Bloomfilter: Wie zuverlässig ist Probabilismus

Florian Eberhard Schierz
TU Bergakademie Freiberg
Freiberg, Germany
florian-eberhard.schierz@student.tu-freiberg.de

ZUSAMMENFASSUNG

Bloomfilter sind probabilistische Datenstrukturen, mit denen platzsparend große Mengen an Elementen gespeichert und schnell auf Mitgliedschaft geprüft werden können. Die Kernidee ist, nicht die Elementinformationen selbst zu speichern, sondern nur eine Art Fingerabdruck dieser. Dafür werden die Objekte gehasht und mithilfe dieser Werte ein Array an den so indizierten Stellen manipuliert. Da dieses Verfahren durch die beschränkte Arraygröße und die Charakteristik der Hashfunktionen uneindeutig ist, kann es zu falsch-positiven Aussagen kommen, wohingegen die negativen Ausgaben sicher wahr sind. Die so entstehende Falsch-Positiv-Rate kann jedoch durch geeignete Parameterwahl beliebig klein gehalten werden, wodurch der Bloomfilter auch häufig in der Praxis, besonders in Netzwerkanwendungen, eingesetzt wird. Schwächen der Standardvariante, wie eine fehlende Lösch-Operation oder ein beschränktes Wachstum, werden von verschiedensten Varianten behandelt und verbessert, sodass es mittlerweile zahlreiche Versionen je nach Anforderung und Einsatzgebiet gibt. Die von mir selbst entwickelte Abwandlung namens Floomfilter wird in dieser Arbeit vorgestellt, sollte aber nicht verwendet werden, da sie keine Verbesserung bietet.

KEYWORDS

Datenstrukturen, Bloomfilter, probabilistisch, Hash

ACM Reference Format:
Florian Eberhard Schierz. 2022. Bloomfilter: Wie zuverlässig ist Probabilismus. In *Proceedings of Seminar on Data Structures in C and Ubiquitous Computing (UbiSys Seminar '22)*. ACM, New York, NY, USA, 10 pages.

1 EINLEITUNG

Mit der zunehmenden Digitalisierung wächst auch die Menge an Daten, welche verarbeitet werden muss. Entsprechend ist es wichtig, diese möglichst platzsparend und effizient zu behandeln, um unnötige Wartezeiten und Speicherplatzverschwendung zu verhindern. Genau diese Eigenschaften verspricht die Verwendung von Bloomfiltern. Mit der 1970 von Burton H. Bloom [4] erfundenen Datenstruktur können Aussagen über die Mitgliedschaft eines Elements in einer zuvor eingespeicherten Menge getroffen werden. Da die Elemente selbst jedoch nicht gespeichert werden, nimmt der entsprechende Bloomfilter nur einen Bruchteil des aufsummierten Gesamtspeicherbedarfs ein. Diese Platzersparnis geht dafür mit einer Fehlerrate der getroffenen Aussagen über die Mitgliedschaft einher. Während alle Aussagen der Nicht-Zugehörigkeit mit absoluter Sicherheit zutreffen, kann es zu falsch-positiven Annahmen kommen. So kann nicht garantiert werden, ob ein Element tatsächlich in der Ursprungsmenge enthalten war. Da die Aussagen also nur zu einer gewissen Wahrscheinlichkeit zutreffen, fallen Bloomfilter in die Kategorie der probabilistischen Datenstrukturen. Die Falsch-Positiv-Rate kann jedoch mit entsprechender Parameterwahl (siehe 2.1) beliebig klein gehalten werden. Deshalb gibt es trotzdem viele sinnvolle Anwendungen für Bloomfilter (siehe 5) in der Praxis. Insbesondere in Netzwerkanwendungen werden sie häufig eingesetzt, da dort die Schnelligkeit und Platzersparnis essentiell sind, während seltene Fehler toleriert werden können [19].

