Bloom Filter

Maximilian C. Zuleger, *Technische Hochschule Ingolstadt*

25. April 2016

Bloom Filter - eine probabilistische Datenstruktur

Dieser Artikel behandelt die Funktionsweise eines Bloom Filters. Dabei wird insbesondere auf die mathematischen Grundlagen des Bloom Filters Wert gelegt und dieses System anhand von beiliegenden Beispielen aufgezeigt. Mit einer speziellen Datenstruktur kann durch dieser Methode die Performance in vielen Anwendungen deutlich gesteigert werden. Heutzutage sind Bloom Filter in mehreren Datenbanken und Browsern integriert und verringert die Wartezeit für Benutzer und Systeme. Durch die minimalistische Implementierung steht der Bloom Filter in zahlreichen Programmiersprachen zur Verfügung und kann bei einer praktischen Anwendung als Modul in der Datenzugriffsschicht eingegliedert werden.

1 Einleitung

Bloom Filter (Abkürzung *Bf*) wurden 1970 von Burton Howard Bloom bei der *Corporation of America* in Cambridge entwickelt. Das System sollte eine performante und effiziente Abfrage implementierten ob ein bestimmtes Element schon in einem bestimmten Datensatz enthalten ist oder nicht (vgl. [1]).

2 Probabilistischer Bloom Filter

Um eine schnelle Abfrage zu garantieren, muss beim *Bf* ein hoher Preis gezahlt werden, der Aufbau als probabilistische Datenstruktur. Aufgrund dieser Struktur kann das System nur zwei Aussagen tätigen.

1. Das Element ist nicht im Datensatz
2. Das Element ist vielleicht im Datensatz

Ein *Bf* kann somit keine sichere Aussage treffen, das ein Element sicher im Datensatz enthalten ist. Mit diesen Vorwissen kann nun die Grundfunktionsweise eines *Bf*s erläutert werden. Ein *Bf* besteht im Grundaufbau aus einem Bitarray, mehreren Hashfunktionen und den einzufügenden Elementen. Zusätzlich kommt noch die Datenbank (*DB*) hinzu, die nicht direkt zum *Bf* gehört. Es soll nun geprüft werden, ob die einzufügenden Elemente sich bereits in der Datenbank befinden. Um die zeitintensive Abfrage gegen die Datenbank zu vermindern, wird nun ein *Bf* in diese Prozedur zwischengeschaltet. Die Abbildung 1 zeigt den Ablauf mit eingeschalteten *Bf*. Der String

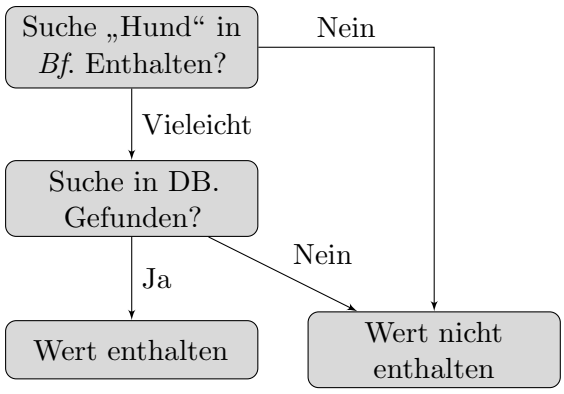


Abbildung 1: *Ablauf mit einem Bloom Filter*

„Hund“ soll validiert werden, ob sich dieser bereits in der Datenbank befindet. Dabei wird zunächst im *Bf* überprüft, ob sich das Element, 1. nicht im Datensatz befindet oder, 2. sich vielleicht in dem Datensatz befindet. Tritt der erste Fall ein, befindet sich das Element nicht in der Datenbank und die Abfrage ist beendet. Tritt der zweite Fall in Kraft, kann durch den *Bf* nur die Aussage getroffen werden, ob sich

das Element vielleicht in der Datenbank befindet. An dieser Stelle kann eine direkte Abfrage gegen die Datenbank getätigt werden, um sicher zu gehen, dass sich das Element auch sicher in dieser befindet. Mit dieser Prozedur kann die Abfrage nun mit einer Vielzahl von Elementen gestartet werden. Wie kann der *Bf* allerdings bestimmen, ob sich ein Element nicht, bzw. vielleicht in der Datenbank befindet?

