Vorlesung Wissensentdeckung

Häufige Mengen in Datenströmen

Katharina Morik, Claus Weihs

LS 8 Informatik
Computergestützte Statistik
Technische Universität Dortmund

21.7.2009

Gliederung

1. Frequent Itemsets über Datenströmen
   - Lossy Counting Algorithmus
2. Finding Hierarchical Heavy Hitters in Streaming Data
3. Problem Definition
   - Notation
   - Definitionen
4. Online Algorithmen
   - Lossy Counting Algorithmus
   - Eindimensionale online Algorithmen
   - Mehrdimensionale Algorithmen
5. Experimente
   - Experimentergebnisse

Frequent Itemsets über Data Streams

- Der Schlüssel für Assoziationsregeln sind die häufigen Mengen.
- Diese häufigen Mengen bei einer Abfrage schnell zu liefern, kann bei großen Datenbanken, die ja ständig wachsen, dauern!
- Also sehen wir Transaktionen als Datenstrom an und verwalten online auf diesen Daten die häufigen Mengen.

Online zählen

- Zählen ist elementar beim Finden häufiger Mengen.
- Aber wie zählen wir auf einem Datenstrom, ohne linear zur Eingabe Speicher zu benötigen?
- Bei großen Datenströmen ist das eine kritische Komponente!

Lossy Counting Algorithmus

- Ein möglicher Algorithmus ist der Lossy Counting Algorithmus

  Manku and Motwani
  Approximate Frequency Counts over Data Streams.
  Proceedings of the 28th VLDB Conference Hong Kong, China, 2002

technische universität dortmund
LS 8 Informatik Computergestützte Statistik Technische Universität Dortmund
Frequent Itemsets über Datenströmen Finding Hierarchical Heavy Hitters in Streaming Data Problem Definition Online Algorithme
Lossy Counting Algorithmus

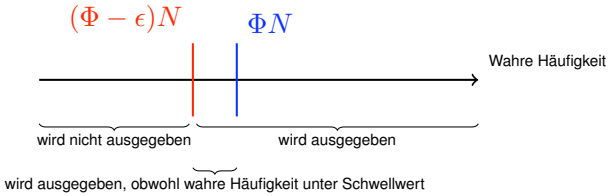
- Wenn wir nicht linear zum Strom speichern wollen, muss die Häufigkeit geschätzt werden.
- Als Parameter gibt es den Schwellwert (analog zu sup_{min}) $\Phi \in (0, 1)$ und den Fehler $\epsilon \ll \Phi$
- N sind die bisherigen Items, die im Strom aufgetaucht sind.

Katharina Morik, Claus Weihs DMV

technische universität dortmund
LS 8 Informatik Computergestützte Statistik Technische Universität Dortmund
Frequent Itemsets über Datenströmen Finding Hierarchical Heavy Hitters in Streaming Data Problem Definition Online Algorithme

Lossy Counting Garantien

- Lossy Counting macht folgende Garantien:
  - Alle Items, deren wahre Häufigkeit ΦN überschreitet, werden ausgegeben. Es gibt also keine "*false negatives*".
  - Keine Items, deren wahre Häufigkeit unter $(\Phi - \epsilon)N$ liegt, werden ausgegeben.
  - Die geschätze Häufigkeit ist stets höchstens ϵN kleiner als die wahre.



Katharina Morik, Claus Weihs DMV

technische universität dortmund
LS 8 Informatik Computergestützte Statistik Technische Universität Dortmund
Frequent Itemsets über Datenströmen Finding Hierarchical Heavy Hitters in Streaming Data Problem Definition Online Algorithme
Lossy Counting Algorithmus

- Der Datenstrom wird aufgeteilt in Fenster (*Buckets*) der Breite $w = \lceil \frac{1}{\epsilon} \rceil$
- Die Fenster werden identifiziert: erstes Fenster *id*$= 1$
- Aktuelles Fenster ist $b_{current} = \lceil \frac{N}{w} \rceil$
- Datenstruktur $\mathcal{D}$ speichert Einträge in der Form (e, f, Δ)
  - e - Element aus dem Strom
  - f - geschätzte Häufigkeit
  - Δ - maximal möglicher Fehler in f

Katharina Morik, Claus Weihs DMV

Methoden des `Lossy Counting` Algorithmus'

- ein neues Element kommt – insert
- am Ende eines Fensters stutzen – compress
- bei Anfrage Ergebnis ausgeben – output

`Lossy Counting` Algorithmus - Insert

Listing 1: Insert-Methode des `Lossy Counting` Algorithmus

```
1  Insert( Element e, int count ) {
2    if( D.contains(e) )
3      D.get(e).f += count
4    else
5      D.put( new Entry(e,1,b_current-1) )
6  }
```


`Lossy Counting` Algorithmus - Compress

- Der Algorithmus stutzt die Datenstruktur am Ende eines Fensters.
- Die Compress-Methode wird also aufgerufen, wenn gilt: $N \equiv 0 \bmod w$

Listing 2: Compress-Methode des `Lossy Counting` Algorithmus

```
7  Compress() {
8    for( Entry entry : D ) {
9      if( entry.f + entry.Δ ≤ b_current )
10       D.remove(entry)
11   }
12 }
```


`Lossy Counting` Algorithmus - Output

! Die gesammelten Daten sind immerzu vorhanden ($\mathcal{D}$). Erst wenn der Benutzer die häufigen Mengen erfragt, muss er den Schwellwert Φ angeben.

Listing 3: Output-Methode des `Lossy Counting` Algorithmus

```
13 Output( double Φ ) {
14   for( Entry entry : D ) {
15     if( entry.f ≥ (Φ − ε)N )
16       result.add( entry )
17   }
18   return result
19 }
```

Wieso funktioniert der Algorithmus? Löschen

- Ein Element wird gelöscht, wenn ...
  - $f + \Delta \leq b_{current}$
- d.h. für ein nichtgelöschtes Element ...
  - Beim Einfügen wird Δ auf das aktuelle Fenster $b_{current} - 1$ gesetzt und bleibt fest.
  - $b_{current}$ erhöht sich jedes Fenster um 1.
  - Wenn ein Element also seine Häufigkeit f nicht mindestens um 1 erhöht in jedem Fenster, dann wird es gelöscht.
  - Es muss also in jedem Fenster mindestens einmal auftauchen, seit es eingefügt wurde, damit es noch in $\mathcal{D}$ vorhanden ist.

