow$

Fachbereich
0000
0001
0010
0000
0011
0100
0101
0000
0001
0011

Dictionary:

Fachbereich	Encoding
Informatik	0000
Wirtschaftswissenschaften	0001
Physik	0010
Elektrotechnik und ...	0011
Mathematik	0100
Biologie	0101