Zunächst wird ein m-stelliges Array, in diesem Beispiel $m = 10$, initial mit 0=False belegt. Es befinden sich zu diesem Zeitpunkt keine Daten in der Datenbank.

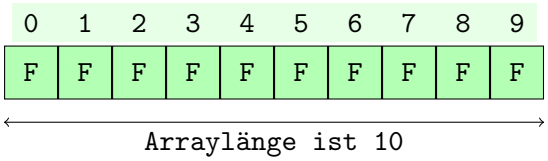
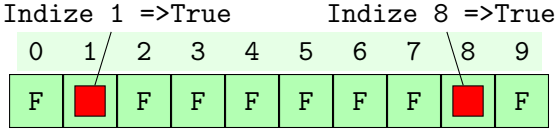
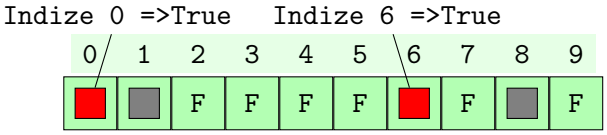


Abbildung 2: *Bloomfilterarray*

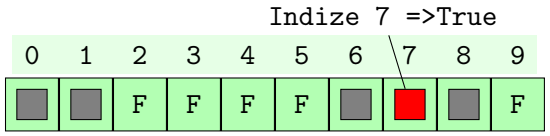
Es soll nun der Datensatz [Max,Julia,Peter,Sandra] in die Datenbank eingetragen werden, wenn sich der Wert nicht bereits in der Datenbank befindet. Um möglichst effizient abzufragen, ob ein Element bereits enthalten ist, werden im *Bf* die eingefügten Elemente mit Hilfe von Hashfunktionen abgebildet. In der Datenbank stehen die Einträge in Klartext. Jeder Bloomfilter kann k Hashfunktionen enthalten, die jeweils auf einen Ganzzahlwert zwischen 0 und der Arraylänge $m-1$ abbilden. An der abgebildeten Stelle wird das Arraybit auf 1=True gesetzt. Zunächst soll das Element „Max“ eingetragen werden. Dafür wird der mit k Hashfunktionen, in diesem Beispiel mit $k = 2$, auf die Stellen des Arrays abgebildet. Für das Element „Max“ bildet die erste Hashfunktion auf den Wert „1“ und die zweite Hashfunktion auf den Wert „8“ ab. An diesen beiden Indizes befindet sich im Array eine 0=False. Der Wert „Max“ kann sich also nicht in der Datenbank befinden. Der Name wird nun in die Datenbank geschrieben und die beiden Indizes auf 1=True gesetzt. Anschließend wird das



Element „Julia“ eingetragen. Der Name wird von den Hashfunktionen auf die Werte 0 und 6 abgebildet. An diesen Indizes im Array befinden sich an beiden Stellen ein 0=False Bit, weshalb sich der Wert in der Datenbank nicht befinden kann. An den beiden Stellen des Arrays wird nun ein 1=True Bit gesetzt und der Name in die Datenbank eingetragen.

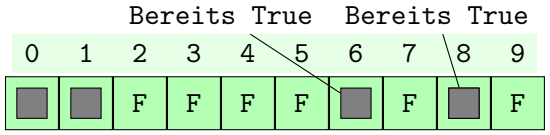


Die gleiche Prozedur geschieht für das Element „Peter“ das durch die Hashfunktionen auf die Indizes 1 und 7 abbildet. An der Stelle 1 des Arrays befindet sich bereits ein Bit mit 1=True, allerdings liegt an der siebten Stelle des Arrays ein 0=False Bit, weshalb das Element „Peter“ sich nicht in der Datenbank befindet. Folgend wird das Element wiederum in die Datenbank, sowie in den *Bf* eingetragen. Zuletzt soll