Wieso funktioniert der Algorithmus? Einfügen

- Ein Element wird im Fenster $b_{current}$ eingefügt ...
- d.h. es hätte schon vorher eingefügt und wieder gelöscht werden können.
- Die Häufigkeit war aber höchstens $b_{current}$, da wir es sonst ja nicht gelöscht hätten!
- Da $b_{current} = \lceil \frac{N}{w} \rceil$ und $w = \lceil \frac{1}{\epsilon} \rceil$ ist unsere Häufigkeit f höchstens ϵN zu klein.
- Dies führt zu unseren Garantien.

Hierarchische häufige Mengen in Datenströmen

Finding Hierarchical Heavy Hitters in Streaming Data

- Heavy Hitters

  One that is predominant, as in influence or power - thefreedictionary.com

  1-elementige häufige Mengen
- Hierarchical Heavy Hitters
  Häufige Mengen in verschiedenen Hierarchiestufen
- Streaming Data
  Kontinuierlicher Strom an Daten

Wofür braucht man das?

Frequent Itemsets auf einem Datenstrom

- Netzwerkanalyse
  - Welche Benutzergruppen greifen auf welche Web-Seite häufig zu? Benutzergruppen nach gemeinsamem Host-Rechner, z.B. de.tu-dortmund oder de.tu-dortmund.cs oder de.tu-dortmund.cs.ls8, also: gemeinsames Präfix der IP-Adresse.
- Attack Monitoring
  - Denial of Service Attacks fluten einen Rechner mit Paketen, die eine Sitzung eröffnen (SYN), ohne eine Bestätigung der Sitzung zu schicken (SYN-ACK). Dadurch verbrauchen sie die Ressourcen des gefluteten Rechners. Zählt man die SYN und SYN-ACK Pakete findet man den Angriff. Zählt man die Zugriffe des Absenders, findet man den Angreifer.

Netzwerkanalyse

- Auswertung von Netzwerkpaketen
- IP-Adresse: 66.241.243.111
  - IP-Adressen sind von Natur aus hierarchisch in Teilnetze gegliedert
  - IP-Pakete sind multidimensional
    - Absenderadresse und -port
    - Empfängeradresse und -port

technische universität dortmund
LS 8 Informatik
Computergestützte Statistik
Technische Universität Dortmund
Frequent Itemsets über Datenströmen Finding Hierarchical Heavy Hitters in Streaming Data Problem Definition Online Algorithme

Netzwerkanalyse

- Analyse von IP-Paketen und ihren Teilnetzen hilft beim intelligenten Routing
- Planen von Netzwerken anhand des IP Flusses
- Bestimmung von Servern für gespiegelte Seiten
- Erkennen von Attacken

technische universität dortmund
LS 8 Informatik
Computergestützte Statistik
Technische Universität Dortmund
Frequent Itemsets über Datenströmen Finding Hierarchical Heavy Hitters in Streaming Data Problem Definition Online Algorithme

Was müssen wir wissen?

- Erst einmal müssen wir Heavy Hitters kennenlernen,
- dann Hierarchical Heavy Hitters und
- schließlich Hierarchical Heavy Hitters auf Datenströmen.

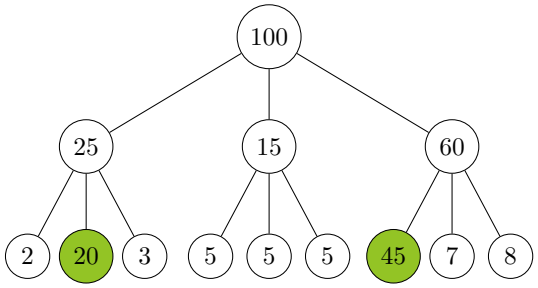
Wir beginnen also statisch, als hätten wir eine Datenbank, bevor wir dynamisch zählen.

technische universität dortmund
LS 8 Informatik
Computergestützte Statistik
Technische Universität Dortmund
Frequent Itemsets über Datenströmen Finding Hierarchical Heavy Hitters in Streaming Data Problem Definition Online Algorithme

Hierarchical Heavy Hitters (HHH) – statisches Beispiel

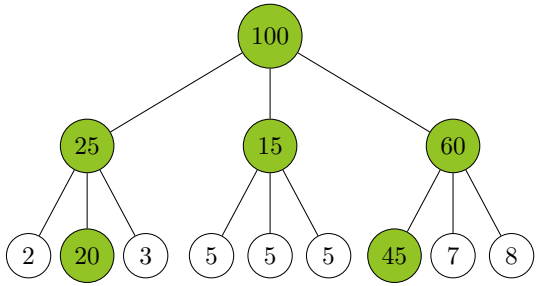
- $N = 100$ Beispiele über eine simple Hierarchie und eine Dimension
- Finde alle HHH mit einer $freq$ von ΦN bei einem Schwellwert Φ zwischen 0 und 1
- Hier: $\Phi = 0.1$

Leaf Heavy Hitters



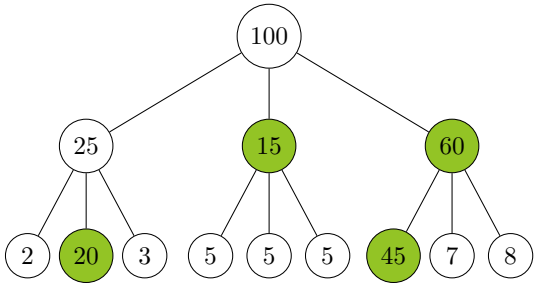
- Traditionelle Heavy Hitter Definition: die Blattknoten müssen häufiger als der Schwellwert sein.
- Keine Aussage über die Hierarchie!
- Was tun?

All Heavy Hitters



- Addieren der Unterknoten ergibt keine gute Hierarchie!
- Ist ein Knoten ein HHH, weil
  - . . . einer seiner Knoten häufig ist oder
  - . . . die Summe aller seiner Knoten?

Hierarchical Heavy Hitters (HHH)

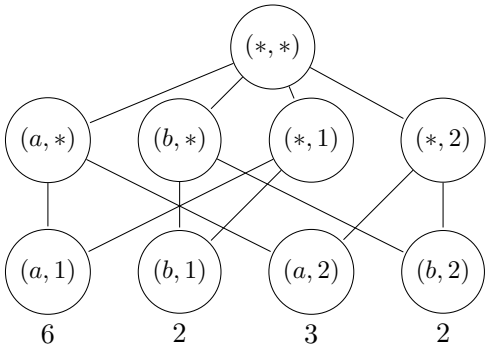


- Heavy Hitters (häufige Blattknoten) sind auch HHH
- Innere Knoten sind HHH, wenn die Summe seiner Kinder den Schwellwert überschreitet
- **und** die Kinder kein HHH sind und keinen anderen HHH Vorgängerknoten haben.