Aufgrund seiner positiven Eigenschaften erhält der Bloomfilter bereits seit seiner Erfindung sowohl in der Praxis als auch in der Literatur viel Aufmerksamkeit. So gibt es zahlreiche Publikationen zu Analysen der Zusammenhänge der Parameterwahl und Falsch-Positiv-Rate wie die von Grandi [6], Fan et al. [8] oder Nayak und Patgiri [13]. Weiterhin wurden und werden ständig neue Varianten des Bloomfilters vorgestellt, welche generelle Verbesserungen oder anwendungsspezifische Erweiterungen präsentieren, wie beispielsweise das Hinzufügen einer Lösch-Operation im Counting Bloomfilter von Fan et al. [8] oder Variable-Increment Counting Bloomfilter von Rottenstreich et al. [18]. Weitere Varianten und ihre Vor- beziehungsweise Nachteile werden im Abschnitt 3 diskutiert. Dort werde ich ebenfalls eine selbst entwickelte Abwandlung namens Floomfilter vorstellen und ihre Leistungsfähigkeit analysieren.

Das Ziel dieser Arbeit ist es, einen Überblick über die Funktionsweise von Bloomfiltern zu verschaffen, auf dessen Varianten (siehe Abschnitt 3) sowie vergleichbare Datenstrukturen (siehe

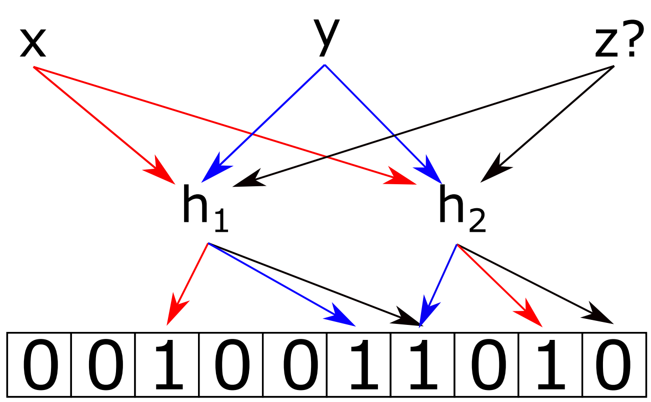

Abbildung 1: Funktionsweise des Standard Bloomfilters

Abschnitt 4) einzugehen und Anwendungsmöglichkeiten (siehe Abschnitt 5) vorzustellen. Dafür habe ich eine Literaturrecherche betrieben und zur Veranschaulichung den Standard Bloomfilter im Vergleich mit dem Floomfilter in der Programmiersprache C implementiert. Der Code dazu ist öffentlich unter https://github.com/Florian2501/Seminar zugänglich.

2 DER STANDARD BLOOMFILTER

Als Standard Bloomfilter oder auch nur Bloomfilter wird die Variante bezeichnet, die ursprünglich von Bloom 1970 [4] entwickelt wurde. Diese Datenstruktur, die die Menge von n Elementen $E = \{e_1, e_2, \dots, e_n\}$ speichern soll, wird als Array der Länge m voller Nullen initialisiert. Die einzutragenden Elemente aus E werden dann der Reihe nach je mittels k Hashfunktionen auf Werte zwischen 0 und $m - 1$ gehasht und der durch den Wert indizierte Eintrag im Array mittels OR auf Eins gesetzt. Als Hashfunktionen eignen sich nach Nayak und Patgiri [13] besonders nicht-kryptografische, unabhängige Funktionen, wie zum Beispiel der Murmur-Hash von Appleby [3].

Dieses Funktionsweise ist in Abbildung 1 mit den Elementen x und y verdeutlicht. Zur Überprüfung der Mitgliedschaft eines beliebigen Elements z in der Ursprungsmenge E wird das gleiche Prinzip verwendet. So werden die k (hier 2) Hashwerte gebildet und die indizierten Stellen des Arrays getestet. Nun können zwei Fälle auftreten:

- Die erste Möglichkeit ist, dass sich an den geprüften Stellen im Array mindestens eine Null befindet. In diesem Fall ist sicher, dass das Element z nicht Teil der Ursprungsmenge war, da sonst jede Position mit Einsen belegt sein müsste.
- Die zweite Möglichkeit ist hingegen nicht so eindeutig. Dadurch dass unendlich viele mögliche Elemente auf die m Stellen des Arrays abgebildet werden, kommt es zwangsweise zu Überschneidungen der Hashwerte und im schlimmsten Fall sogar zu Überschneidungen aller k Werte. Sind also alle geprüften Stellen im Array mit Einsen gefüllt, ist nicht sicher, ob das getestete Element tatsächlich Teil der Ursprungsmenge war. Die Stellen könnten auch zufällig durch das Einfügen anderer Elemente auf Eins gesetzt worden sein.

Dies spiegelt den probabilistischen Charakter der Bloomfilter wieder. Während negative Aussagen über die Zugehörigkeit immer zutreffen, haben positive Aussagen eine Fehlerrate, welche als Falsch-Positiv-Rate (*FPR*) bezeichnet wird und von der Anzahl einzufügender Elemente n sowie der Arraygröße m abhängt. Durch geeignete Wahl der Parameter kann diese Falsch-Positiv-Rate beliebig klein gehalten werden, wie im Folgenden aufgeführt wird.

2.1 Parameterwahl des Standard Bloomfilters

Die vier Parameter des Bloomfilters (Arraylänge m, Anzahl einzufügender Elemente n, Anzahl unabhängiger Hashfunktionen k und die Falsch-Positiv-Rate *FPR* wie in Abbildung 2 dargestellt) hängen voneinander ab. Um sie optimal zu bestimmen, leite ich im Folgenden die Beziehungen der Parameter untereinander her. In einem realistischen Anwendungsszenario ist oft die maximale Fehlerrate sowie die ungefähre Anzahl zu speichernder Elemente bekannt. Darum stelle ich die Formeln so um, dass mithilfe dieser beiden Größen, die anderen bestimmt werden können.

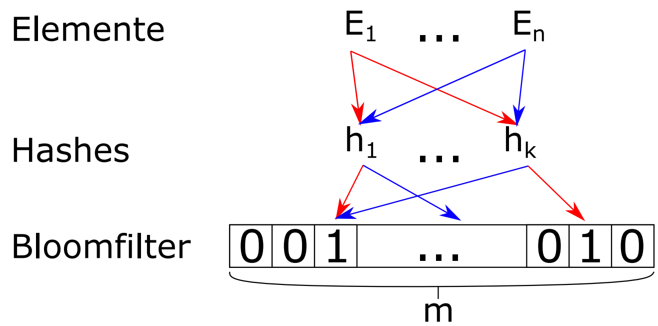

Abbildung 2: Parameterwahl beim Standard Bloomfilter

Unter der Annahme, dass die Hashfunktionen vollständig zufällig und gleichverteilt arbeiten, liegt die Wahrscheinlichkeit für das Ereignis X - *„Ein einzelnes Bit im Bloomfilter wird nicht von einer Hashfunktion ausgewählt“* nach Grandi [6] bei:

$$P(X) = \left(1 - \frac{1}{m}\right) \tag{1}$$

Da pro Element, welches eingefügt wird, jeweils k Hashvorgänge stattfinden und dies wiederum für jedes der n einzufügenden Elemente geschieht ergibt sich für das Ereignis Y - *„Ein einzelnes Bit ist nach dem Einfügen aller Elemente noch Null“*:

$$P(Y) = \left(1 - \frac{1}{m}\right)^{kn} \tag{2}$$

Damit ein Element, welches nicht eingespeichert wurde, fälschlicherweise als in der Ursprungsmenge enthalten ausgegeben wird, müssen an den k indizierten Einträgen des Arrays überall Einsen stehen. Dies passiert zufälligerweise durch das Eintragen anderer Elemente mit der Wahrscheinlichkeit:

$$FPR = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k} \tag{3}$$

Dies entspricht der Falsch-Positiv-Rate des Bloomfilters, die in der Anwendung möglichst klein sein soll. Der Ausdruck lässt sich daher nach Nayak und Patgiri [13] für die Arraylänge minimieren:

$$m = -\frac{n \cdot \ln FPR}{(\ln 2)^2} \tag{4}$$

Über diese Formel kann die optimale Arraylänge m des Bloomfilters bestimmt werden, wenn die maximale Falsch-Positiv-Rate *FPR* und die Anzahl einzutragender Elemente n gegeben ist. Dabei wird auch der lineare Zusammenhang zwischen m und n deutlich. Der Platzbedarf des Bloomfilters wächst also linear mit der Anzahl zu speichernder Elemente an.
Mit dieser berechneten Größe ergibt sich nach Fan et al. [8] auch eine Formel für den letzten offenen Parameter, die optimale Anzahl unabhängiger Hashfunktionen:

$$k = -\frac{m}{n} \cdot \ln 2 \tag{5}$$

Entsprechend dieser Formeln, berechne ich in meiner Implementierung die Parameter m und k mit den Funktionen:

```c
int berechneM(int n, double FPR)
{
    return (int)(ceil((-1)*n*log(FPR)/(log(2)*log(2))));
}

int berechneK(int m, int n)
{
    return (int)(round( (double)m/n * log(2) ));
}
```

Die Anzahl zu speichernder Elemente n und die Falsch-Positiv-Rate kann dabei dem Programm beim Start übergeben werden oder wird auf die Standardwerte $FPR = 0.01$ und $n = 50000$ gesetzt. Wichtig ist hierbei, auf die Art der Rundung zu achten. Während der Wert bei m mittels $ceil()$ aufgerundet wird, um mindestens den benötigten Platz im Array bereitzustellen, wird der Wert für k mittels $round()$ abgerundet, um nicht zu viele verschiedene Hashfunktionen zu verwenden und somit Rechendauer zu sparen, da eben jene Anzahl k, wie im Folgenden aufgeführt, für die Komplexität der Operationen entscheidend ist.

2.2 Operationen und Komplexität

Wie bereits bei der prinzipiellen Funktionsweise erläutert, gibt es im Standard Bloomfilter lediglich zwei Operationen: Einfügen und Prüfen. Da beide im Kern sehr ähnlich ablaufen, lassen sich die Betrachtungen zur Komplexität zusammenfassen.

Im ersten Schritt werden die k Hashwerte des Elements ermittelt. Dazu muss entsprechend oft eine Hashfunktion ausgeführt werden. Es ist also offensichtlich, dass beide Operationen von k abhängig sind. Es bietet sich an, diese Berechnungen in einer Schleife mit k Durchläufen zu implementieren:

```c
void einfuegen(bloomfilter* bf, char* element)
{
    int k = bf->k;
    int m = bf->m;

    for(int i = 0; i < k; i++)
    {
        unsigned int position =
        murmur2(element, strlen(element), i) % m;

        unsigned int char_position = position / 8;
        unsigned int bit_position = position % 8;

        bf->filter[char_position] |= (1<<bit_position);
    }
}
```

Steigt k an, muss auch die Schleife entsprechend häufiger durchlaufen werden. In Abbildung 3 wird der lineare Zusammenhang zwischen der Rechendauer und der Anzahl Hashfunktionen k anschaulich für die Funktion $einfuegen()$ verdeutlicht.

Bei der Betrachtung der Funktion $pruefen()$ ist der Zusammenhang nicht ganz so offensichtlich. Der Graph in Abbildung 4 zeigt ebenso wie der der Funktion $einfuegen()$ einen eindeutigen linearen Zusammenhang zwischen k und der benötigten Rechenzeit. Diese Messungen erfolgten ausnahmslos mit Elementen, die tatsächlich Teil des Bloomfilters waren und zu Beginn eingespeichert wurden.