Datenbank: [Max,Julia,Peter]

das Element „Sandra“ in die Datenbank eingefügt werden. Die Hashfunktionen bilden dieses Element auf die Werte 6 und 8 ab. An diesen beiden Stellen sind allerdings bereits 1=True Bits vorhanden, weshalb der Bloomfilter nun die Aussage macht, dass sich das Element vielleicht in der Datenbank befindet. Es



sollte nun in der Datenbank geprüft werden, ob sich das Element dort schon befindet, oder nicht. Da sich der Name dort noch nicht befindet, wird dieser in die Datenbank eingetragen. Im Bloomfilter bleibt die Besetzung des Arrays wie zuvor und es können weitere Elemente eingefügt werden. Anhand der Funktionsweise können folgende Ereignisse, aufgezeigt durch eine Klassifikation mit einer Wahrheitsmatrix, im *Bf* auftreten (vgl. [5]):

Klassifikation	negative	positive
False	*1	*2

1. Das False-negative Event ist in einer probabilistischen Datenstruktur eine sichere Aussage und kann deshalb nicht auftreten. Ein Element e ist sicher nicht in der Datenstruktur.

2. Das False-positive Event ist in der probabilistischen Datenstruktur eine unsichere Aussage und kann deshalb Auftreten. Obwohl das Element e sich nicht in der Datenbank befindet wird vom *Bf* trotzdem die Aussage getroffen, das es sich vielleicht in der Datenstruktur befindet.

Bei der Implementierung muss also Berücksichtigt werden, ob eine false-positive (Abkürzung *fp*) Ereignis auftritt. Anhand der Größe m des Bitarrays, der Anzahl der voraussichtlich eingefügten Elemente und der Anzahl der Hashfunktionen kann die Wahrscheinlichkeit eines *fp* Ereignisses errechnet werden. Um die Berechnung allerdings genauer zu Begutachten, werden zunächst die Grundlagen von Hashfunktionen im folgenden Unterkapitel besprochen.

2.1 Hashfunktion

Eine Hashfunktion ist eine Abbildung, die eine bestimmte Eingabemenge auf eine kleinere Zielmenge abbildet. Die Eingabemenge stellt dabei die sogenannten Schlüssel und die Zielmenge die Hashwerte dar. Im Fachjargon wird dieser Vorgang auch als „Zerhacken“ (von engl. Hash) bezeichnet. Eine gute Hashfunktion liefert dabei für die Eingabedaten Werte so, dass zwei bzw. viele unterschiedliche Eingaben auch zu unterschiedlichen Ausgabewerten führen. Es wird dabei von einer Gleichverteilung der Hashwerte auf die erwarteten Eingabewerte geredet. Zusätzlich soll

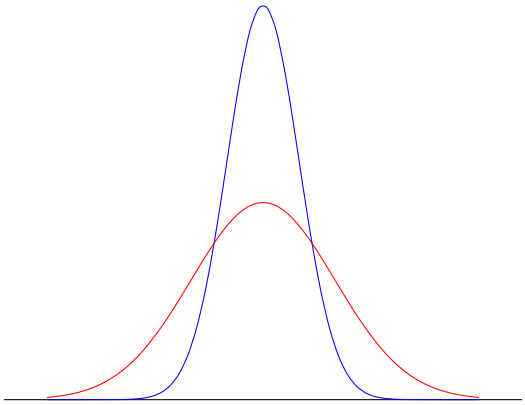
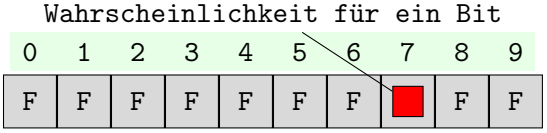


Abbildung 3: *Hashes nähern sich eher der Poisson-Verteilung als der Gleichverteilung an*

der Speicherbedarf des Hashwertes deutlich kleiner sein als jener der Nachricht. Eine weitere Eigenschaft ist das tatsächliche Vorkommen jedes möglichen Hashes, das auch als Surjektivität bezeichnet wird. Ein wichtiger Faktor einer Hashfunktion ist zudem die schnelle und effiziente Berechenbarkeit. Die Prozedur sollte im besten Fall nur wenige Millisekunden dauern. Je länger dieser Vorgang dauert, umso höher wird der Overhead des Bloomfilters, der jeden Eingabewert in einen Hashwert umwandelt. Laut der oben genannten Definition von Hashfunktionen, sind die Abbildungen für unterschiedliche Eingabewerte unterschiedliche Hashes. In der Praxis nähert sich die Verteilung aber eher an eine Poisson-Verteilung (ab $\lambda = 30$ ähnelt Normalverteilung) an, weshalb die anschließende Berechnung vom Bloomfilter nur eine Annäherung ist. Berühmte Hashfunktionen sind u.a. *MD5, SHA-1* oder *SHA-2*. Diese Funktionen haben sich in der heutigen Zeit aufgrund ihrer Zuverlässigkeit bewährt. Damit ein Eingabewert auf einen Indize in einem Array abgebildet werden kann, muss die Hashfunktion auf einen Ganzzahlwert abbilden. Dieser Wert muss zudem innerhalb der Grenzen des Arrays liegen (0 bis $m-1$).