Mehrdimensionale Hierarchical Heavy Hitters (HHH)

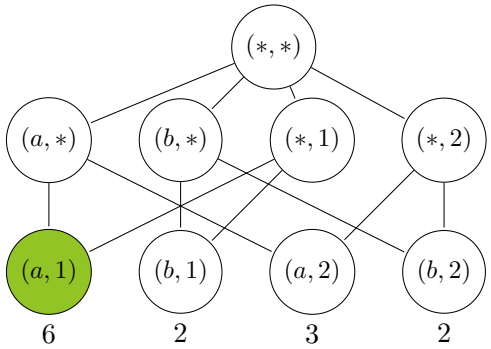
- In der Praxis brauchen wir mehr als eine Dimension
- Ein einfaches Beispiel mit zwei Attributen
- Das erste Attribut kann die Werte a und b haben,
- das zweite 1 und 2.

Mehrdimensionale Hierarchical Heavy Hitters (HHH)



- $N = 13$
- Häufigkeiten der Blätter stehen unter dem Knoten

Leaf Heavy Hitter mit $\Phi = 0.35$

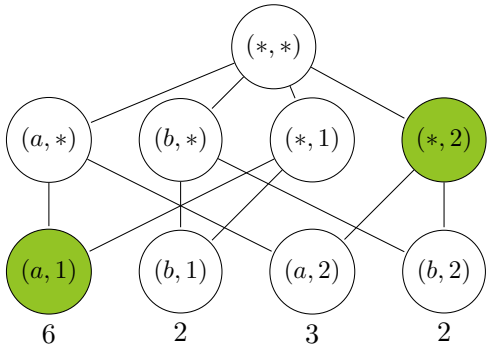


- Mit $\Phi = 0.35$ brauchen wir die Häufigkeit 5
- Hier also nur $(a, 1)$

`overlap`- oder `split`-case

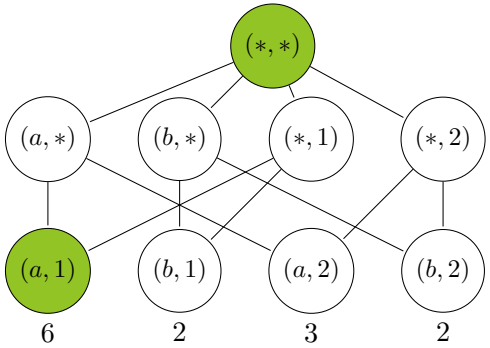
- Im eindimensionalen Fall hatte jeder Knoten nur einen Elternknoten
- Somit wurde die Häufigkeit einfach nach oben propagiert, wenn es sich um keinen HHH handelt.
- Bei mehreren Elternknoten, haben wir zwei mögliche Fälle:
  - `overlap`-case: die Häufigkeit wird zu allen Elternknoten propagiert
  - `split`-case: die Häufigkeit wird aufgeteilt, z.B. 50:50

HHH mit $\Phi = 0.35$ und `overlap`-case



- $count(*, 2) = 5 \rightarrow$ `HHH`
- $count(*, 1) = 2$
- $count(b, *) = 4$
- $count(a, *) = 3$
- $count(*, *) = 2$, da $(a, 1)$ als HHH nicht gezählt wird und $(a, 2)$, $(b, 2)$ auch nicht, da sie Nachfahren vom HHH $(*, 2)$ sind

HHH mit $\Phi = 0.35$ und `split`-case



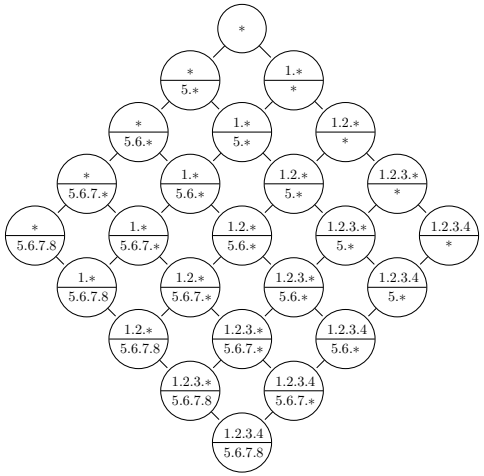
- $count(*,2) = 5/2 = 2.5$
- $count(*,1) = 2/2 = 1$
- $count(b,*) = 4/2 = 2$
- $count(a,*) = 3/2 = 1.5$
- $count(*,*) = 7 \rightarrow$ HHH

Notation

- Wir orientieren uns am Beispiel von IP-Adressen $(1.2.3.4, 5.6.7.8)$ Zwei Attribute:
  - Absenderadresse - 1.2.3.4
  - Empfängeradresse - 5.6.7.8

	Erklärung	Beispiel $(1.2.3.4, 5.6.7.8)$
N	Anzahl der Tupel	$N = 1$
d	Dimensionalität eines Tupels	$d = 2$
h_i	Hierarchietiefe am $i$ten Attribut	$h_1 = h_2 = 4$
$par(e,i)$	Generalisierung am $i$ten Attribut	$par((1.2.3.4, 5.6.7.8), 2) = (1.2.3.4, 5.6.7.*)$
$p \prec q$	p ist generalisierbar zu q	$(*, 5.6.7.8) \prec (*, 5.6.7.*)$
$p \preceq q$	$(p \prec q) \vee (p = q)$	$(*, 5.6.7.8) \preceq (*, 5.6.7.8)$
$e \preceq P$	$\exists p \in P : e \preceq p$	$(1.2.*) \preceq \{(2.*), (1.*)\}$