Im Graph der Abbildung 5 sind deutlich mehr und insbesondere größere Ausreißer sichtbar. Ignoriert man diese, sieht er außerdem eher wie eine konstante Funktion aus. Das liegt daran, dass hier nur

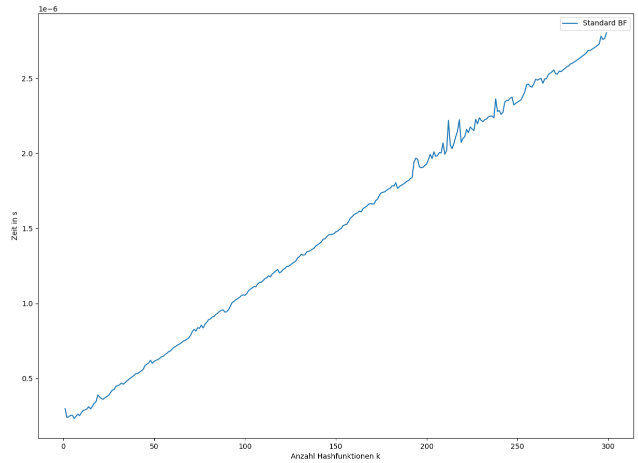

Abbildung 3: Rechendauer der Funktion $einfuegen()$ in s in Abhängigkeit von der Anzahl Hashfunktionen k

Elemente getestet wurden, die nicht im Bloomfilter enthalten waren, die Funktion in meiner Implementierung aber direkt zurückspringt, sobald auch nur eine indizierte Stelle gleich Null ist. Somit finden nicht immer alle k Schleifendurchläufe statt. Wird ein zuvor nicht eingespeichertes Element auf Mitgliedschaft geprüft, besteht also die Möglichkeit, dass die Funktion Rechenleistung spart. Dadurch wird der lineare Zusammenhang im Graph verwischt. Die Spitzen im Graph sind Falsch-Positive, bei denen trotzdem alle Schleifen durchlaufen wurden.

Aufgrund der Unvorhersehbarkeit dieser vorzeitigen Abbrüche, da die Verteilung der Nullen zufällig vorliegt, ist im Durchschnittsfall trotzdem davon auszugehen, dass der Zusammenhang zu k linear ist und die Rechendauer somit mit der Anzahl Hashfunktionen steigt beziehungsweise fällt. In Abbildung 6 sieht man den Graphen zu einer Messung in der sowohl enthaltene als auch nicht

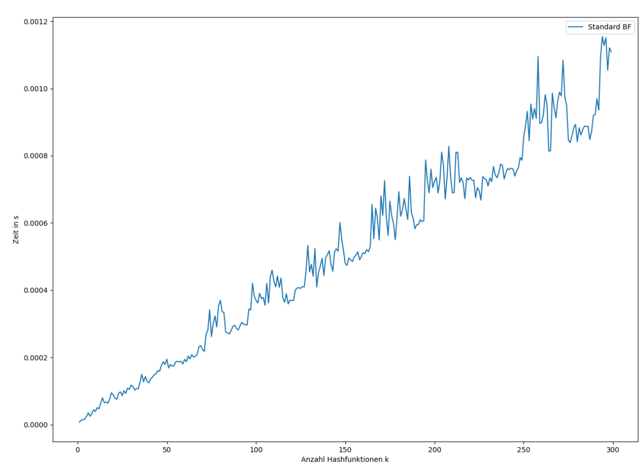

Abbildung 4: Rechendauer der Funktion $pruefen()$ in s in Abhängigkeit von der Anzahl Hashfunktionen k mit ausnahmslos enthaltenen Elementen

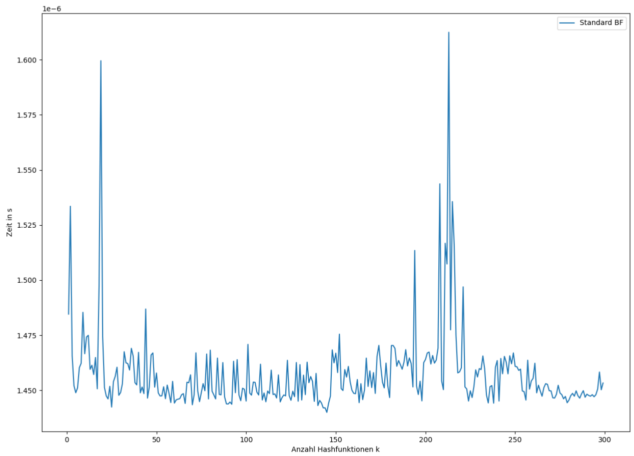

Abbildung 5: Rechendauer der Funktion $pruefen()$ in s in Abhängigkeit von der Anzahl Hashfunktionen k mit ausnahmslos nicht enthaltenen Elementen