2.2 Bloom Filter Formeln

Unter Annahme der Gleichverteilung gilt:



Die Wahrscheinlichkeit, dass eine Hashfunktion dieses bzw. ein bestimmtes Bit auf 1=True setzt liegt bei: $\frac{1}{m}$. Das Komplementärereignis, die Wahrscheinlichkeit, dass das Bit auf 0=False bleibt ist: $1-\frac{1}{m}$. Beruhend auf der Formel einer statistischen Variation mit Zurücklegen, ist die Wahrscheinlichkeit, dass das Bit nach k Hashfunktionen auf 0=False bleibt: $\left(1-\frac{1}{m}\right)^k$. Unter Einbezug der einzufügenden Elemente: $\left(1-\frac{1}{m}\right)^{kn}$. Das daraus gebildete Komplementärereignis, die Wahrscheinlichkeit das k Hashfunktionen und n Elemente ein Bit auf 1=True gesetzt wurde: $1-\left(1-\frac{1}{m}\right)^{kn}$. Wird zudem Berücksichtigt, dass bei mehreren Hashfunktionen nicht nur ein Bit gesetzt wird ($k > 1$) lautet die Formel:

$$\left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^k \approx \left(1-e^{\frac{-kn}{m}}\right)^k$$

Abbildung 4: *Bloom Filter Approximation zur Berechnung der fp Wahrscheinlichkeit*

Die Formel kann zudem über eine Taylor Reihe approximiert werden (siehe Taylor Reihe: $1-\frac{1}{m} \approx e^{\frac{-1}{m}}$). Wegen der effizienteren Berechnung der Approximation wird diese für die folgenden Formeln verwendet. Für eine bessere Übersicht werden die beiden For-

meln wie folgt benannt: (vgl. [2] und [4])

$$f = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \text{ und } g = \left(1 - e^{\frac{-kn}{m}}\right)^k$$

Zudem kann die Funktion g umgeformt werden zu:

$$g = (1-p)^k \text{ mit } p = e^{\frac{-kn}{m}}$$

Wird diese nun Formel nun umgeschrieben zu: $\ln(g) = k * \ln(1-p)$ und für k: $p = e^{\frac{-kn}{m}} => \ln(p) = \frac{-kn}{m} => k = -\frac{n}{n} * \ln(p)$ eingesetzt, kommt man auf folgende Form: $\ln(g) = -\frac{n}{n} * \ln(p) * \ln(1-p)$. Mit Betracht auf die Hashfunktionen k und den unbekannten Variablen m und n ist die false-positive Wahrscheinlichkeit g möglichst gering, wenn: $\ln(g) = y$ besonders klein für große negative y Werte ist. Aufgrund des negativen Wertes der rechten Seite muss für p gelten: $\ln(p) * \ln(1-p)$ muss möglichst groß sein. Diese Funktion hat ein globales Maximum für $p = 0.5$. Somit wurde festgestellt, dass es eine möglichst geringe *fp* Wahrscheinlichkeit für $p = 0.5$ gibt. Wird nun $p = 0.5$ in die Formel $g = (1-p)^k$ eingesetzt, ergibt sich dafür eine *fp* Wahrscheinlichkeit von $g = \left(\frac{1}{2}\right)^k$. Setzt man für $k = -\frac{n}{n} * \ln(0.5) => \frac{n}{n} * \ln(2)$ ein, erhält man die Formel: $g = 0.5^{\frac{m}{n}\ln(2)}$, welche approximiert: $g \approx 0.6185^{\frac{m}{n}}$ entspricht. Die Wahrscheinlichkeit kann deshalb mit folgenden Formeln berechnet werden:

$$g \approx 0.6185^{\frac{m}{n}} \text{ und } g = \left(\frac{1}{2}\right)^k$$

Abbildung 5: *Berechnung der fp Wahrscheinlichkeit*

Durch das Umformen der Formel kann nun bestimmt werden, wie viele Bits in einem Bloomfilter, unter Einbezug der benötigten *fp* Wahrscheinlichkeit g und der einfügenden Elemente n, benötigt werden. Diese Berechnung wird im *Bf* meistens verwendet, da diese zwei Variablen schon oft vorher dem Benutzer bekannt sind. Dafür wird die Formel: $g = 0.5^{\frac{m}{n}\ln(2)} => \ln(g) = \ln(0.5) * \ln(2) * \frac{m}{n}$ nach m aufgelöst. Nach der Umstellung ergibt sich folgende Form:

$$m \geq -\frac{\ln(g) * n}{\ln(2)^2}$$

Abbildung 6: *Benötigte Arraylänge unter Berücksichtigung der fp Wahrscheinlichkeit und der einzufügenden Elemente*

(Beachte: $\ln(0.5) = -\ln(2)$). Daraus gilt für n eine Maximalanzahl von einzufügenden Elementen von:

$$n \leq -\frac{m}{\ln(g)}\ln(2)^2$$

Abbildung 7: *Berechnung der maximalen Anzahl von einzufügenden Elementen unter Berücksichtigung der fp Wahrscheinlichkeit und der Arraylänge*

Mit diesen Formel kann der *Bf* berechnet werden.

2.3 Eigenschaften und Wahl der Parameter

Um einen effizienten Bloomfilter zu Implementieren, sollten die oben genannten Formel angewendet werden. Natürlich kann auch eine niedrige *fp* Wahrscheinlichkeit erzeugt werden, wenn die Arraylänge vergrößert wird, allerdings wird dadurch der Overhead gesteigert. Gleiches gilt für die Anzahl der Hashfunktionen, die den größten Teil des Overheads in einem *Bf* erzeugen. Mit einem *Bf* kann insbesondere Zeit eingespart werden, wenn der *Bf* die Aussage trifft, dass das Element nicht enthalten ist. Lautet allerdings die Aussage, dass das Element vielleicht enthalten ist, kann in der *DB* geprüft werden ob dies der Fall ist. Wählt man die *fp* Wahrscheinlichkeit so klein das fast nie ein *fp* Ereignis auftreten kann, könnte man die Abfrage gegen die *DB* zusätzlich sparen. Dies gefährdet die Konsistenz in der *DB*, da es dann vorkommen kann das ein Element mehrmals/nie in der *DB* enthalten sein wird (vgl. [6]).

2.4 Geschwindigkeitsvorteil

In dem beiliegenden Beispiel berechnet sich, bei einer gemischten Anfrage (Elemente Enthalten – nicht Enthalten zu je 50 %), der Geschwindigkeitsvorteil. Eine Datenbankabfrage dauerte im Beispiel ca. 0.05 Sekunden, eine Abfrage gegen den Bloomfilter ca. 0.0004 Sekunden. Werden nun 500 *Select* Abfragen direkt gegen die Datenbank ausgeführt, bedeutete dies eine Ausführzeit von $500 * 0.05 = 25$ Sekunden. Wurde ein Bloomfilter für diese Aufgabe verwendet, bei der zusätzlich ein eindeutiges Ergebnis erzielt werden sollte, dauerte die Ausführung: $(250 * (0.0004 + 0.05)) + (250 * 0.0004) = 12.7$ Sekunden. Mit einem *Bf* werden somit 12.3 Sekunden gespart. Ist es nicht notwendig ein eindeutiges Ergebnis zu erzielen (s. Kapitel 2.3) ergibt sich sogar ein Vorteil von $25 - (500 * 0.0004) = 24.8$ Sekunden.