Halbordnung über der Hierarchie



- Die Hierarchie bildet eine Halbordnung - hier induziert von unserem Beispiel.
- Diese Halbordnung ist ein Produkt der eindimensionalen Hierarchien.
- Gesamtanzahl der Knoten in der Halbordnung:

$$H = \prod_{i=1}^{d} (h_i + 1)$$

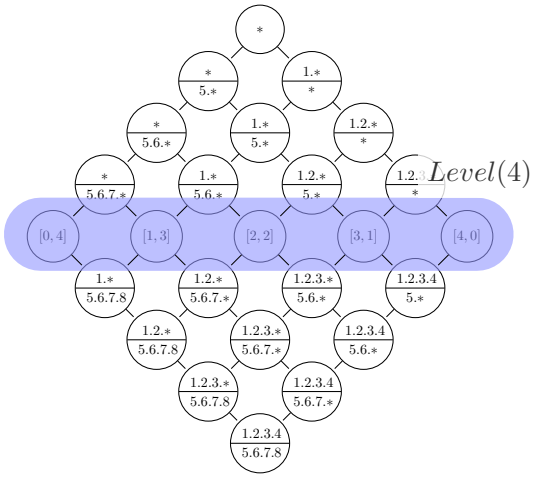
Label in der Halbordnung

- Bestimmte Punkte in der Halbordnung sprechen wir mit einem Label an. So drücken wir die Generalisierung aus, die an diesem Punkt besteht.
  An wie vielen Stellen haben wir noch den konkreten Eintrag und nicht zu $*$ generalisiert?
- Das Label ist ein d-dimensionaler Vektor $[a_1, \ldots, a_d]$, dessen $i$ter Eintrag eine positive Zahl nicht größer als h_i (Anzahl vorausgehender Stellen) ist.

Notationen

	Erklärung	Beispiel $(1.2.3.4, 5.6.7.8)$
$[a_1, \ldots, a_d]$	Knoten in der Halbordnung, gelabelt anhand seiner Generalisierung	$(1.2.3.4, 5.6.7.8) \in [4, 4]$ $(*, 5.6.7.*) \in [0, 3]$
$Level(i)$	Menge aller Elemente, deren Summe der Komponenten von $[a_1, \ldots, a_d]$ i beträgt	$Level(8) = \{[4, 4]\}$ $Level(0) = \{[0, 0]\}$ $Level(5) = \{[1, 4], [2, 3], [3, 2], [4, 1]\}$

$Level$



- Label x und y sind vergleichbar, wenn x für jedes a_j größer oder gleich dem a_j von y ist
- Label in einem $Level(i)$ sind nicht vergleichbar.
- $Level$ reichen von 0 bis h.

`GeneralizeTo` und Antikette

	Erklärung	Beispiel
`GeneralizeTo`$(e, [a_1, \ldots, a_d])$	generalisiert ein Element e zum passenden Label $[a_1, \ldots, a_d]$	`GeneralizeTo`$((1.2.3.4, 5.6.7.8), [0, 3]) = (*, 5.6.7.*)$
Antikette	Eine Menge von Elementen in einer Halbordnung, die nicht miteinander vergleichbar sind	$Level(4)$

Heavy Hitters

Definition (Heavy Hitter)

Gegeben eine Multimenge S, $\mid S \mid = N$, und ein Schwellwert Φ. Ein Heavy Hitter (HH) ist ein Element $e \in S$ dessen Häufigkeit f_e nicht kleiner ist als $\lfloor \Phi N \rfloor$:

$$\mathcal{HH} = \{e | f_e \geq \lfloor \Phi N \rfloor\}$$

- Nach Definition gibt es nicht mehr als $\frac{1}{\Phi}$ HH.
- Auf einer Datenbank wird das Problem exakt mit folgendem SQL-statement gelöst:

Listing 4: SQL-Query für das Heavy Hitter Problem

```
1 SELECT S.elem, COUNT(*)
2 FROM S
3 GROUP BY S.elem
4 HAVING COUNT(*) >= ΦN
```

- Problem jetzt bei Data Streams und begrenztem Platz.
- Behandelt in:

Cormode and Muthukrishnan 2003
What's hot and what's not: Tracking most frequent items dynamically.
Proceedings of the ACM Conference on Principles of Database Systems, 296-306

Hierarchical Heavy Hitters über eine Dimension

Definition (Hierarchical Heavy Hitter)

Gegeben eine Menge S, $\mid S \mid = N$, mit der hierarchischen Domäne D der Tiefe h und ein Schwellwert Φ. Dann definieren wir Hierarchical Heavy Hitters $\mathcal{HHH}$ von S induktiv über:

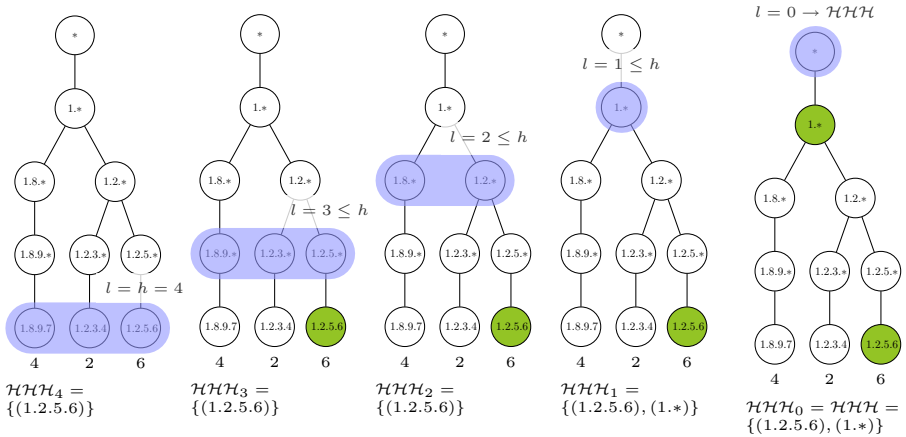
- Die HHH auf Level h ($\mathcal{HHH}_h$) sind die Heavy Hitters von S.
- Für ein Präfix p vom $Level(l), 0 \leq l < h$ ist die Häufigkeit $F_p = \sum f(e) : (e \in S) \wedge (e \preceq p) \wedge (e \notin \mathcal{HHH}_{l+1})$

$$\mathcal{HHH}_l = \mathcal{HHH}_{l+1} \cup \{p | (p \in Level(l)) \wedge (F_p \geq \Phi N)\}$$

- $\mathcal{HHH} = \mathcal{HHH}_h$

Beispiel

Ein Beispiel zum induktiven Vorgehen mit $\Phi N = 5$



Jetzt kennen wir die statischen (Hierarchical) Heavy Hitters!

Jetzt kennen wir

- Heavy Hitters
- Hierarchical Heavy Hitters
  - overlap-Propagierung an alle Elternknoten
  - split-Propagierung anteilig an Elternknoten

und wie man sie berechnen könnte, wenn sie in einer Datenbank statisch vorhanden wären, wir genügend Platz hätten.