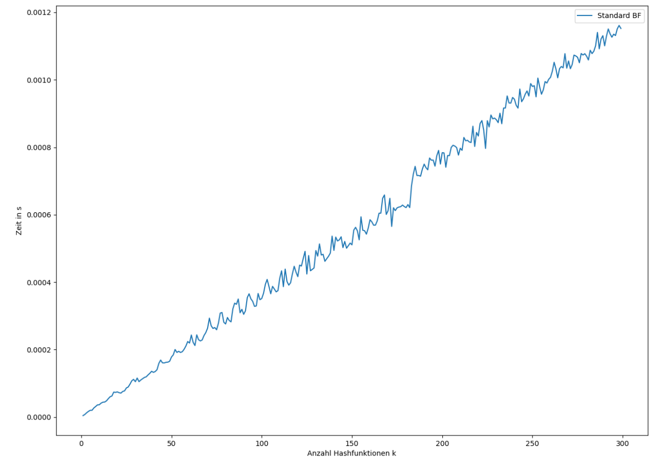

Abbildung 6: Rechendauer der Funktion $pruefen()$ in s in Abhängigkeit von der Anzahl Hashfunktionen k mit sowohl enthaltenen Elementen als auch nicht enthaltenen Elementen im Verhältnis $10:1$

enthaltene Elemente getestet wurden. Das Verhältnis betrug dabei $10:1$. Die lineare Abhängigkeit von k ist auch hier gut sichtbar.

Mit den berechneten Hashwerten werden die dadurch indizierten Positionen im Array gesetzt beziehungsweise überprüft. Die Dauer dieses Vorgangs ist dabei natürlich unabhängig von der Größe des Arrays m, da die Referenzierung eines Eintrags in Arrays immer gleich aufwendig ist. Schließlich wird nur eine andere Stelle im Arbeitsspeicher abgerufen, welche die gleichen Zugriffszeiten hat.

Weiterhin ist dieser Vorgang unabhängig von der Anzahl eingefügter Elemente n, da die OR-Operation beim Einfügen, sowie das Prüfen auf Null davon nicht beeinflusst werden, wie viele Elemente bereits im Array gespeichert wurden [19]. Auch wenn man zunächst denken könnte, dass zumindest beim Prüfen die Anzahl eingefügter Elemente eine Rolle spielt, da gilt: Je weniger eingefügte Elemente, desto mehr Nullen im Array. Das wiederum erhöht die Chance beim Prüfen eher eine Null zu treffen und somit Rechenzeit zu sparen. Die Wahrscheinlichkeit, dass ein indizierter Eintrag Null ist oder anders gesagt das Verhältnis von Nullen und Einsen im Array ist jedoch immer gleich, da die Arraygröße m sich der Anzahl einzufügender Elemente n anpasst, wie in Formel (4) gezeigt. Der Parameter n hätte lediglich dann einen Einfluss, wenn man das Prüfen beginnt, bevor alle Elemente eingespeichert sind, da dann das Verhältnis verzerrt wäre. Beim Standard Bloomfilter geht man aber davon aus, dass zu Beginn alle Elemente eingetragen werden und erst danach das Prüfen beginnt.

Daraus wird deutlich, dass die Komplexität der beiden Operationen lediglich linear von k abhängt, also $O(k)$ für Einfügen und Prüfen gilt [1].

