Bloom-Filter

- Bit-Coding, Hash-Coding, Bloom-Filter

Advertisments

- Advertisments geben Aufschluss über die Verfügbarkeit einer Information und beinhalten nicht die Information
  - müssen kleiner als Daten sein
  - häufig in Form von Tags (Freitext-Annotationen)
  - In Kommunikationsnetzen möglichst feste (kleine) Größe

Wie erstellt man Advertisments?

Eine Methode: Interesse gegeben durch Teilmenge M eines Universums U

$$M \subseteq U$$

etwa M = {Temperatur, Luftdruck} U={alle denkbaren Sensoren}

Bit-Coding

- Sei U eine endliche Menge mit k Elementen
- Sei P(U) die Potenzmenge von U (= Menge aller Teilmengen von U)
  - P(U) hat 2^k Elemente
- Durch klassisches Bit-Coding kann man die Menge U mit

$$\log_2 k$$

Bits darstellen.

➔Alle Teilmengen sind mit

$$\mathrm{k} = 2^{\log_2 k}$$

Bits darstellbar

(Intuition: Je ein Bit für Enthaltensein jeden Elements)

Bit-Coding

- Vorteile
  - schnell
  - einfach

- Nachteile
  - viel Speicherplatz (exponentiell)

Hash-Coding

- Bei großen Universen ist man in der Praxis nur an sehr kleinen Teilmengen interessiert
  - Alle denkbaren Dateien ➔ alle Dateien in einer Datenbank
- Sei dann also

$$M \subseteq U, |M| = m \ll k$$

- Sei F ein Bit-Array mit m Zellen von je (k+1) Bits
  - k Bit stellen ein Element dar
  - das eine Bit stellt die Belegung einer Hash-Zelle fest
- Insgesamt kann F dann alle Teilmengen

$$M \subseteq U$$

mit |M| < m darstellen

Mengendarstellung

- Durch welche Operationen wird eine vollständige Mengendarstellung durch eine Datenstruktur F möglich?
  1. Darstellung der leeren Menge (F=$F_\emptyset$ für $M = \emptyset$)
  2. Einfügen eines Elements (F:=Insert(F,u))
  3. Testen eines Elements (Test(F,u))
- Ein Tripel bestehend aus Definitionen für 1..3 stellt Mengen korrekt dar, falls
  - $Test(F_\emptyset, a)$ = falsch für alle $a \in U$
  - $Test(F, a) \Leftrightarrow Insert(F, a)$ wurde angewendet

Hash-Coding

- Fixiere U und eine (injektive) Bit-Codierung $U \to \mathcal{B}^k$
- Für traditionelles Hash-Coding gelten die folgenden Definitionen:

  Die **leere Menge** wird durch F=0 dargestellt

  F:=Insert(F,a)

  1. Erzeuge mit dem Element a „geseedete“ Zufallssequenz
     $$(z_a)\colon \mathbb{N} \to \underline{m} = \{0,1,\dots,m-1\}$$
  2. Durchsuche die durch (z_a) indizierten Zellen, bis das Belegungsbit der Zelle 0 ist
  3. Setze dieses Belegungsbit und speichere die Bitdarstellung von a in den restlichen k Bit.

Test(F,a)

1. Erzeuge mit dem Element a „geseedete“ Zufallssequenz
   $$(z_a)\colon \mathbb{N} \to \underline{m} = \{0,1,\dots,m-1\}$$
2. Durchsuche die durch (z_a) indizierten Zellen
   1. Ist das Belegungsbit 0, so gebe falsch zurück.
   2. Ist das Belegungsbit 1, so prüfe, ob der Rest der Zelle die Bit-Codierung von a ist. In diesem Fall gebe wahr zurück, sonst fahre fort.

Hash-Coding

- Eigenschaften
  - $Test(F_\emptyset, a)$ = falsch für alle $a \in U$
  - $Test(F, a) \Leftrightarrow Insert(F, a)$ wurde angewendet

[Beispiel an der Tafel]

Diskussion

- Hash-Coding:
  - Benötigt $(k+1)m$ Bits
  - kurze Zugriffszeiten auf Elemente
    (je nachdem, wann eingefügt und wie viele Hash-Kollisionen es gibt)
  - moderate Zugriffszeiten auf Nicht-Elemente
    (durchsuche alle Kollisionen, bis eine leere Zelle gefunden wird)
  - Worst-Case [Tafel]
  - Break-Even [Tafel]

Hash-Filter

- Wenn man die Bedingungen über die Darstellung abschwächt, indem man „false-positives“ zulässt, so kann man eine erhebliche Steigerung der Effizienz erreichen.
- Ziel: Kodiere Menge durch Null, Insert und Test, sodass
  - $Test(F_\emptyset, a)$ = falsch für alle $a \in U$
  - $Test(F, a) \Leftarrow Insert(F, a)$ wurde angewendet
- Beschreibt stets eine Obermenge von M.
  - Kann man verwenden, um aufwändige Operationen auf den Elementen zu reduzieren. („Filter“)
  - False-Positives (Check(F,a) = wahr, obwohl Insert(F,a) nie angewendet wurde) erhöhen den Aufwand, führen aber nicht zu einem Fehler.