HHH in Datenströmen

- Unsere Aufgabe ist es, HHH und ihre Häufigkeit im Datenstrom zu ermitteln.
- Dies ist nicht möglich, ohne Platz linear zur Eingabe-Größe.
- Wir wollen aber dynamisch zählen! Dabei können wir nun nicht die Datenbank verdoppeln.
- Also beschäftigen wir uns mit folgendem Annährungsproblem:

HHH-Problem

Definition (HHH-Problem)

- Gegeben ein Datenstrom S mit N bisher gesehenen Items der hierarchischen Domäne D, ein Schwellwert $\Phi \in [0, 1]$, ein Fehler $\epsilon \in [0, \Phi]$,
- finde alle Präfixe $p \in D$ und die dazugehörigen abgeschätzen Häufigkeitsschranken f_{min} und f_{max}, so dass:

Accuracy: Die wahre Häufigkeit $f^*(p)$ ist angenähert
$f_{min}(p) \leq f^*(p) \leq f_{max}(p)$ und
$f_{max}(p) - f_{min}(p) \leq \epsilon N$

Coverage: Alle nicht gefundenen Präfixe q sind auch nicht häufig
$\Phi N > \sum f_e^* : (e \preceq q) \wedge (e \not\preceq P)$

Güte der HHH-Problemlösungen

- Definition sagt nur etwas über die Korrektheit und nicht über die Güte einer Lösung aus.
- Z.B. erfüllen “Leaf Heavy Hitters” und “All Heavy Hitters” die Bedingungen.
- Die Güte einer Lösung ist hier, dass sie wenig Speicherplatz braucht.
  - Wir wollen keine überflüssigen Angaben in der Lösung haben wie bei “All Heavy Hitters”.
  - Wir wollen Speicher sparen.
- Wir berurteilen die korrekten Lösungen unserer Algorithmen nach dem Platz, den sie brauchen.

Minimalität von HHH-Problemlösungen

Lemma (1)

Für eine Dimension gilt, dass $\mathcal{HHH}$ unter den Mengen, die die **Coverage***-Bedingung erfüllen, minimal ist.*

Beweis.

- $\mathcal{HHH}$ erfüllt die ***Coverage***-Bedingung wegen F_p
- Sei X eine Menge, die ebenfalls die ***Coverage***-Bedingung erfüllt, aber kleiner ist als $\mathcal{HHH}$.
- Sei nun p ein Präfix mit dem tiefsten Level l aus der symmetrischen Differenz von X und $\mathcal{HHH}$.

Fortsetzung des Beweises.

- Es gibt zwei Fälle
  1. $p \in \mathcal{HHH} \setminus X$
     - X und $\mathcal{HHH}$ stimmen überein in allen Level größer als l
     - Da $p \in \mathcal{HHH}$ gilt ja $F_p \geq \Phi N$
     - Dies ist ein Widerspruch zur ***Coverage***-Bedingung in X
  2. $p \in X \setminus \mathcal{HHH}$
     - X und $\mathcal{HHH}$ stimmen in allen Leveln größer als l überein
     - Wegen F_p erfüllt $\mathcal{HHH}$ die ***Coverage***-Bedingung
     - Ersetze p durch $par(p)$ ohne die ***Coverage***-Bedingung zu verletzen, da $\mathcal{HHH}$ sie erfüllt.
     - Dies verkleinert X höchstens, und zwar wenn $par(p) \in X$.
     - Wiederholt man dies, ist die symmetrische Differenz irgendwann leer, d.h. X und $\mathcal{HHH}$ sind identisch.
     - Da wir $|X|$ höchstens verkleinern, war die Annahme falsch, dass $|X| < |\mathcal{HHH}|$.

□

HHH-Problemlösungen

- Was bringt uns dieses Lemma?
- Bei der Auswertung unserer Lösungen können wir sie nun anhand der Größe vergleichen!
- Dabei benutzen wir die Größe der offline berechneten $\mathcal{HHH}$.

Hierarchical Heavy Hitters mit `overlap`-Regel

Definition (Hierarchical Heavy Hitters mit `overlap`-Regel)

Sei S eine Menge von Elementen e mit den Häufigkeiten f_e, $L = \sum_i h_i$ und Schwellwert $0 < \Phi < 1$. Dann sind die HHH folgendermaßen induktiv definiert:

- $\mathcal{HHH}_L$ beinhaltet alle Heavy Hitter des Levels L, also alle $e \in S$ mit $f_e \geq \Phi N$.
- Für ein Element p auf Level $l < L$ ist die Häufigkeit $f_p^l = \sum f_e : (e \in S) \wedge (e \preceq p) \wedge (e \not\preceq \mathcal{HHH}_{l+1})$. Damit ist $\mathcal{HHH}_l$ folgendermaßen definiert:

$$\mathcal{HHH}_l = \mathcal{HHH}_{l+1} \cup \{p : (p \in Level(l)) \wedge (f_p^l \geq \Phi N)\}$$

- $\mathcal{HHH} = \mathcal{HHH}_0$

`Lossy Counting` Algorithmus

- Die vorgestellten Algorithmen basieren teilweise auf den `Lossy Counting` Algorithmus.
- Dieser zählt Häufigkeiten über einen Strom.

Manku and Motwani 2002
Approximate Frequency Counts over Data Streams.
Proceedings of the 28th VLDB Conference Hong Kong, China, 2002

Naiver Algorithmus

- Zunächst werden alle Hierarchical Heavy Hitters gesucht.
- Für alle Labels in der Halbordnung läuft dann eine Instanz des `Lossy Counting` Algorithmus.
- Wenn ein neues Ereignis auftaucht, berechnen wir alle Generalisierungen und übergeben sie den passenden Labeln und deren Algorithmen.
- Da wir nur H Knoten in der Halbordnung haben, brauchen wir nur $O(\frac{H}{\epsilon}\log(\epsilon N))$ Platz!
- Jetzt geben wir alle Heavy Hitters als Hierarchical Heavy Hitters aus.

Naiver Algorithmus

- Qualität der Lösung ist zwar nicht gut (es sind zuviele), aber die `accuracy`- und `coverage`-Bedingungen sind eingehalten

  Accuracy : $f_{min}(p) \leq f^*(p) \leq f_{max}(p)$, wobei $f^*(p)$ die wahre Häufigkeit von $p \in S$ ist. Außerdem: $f_{max}(p) - f_{min}(p) \leq \epsilon N$
  - $f_{min} = f$ die, die wir zählen, sind auf jeden Fall da
  - $f_{max} = f + \epsilon N$, da wir höchstens $\epsilon N = \Delta$ nicht zählen

  Coverage : $\Phi N > \sum f_e^* : (e \preceq q) \wedge (e \not\preceq P)$
  - Ist erfüllt, da wir keine “false negatives” haben.