2.5 Bloom Filter in Datenbanken

In vielen Datenbanken ist der *Bf* fester Bestandteil. In den Datenbanksystemen wie *Apache HBase, Post-*

greSQL und *Apache Cassandra* ist der *Bf* fest in der Logik integriert. Zum Erstaunen vieler Benutzer ist dies allerdings nicht der Normalfall. *MySQL* integriert diese Funktionalität nicht in ihr System. Es müsste als eigene Implementierungen in die Programmlogik mit einbezogen werden. Bei der verteilten Datenbank *HBase* gibt es mehrere *Bf* auf den unterschiedlichen Knoten. Um die Bloomfilter konsistent zuhalten ist es möglich diese zusammenzuführen. Bei einem normalen Bloomfilter mit Bitarray lässt

$$Bf(a) \vee Bf(b) = Bf(neu)$$

T	F	F	T	F	F	F	T	F	F
+									
F	F	F	T	T	T	F	F	T	F
=									
T	F	F	T	T	T	F	T	T	F

sich diese Operation leicht über ein Oder Verknüpfen.

2.6 Vor- und Nachteile

Mit einem Bloomfilter kann man wie in den vorigen Kapiteln zu sehen viel Zeit einsparen. Durch die simple Implementierung lässt sich der Filter zudem leicht auf andere Probleme anwenden. Der Hauptvorteil eines Bloom Filters gegenüber einer direkten Datenbankabfrage ist die parallelisierbarkeit der Anfragen. So können z.B. die Hashfunktion simultan berechnet werden. Zusätzlich können alle Anfragen gleichzeitig gestellt werden, solange bis ein neues Element in die Datenbank geschrieben wird und das Bloomfilterarray deshalb geändert wird. Ziel ist aber immer in der Datenbank sowie im Array konsistent zu bleiben. Es stellt sich die Frage, ob ein Bloom Filter überhaupt Nachteile besitzt. Der niedrige Overhead der Anwendung ist in den meisten Applikationen zu vernachlässigen und erübrigt sich schon bei einer verhinderten Anfrage an die Datenbank. Selbst wenn der *Bf* immer die Aussage tätigt, dass das Element vielleicht in der Datenbank ist, sind es nicht mehr Aufrufe als bei einem direkten Zugriff auf die Datenbank. Selbst ein IT-Sicherheitsrisiko Aspekt, kann durch die „Verschlüsselung“ und die multiple Verwendung von Bits im Bitarray ausgeschlossen werden. Ein Nachteil könnte das Debugging dieses Systems sein in Verbindung mit der Konsistenz bei verteilten Systemen (vgl. [3] und [6])

3 Alternativen

Für den Bloom Filter gibt es kaum Alternativen. Meist stellen sich diese als Abwandlungen des Bloomfilters da. Ein weiterer Ansatz wäre den Bloomfilter als probabilistische Datenstruktur abzulösen und z.B. ein neuronales Netzwerk als Alternative zu betreiben. Dieser Ansatz ist allerdings noch kaum erforscht, weil der gezeigte Aufbau des Bloomfilters derzeit als kleinste und effizienteste Struktur für dieses Problem gilt.

4 Fazit

Der Bloomfilter ist eine simple und effektive Implementierung. Durch die probabilistische Datenstruktur können Anfragen effizient und zeitsparend prozessiert werden. In fast keinem Anwendungsfall könnte der Bloomfilter einen Nachteil mit sich ziehen. Insbesondere im Big-Data Bereich lohnt sich die Verwendung eines Bloomfilters und der daraus resultierende Performance- und Zeitgewinn. Schon jetzt ist der Filter in vielen Datenbanken integriert und sollte bei Bedarf auf jeden Fall benutzt werden.

Literatur

[1] Ashwin Lall and Mitsunori Ogihara. *The Bitwise Bloom Filter*. Paper, Rochester.

[2] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. *An Improved Constructionfor Counting Bloom Filters*. PhD thesis, Harvard.

[3] Alan J. Hu and Andrew K. Martin, editors. *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings*, volume 3312 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2004.

[4] Pei Cao. *Bloom Filters - the math*. PhD thesis, University of Wisconsin.

[5] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. *ONTHEFALSE-POSITIVERATEOFBLOOMFILTERS*. PhD thesis, Carleton.

[6] Tim Kaler. *Cache Efficient Bloom Filters for Shared MemoryMachines*. Paper.

Implementierungen im Ordner ../Beispiele.