Bloom-Filter

- Fixiere U und eine (injektive) Bit-Codierung $U \to \mathcal{B}^k$
- Sei $d \geq 1$ fest. Sei $N \geq 1$ fest und F ein Hashfeld von N Bits
- Seien (ϕ_i): $U \to \underline{N} = \{0,1, \ldots N-1\}$ gleichverteilte, paarweise unabhängige Hash-Funktionen

Die **leere Menge** wird durch F=0 dargestellt

F:=Insert(F,a)

1. Setze jeweils das Bit, welches durch

$$\phi_1(a) \ldots \phi_d(a)$$

indiziert wird

Test(F,a)

1. Prüfe alle durch

$$\phi_1(a) \ldots \phi_d(a)$$

indizierten Bits.

   1. Sind alle gesetzt, gebe wahr zurück
   2. Sonst gebe falsch zurück

Bloom-Filter

- Eigenschaften
  - $Test(F_\emptyset, a)$ = falsch für alle $a \in U$
  - $Test(F, a) \Leftarrow Insert(F, a)$ wurde angewendet

[Beispiel an der Tafel]
Insert, Check, False-Positives

Fehler…

- Die entscheidenden Parameter für einen Bloom-Filter sind die Anzahl d der Hash-Funktionen und die Größe N des Hash-Feldes.
- Wenn bereits n Elemente eingefügt wurden, wie groß ist dann die Wahrscheinlichkeit für ein „false-positive“?
  - Ermittle Anteil p der Bits, die noch Null sind.

$$p = \left(1 - \frac{1}{N}\right)^{dn} \approx \mathrm{e}^{-\frac{\mathrm{dn}}{\mathrm{N}}}$$

  - Für ein False-Positive muss man d Stellen treffen, die 1 sind
  - $P(false\ positive\ |n\ inserts) = (1-p)^d$

Optimaler Filter

- Problem: Seien n und N fest. Wie viele Hash-Funktionen (d) sollte man nehmen?
  - Mehr Hash-Funktionen ➔ mehr Einsen ➔mehr False-Positives (INSERT)
  - Mehr Hash-Funktionen ➔Mehr Möglichkeiten, eine Null zu finden ➔ weniger False-Positives (CHECK)
- Optimal?
  - Nehme Darstellung mit Exponentialfunktion, ableiten, Null-Setzen

$$d = \log(2)\left(\frac{N}{n}\right)$$

Weitere Operationen

- **Vereinigung**
  - $M, N \subseteq U$ dann ist $F_M \vee F_N = F_{N \cup M}$
- **Schnitt**
  - $M, N \subseteq U$ dann ist $F_M \wedge F_N \supseteq F_{N \cap M}$
  - leider aber mehr False-Positives
- **Entfernen nicht möglich**
  - könnte andere Elemente „zerstören“, weil mehrfaches Bit-Setzen nicht erkannt wird.
- **Halbieren**
  - Falls, $N = 2^k$so kann man die Größe des Filters halbieren indem man die zwei Hälften per OR kombiniert.

Counting Bloom Filter

- Um das Entfernen von Elementen doch zu ermöglichen, wird der Bloom-Filter erweitert.
- Jede Zelle des Hash-Arrays enthält dann nicht mehr ein Bit, sondern einen kleinen Counter.
  - Entfernen durch Dekrementieren des Counters möglich

Compressed Filter (Mitzenmacher, 2002)

- Kompression von Bloom-Filtern (mit optimaler Hash-Zahl) nicht möglich, weil in diesem Fall jedes Bit mit Wahrscheinlichkeit $p = \frac{1}{2}$ gesetzt ist
- Wenn aber größere Felder genommen werden (k nicht mehr optimal) lässt sich das Ergebnis komprimieren

Array Bits pro Element	16	28	48
Übertragungsbits pro Elt	16	15,84	15,82
Hash-Funktionen	11	4	3
False-Positive Rate	0.000459	0.000314	0,000222

Anwendungen von Bloom-Filtern

- **Wörterbücher**
  Im Originalbeispiel wurden für ein Silbentrennprogramm die Datenbank-Lookups reduziert
- **Datenbanken**
  In verteilten Datenbanken werden Abfragen geteilt, indem die erste Datenbank eine Teilliste erstellt und als Bloom-Filter abstrahiert. So kann eine zweite Datenbank die Abfrage nur noch auf einer Obermenge des Abfrageresultats berechnen.
- **Web-Cache Sharing (Summary Cache)**
  Kooperative (HTTP-)Proxies erstellen und teilen Bloom-Filter über ihren Inhalt. So kann ein Proxy an einen anderen Proxy delegieren, der eine Seite im Cache hat und so unnötige Requests vermeiden.

- **Routing**
  - Resource-Routing
  - P2P-Network
  - Content-Delivery
  - Geographic Routing
  - Loop-Detection
  - Multicast
  - uvm…
- Einiges kommt später …

Aktuelle Nutzung

- **Google BigTable** and **Apache Cassandra** use Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.[5]
- The **Google Chrome** web browser uses a Bloom filter to identify malicious URLs. Any URL is first checked against a local Bloom filter and only upon a hit a full check of the URL is performed.[6]
- The **Squid Web Proxy Cache** uses Bloom filters for cache digests.[7]
- **Bitcoin** uses Bloom filters to verify payments without running a full network node.[8][9]
- The **Venti archival storage system** uses Bloom filters to detect previously stored data.[10]
- The **SPIN model checker** uses Bloom filters to track the reachable state space for large verification problems.[11]
- The **Cascading analytics framework** uses Bloomfilters to speed up asymmetric joins, where one of the joined data sets is significantly larger than the other (often called Bloom join[12] in the database literature).[13]

http://en.wikipedia.org/wiki/Bloom_filter