Full Ancestry Algorithmus

- benutzt den `Lossy Counting`-Algorithmus als Basis,
- behandelt aber nicht jeden Knoten in der Halbordnung unabhängig.
- Als Datenstruktur wird ein “Trie” (Ltree) Präfixbaum $\mathcal{T}$ verwendet.
- Bemerkung: Wir sind gerade bei eindimensionalen Daten

`Full Ancestry` Algorithmus

Der Full Ancestry Algorithmus besteht aus drei Prozeduren:

- `insert`
- `compress`
- `output`

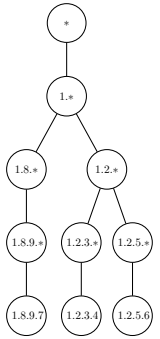
Es gibt (wieder) die Eigenschaften

- `g` Zähler der Häufigkeit
- Δ maximal möglicher Fehler
- `hasChild (e)` Kinderknoten von e
- `par(e)` Elternknoten von e

Full Ancestry Algorithmus - Insert

Listing 5: Insert-Methode des Full Ancestry Algorithmus

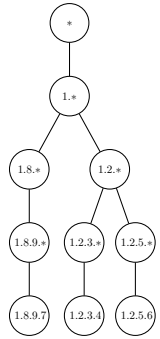
```
1  Insert( Element e, int count ) {
2    if( T.contains(e) )
3      T.get(e).g += count
4    else
5      T.put(e)
6      T.get(e).g = count
7      if( e ≠ '*' )
8        Insert( par(e), 0)
9        T.get(e).m = T.get(par(e)).m
10       T.get(e).Δ = T.get(par(e)).m
11     else
12       T.get(e).m = b_current−1
13       T.get(e).Δ = b_current−1
14 }
```



Full Ancestry Algorithmus - Compress

Listing 6: Compress-Methode des Full Ancestry Algorithmus

```
15 Compress() {
16   for( Element e : T in postorder) {
17     t_e = T.get(e)
18     if( ¬T.hasChild(e) ∧
19           t_e.g + t_e.Δ ≤ b_current ) {
20
21       t_par(e) = T.getParent(e)
22       t_par(e).g += T.get(e).g
23       t_par(e).m += max(t_e.g+t_e.Δ, t_par(e).m)
24       T.delete(e)
25     }
26   }
27 }
```



Full Ancestry Algorithmus - Compress

- Die Compress-Methode stutzt also die Blätter oder Äste, die nicht häufig genug sind ($g + \Delta \leq b_{current}$ - vergleiche Lossy Counting)
- Die gestutzte Grenze wird "*Fringe*" genannt.
- Unter dem "*Fringe*" kann es keine HHH geben.
- Über dem "*Fringe*" zählen wir richtig, da die Häufigkeit g nach oben propagiert wird.

Full Ancestry Algorithmus - m

- Δ hat eine ähnliche Aufgabe, wie bei Lossy Counting ...
- ... aber wofür brauchen wir m?
  - Naiv gesehen gar nicht - man kann es auch weglassen.
  - Beim Einsetzen von einem neuen Element, könnten wir Δ auf $b_{current}$ setzen (Lossy Counting) –
  - Wenn wir aber $\Delta = m_{par(e)}$ setzen und $m_{par(e)} = max_{d \in \textit{gelöschte Kinder}}(g_d + \Delta_d)$ verbessern wir die Schranken.

`Full Ancestry` Algorithmus - m

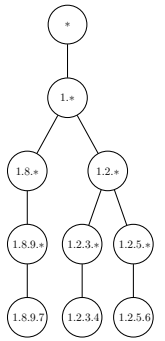
Wieso verbessern wir die Schranke?

- Wegen $m_{par(e)} = max(m_{par(e)}, g_e + \Delta_e)$ kann ein gelöschtes, aber wiedereingefügtes Element nie häufiger vorkommen als $m_{par(e)}$.
- Da aber zwischendurch Zeit vergangen ist, verbessern wir die Schranke gegenüber $b_{current}$.

`Full Ancestry` Algorithmus - Output

Listing 7: Output-Methode des `Full Ancestry` Algorithmus

```
28 Output( double Φ) {
29   for( Element e : T) {
30     T.get(e).F = 0
31     T.get(e).f = 0
32   }
33   for( Element e : T in postorder) {
34     t_e = T.get(e)
35     if( t_e.g + t_e.Δ + t_e.F > ΦN ) {
36       print(e, t_e.f + t_e.g, t_e.f + t_e.g + t_e.Δ)
37     } else {
38       T.getParent(e).F += t_e.F + t_e.g
39     }
40     T.getParent(e).f += t_e.f + t_e.g
41   }
42 }
```



`Full Ancestry` Algorithmus - Lösung für das `HHH`-Problem?

Accuracy

- $f_{min} = f + g = \sum_{e \preceq p} g_e$
- $f_{max} = f_{min} + \Delta$
- Damit haben wir die gleiche Situation wie beim `Lossy Counting` Algorithmus.
- Und da Δ von m abhängt, sind wir mit $f_{max} - f_{min}$ meistens kleiner als ϵN.

Coverage

- Wir geben ja nur die Items nicht aus, die den Schwellwert nicht überschreiten.
- Die, die gar nicht in der Datenstruktur sind, könnten ihn auch nicht überschreiten.

`Full Ancestry` Algorithmus - Platz

Lemma (Platzbedarf von `Lossy Counting`)

Der `Full Ancestry` Algorithmus speichert höchstens $O(\frac{H}{\epsilon}\log(\epsilon N))$ *Einträge.*

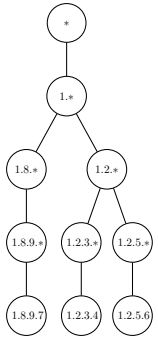
Partial Ancestry Algorithmus

- Der Full Ancestry Algorithmus braucht Platz für innere Knoten, auch wenn diese eine niedrige Häufigkeit haben.
- Für zusätzliche Informationen wird eigentlich nur ein Vorgänger benötigt.
- Wichtig ist nur die Grenze ("*Fringe*").
- Dazu werden nur noch die nächsten Vorfahren für die Speicherung von m benötigt.

Partial Ancestry Algorithmus - Insert

Listing 8: Insert-Methode des Partial Ancestry Algorithmus

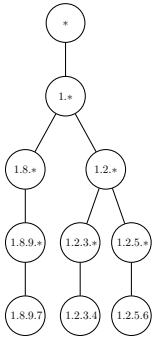
```
1  Insert( Element e, int count ) {
2    if( T.contains(e) )
3      T.get(e).g += count
4    else
5      T.put(e)
6      T.get(e).g = count
7      if( T.ancestorExist(e) )
8        T.get(e).m = T.get(anc(e)).m
9        T.get(e).Δ = T.get(anc(e)).m
10     else
11       T.get(e).m = b_current−1
12       T.get(e).Δ = b_current−1
13 }
```



Partial Ancestry Algorithmus - Compress

Listing 9: Compress-Methode des Partial Ancestry Algorithmus

```
14 Compress() {
15   for( Element e : T in postorder) {
16     t_e = T.get(e)
17     if( t_e.g + t_e.Δ ≤ b_current ) {
18       if( e ≠' *') {
19         Insert( par(e), t_e.g)
20         t_par(e) = T.getParent(e)
21
22         t_par(e).m += max(t_e.g+t_e.Δ, t_par(e).m)
23         T.delete(e)
24       }
25     }
26   }
27 }
```

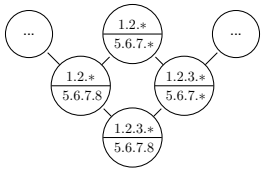


Wirklich weniger Platz?

- Prinzipiell ja, da potenziell weniger Präfixe eingefügt werden.
- Dadurch, dass sie nicht schon am Anfang eingefügt werden, ist der Δ Wert höher, wenn sie später eingefügt werden, weil ein Knoten gelöscht wurde.
- Dadurch sind sie aber potenziell schwerer zu löschen.
- Die Experimente zeigten, dass dieser Algorithmus aber tatsächlich weniger Platz braucht.

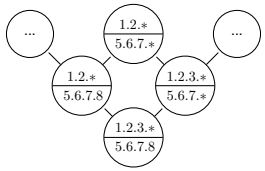
Mehrdimensionale Algorithmen

- Wir bauen auf den `Full`- bzw. `Partial Ancestry` auf
- Dabei ist `Insert`-Methode identisch zu `Full`- bzw. `Partial Ancestry`
- Der Unterschied ist beim Zählen, da wir z.B. beim Löschen die Häufigkeit nicht nur an einen Elternknoten propagieren.
- Wie schon gesagt behandeln wir bei diesem Thema den `overlap`-Fall: da zählen wir schnell zuviel!
- Dem wird entgegengewirkt mit dem sogenannten Inklusions-Exklusions Prinzip.



Inklusions-Exklusions Prinzip

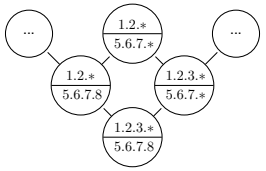
- Alternierend mit der “Generation” wird die Häufigkeit dazu addiert (Eltern, Urgroßeltern) oder subtrahiert (Großeltern)
- bis wir den Knoten erreichen, an dem alle Attribute genau einmal generalisiert wurden.
- Dies ist eine Unterhalbordnung in Diamantform.



Inklusions-Exklusions Prinzip - Realisierung

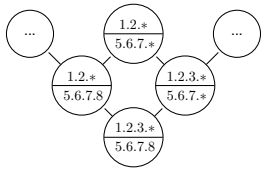
- Angenommen, e ist ein Item, das an jedem Attribut noch generalisierbar ist.
- Dann sei $a = par(par(...(e, 1), 2), ...d)$
- e und a induzieren eine Unterhalbordnung mit 2^d Präfixen (items) p ($e \preceq p \preceq a$).
- Für alle p gibt es einen Bit-String, der an der $i$ten Stelle 1 ist, wenn das $i$te Attribut generalisiert wurde:

$$e = 0^d, a = 1^d, par(e, 1) = 10^{d-1}$$



Inklusions-Exklusions Prinzip - Realisierung

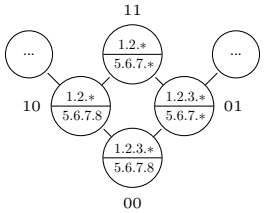
- Die Gewichtsfunktion wt, angewandt auf einen Bit-String, ergibt die Anzahl der 1en
- $wt(b)\%2 = 0 \rightarrow$ addiere hinzu
- $wt(b)\%2 = 1 \rightarrow$ subtrahiere
- Im Algorithmus ist das $parcount$.



Multidimensionaler `Partial Ancestry` Algorithmus - Compress

Listing 10: Compress-Methode des multidimensionalen `Partial Ancestry` Algorithmus

```
1  Compress() {
2    for( l = L to 0 ) {
3      for( t_e : l.nodes ) {
4        if( |t_e.g| + t_e.Δ ≤ b_current ) {
5          for( j = 1 to 2^d − 1 ) {
6            p = t_e
7            parcount = 0
8            for( i = 1 to d ) {
9              if( bit(i, j) = 1 ) {
10               p = par(p, i)
11               parcount += 1
12             }
13           }
14           if( T.contains(p) ) {
15             factor = 2 × bit(1, parcount) − 1/2
16             insert(p, g_e × factor)
17             if( parcount = 1 )
18               T.get(p).m = max(T.get(p).m, |t_e.g| + t_e.Δ)
19           }
20         }
21       }
22       T.delete(t_e)
23 3×}
```



Accuracy

- Cormode et al. geben folgendes an:

$$f_{min} = f(p) - \Delta_p \textbf{ und } f_{max} = f(p) + \Delta_p \textbf{ mit } f(p) = \sum_{e \preceq p} g_e$$

- $f_{max} - f_{min} = 2\Delta_p$?
  - Passiert, da nun Knoten mit negativer Häufigkeit gelöscht werden können.
  - Accuracy-Bedingung ist aber dann nicht eingehalten!
- f_{max} bleibt wie gewohnt gleich, da wir durch das Inklusion-Exklusion Prinzip nicht zuviel zählen.

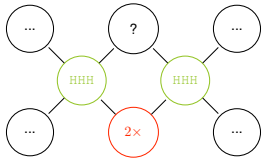
Output und Coverage?

- Reicht $f_{max} \geq \Phi N$ aus?
  - `Coverage`-Bedingung: Ja
  - Qualität der Lösung ähnlich zum naiven Algorithmus!
- Für besseres (kleineres) Ergebnis müssen wir HHH-Vorgänger abziehen
- Sei $H_p = \{h | \neg\exists h' \in P | h \prec h' \prec p\} \subseteq P$ (P =bisherigen HHHs)
- Warum nur von der Stufe davor? Beobachtungen haben gezeigt, dass die Schranken sonst schlechter werden.

Output und Coverage?

$f_{max}(p) - \sum_{h \in H_p} f_{min}(h)$?

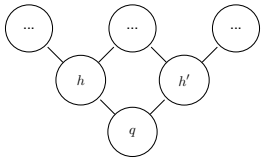
- Passt auch nicht ganz, weil manche Werte doppelt abgezogen werden können: Es kann zwei Knoten geben, die beide HHHs sind und beide einen Knoten als Nachfolger besitzen.
- Also wieder etwas dazu addieren!



Output und Coverage?

- Was dazu addieren? Zunächst ein paar Notationen
- $q = glb(h, h')$ (greatest lower bound), wenn q das Einzige Element ist, dass Folgendes erfüllt

$$\forall p : (q \preceq p) \wedge (p \preceq h) \wedge (p \preceq h') \Rightarrow p = q$$

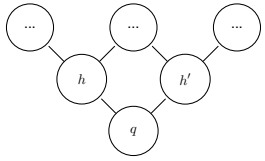


- Dominante Menge von q zu H_p

$$Dom(q, H_p) = \{h \in H_p | q \prec h\}$$

Output und Coverage?

- Mit glb finden wir also alle gemeinsamen Vorgängerknoten in H_p
- und addieren dessen Häufigkeit wieder auf,
- multipliziert mit $|Dom(q, H_p)| - 1$, falls dieser Knoten von mehr als 2 Vorgängerknoten geteilt wird.



$$f_{max}(p) - \sum_{h \in H_p} f_{min}(h) + \sum_{q = glb(h \in H_p, h' \in H_p)} f_{max}(q)(|Dom(q, H_p)| - 1)$$

Output und Coverage?

- Den Output-Code lasse ich mal weg, da er
  - sehr groß und verwirrend ist
  - und genau das oben genannte umsetzt.
- Dass aber die `Coverage`-Bedingung durch diese Anpassungen nicht gebrochen wird, ist klar!

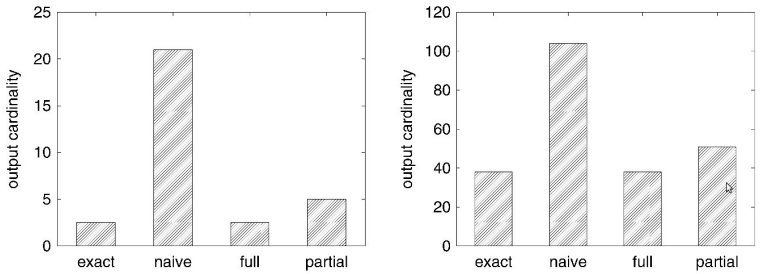
Output und Coverage?

Lemma (Platzbedarf vom mehrdimensionalen `Partial`- bzw. `Full`ancestry Algorithmus)

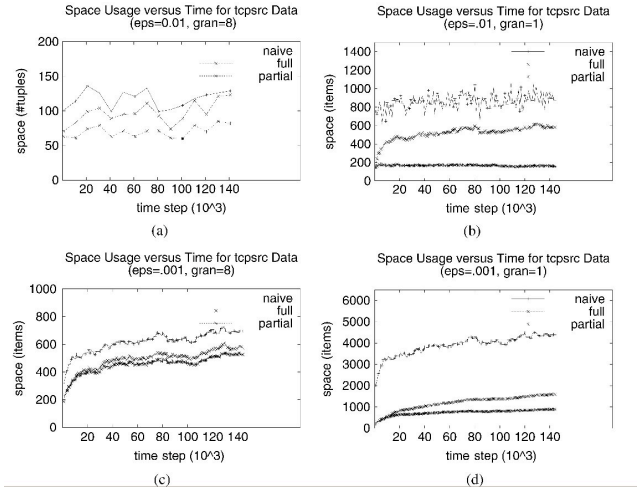
Der mehrdimensionale `Partial`- bzw. `Full`ancestry Algorithmus speichert höchstens $O(\frac{H}{\epsilon} log(\epsilon N))$ Einträge.

Experimente

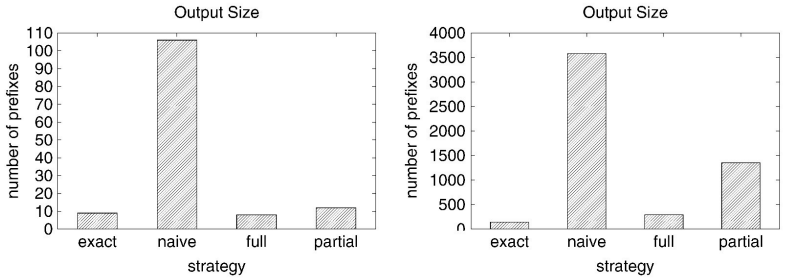
- Alle drei vorgestellten Algorithmen wurden für eine wie auch mehrere Dimensionen verglichen.
- Für die Güte der Lösungen wurde die exakte Größe der offline berechneten `HHH`s benutzt.
- Als Daten gab es echten IP traffic von einem AT&T backbone (400Mbits/s).



(a) $\phi = 0.2, \epsilon = 0.1$ (b) $\phi = 0.02, \epsilon = 0.01$



Algorithmen in mehreren Dimension



(a) $\phi = 0.2, \epsilon = 0.01$ (b) $\phi = 0.02, \epsilon = 0.001$

Was wissen Sie jetzt?

- Sie wissen, was ein Heavy Hitter und ein Hierarchical Heavy Hitter ist.
- Sie wissen, wie das Problem der HHH formal definiert ist und kennen das wichtige Gütekriterium der Minimalität (geringer Platz).
- Sie wissen, wie `lossy counting` für HHH verbessert wurde. Der Algorithmus heißt `Full Ancestry`.
- Sie wissen, dass es algorithmisch auch noch komplizierter wird, wenn mehrdimensionale Daten strömen (`Partial Ancestry`).
- Sie haben ein Beispiel für die Informatiktheorie gesehen: Beweise,
  - der Korrektheit (accuracy) und Vollständigkeit (coverage) der Lösung,
  - wie viel Platz gebraucht wird,
  - wie viel Laufzeit gebraucht wird.