In anderen Varianten des Bloomfilters gibt es noch weitere Operationen, wie das Löschen eines Elements aus dem Array, beispielsweise im Counting Blomfilter, Variable-Increment Counting Blomfilter oder Deletable Blomfilter. Darauf werde ich im folgenden Abschnitt 3 eingehen, jedoch auf die Komplexitätsanalyse verzichten.

3 VARIANTEN DES BLOOMFILTERS

Durch seine Schnelligkeit und den geringen Speicherbedarf bringt der Bloomfilter zwei wertvolle Eigenschaften mit, die ihn von anderen Datenstrukturen abheben. Jedoch hat die Standardvariante auch einige Schwächen.

So müssen für eine optimale Parameterwahl bereits vor der Erstellung mindestens zwei der vier Parameter bekannt sein. Erhöht sich im Verlauf der Ausführung dann beispielsweise die Anzahl zu speichernder Elemente n ist es nicht mehr möglich die Arraygröße m anzupassen und somit verschiebt sich deren Verhältnis. Das wiederum hat gemäß Formel (3) Einfluss auf die Falsch-Positiv-Rate. Der Standard Bloomfilter ist, nachdem er initialisiert wurde, also unflexibel. Dies zeigt sich auch in der fehlenden Möglichkeit, Elemente zu löschen. Um diese und weitere Schwächen auszugleichen und den Standard Bloomfilter spezifisch an Anwendungsszenarien anzupassen, wurden daher zahlreiche Abwandlungen erfunden. Im Folgenden werde ich einen Überblick über einige verbreitete Varianten geben.

3.1 Counting Blomfilter

Der Counting Bloomfilter von Fan et al. [8] ist eine der einfachsten Abwandlungen des Standard Bloomfilters. Der einzige Unterschied liegt darin, dass das Array hier aus Zählern besteht, statt je nur aus einzelnen Bits. Für gewöhnlich werden hierfür 4-Bit-Zähler verwendet, sodass der Zahlenraum von 0 bis 15 abgedeckt ist. Wie beim Standard Bloomfilter wird auch dieser zu Beginn mit Nullen gefüllt. Im Anschluss werden beim Eintragen der Elemente die indizierten Stellen aber nicht auf Eins gesetzt, sondern um Eins erhöht. Beim Prüfen gilt weiterhin die selbe Regel, dass Nullen bedeuten, das Element kann nicht in der Ursprungsmenge enthalten gewesen sein. Sind alle Stellen ungleich Null, wird vermutet, dass das Element Teil war. Dieses Prinzip ist in Abbildung 7 beispielhaft dargestellt. Element x und y indizieren beide die dritte Stelle im Array, weshalb der Zähler dort zweimal erhöht wird.

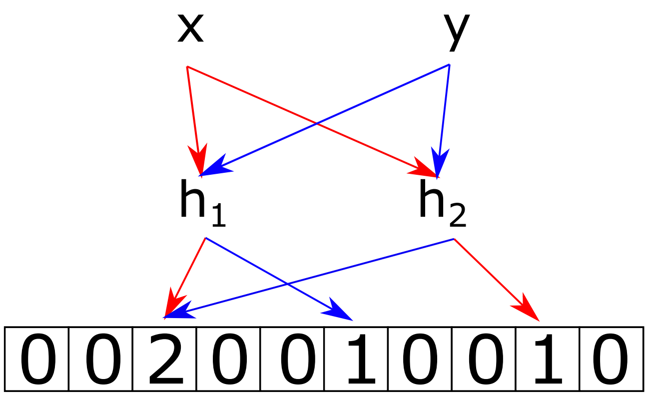

Abbildung 7: Funktionsweise des Counting Bloomfilters

Der eigentliche Mehrwert des Counting Bloomfilters liegt darin, dass durch das zusätzlich vorhandene Wissen über die Anzahl der Elemente, die eine bestimmte Stelle referenzieren, das Löschen als weitere Operation ermöglicht wird. Im Standard Bloomfilter ist dies nicht möglich, da nicht bekannt ist, ob ein weiteres Objekt eine Stelle indiziert hat und dieses so unbeabsichtigt mit gelöscht werden könnte. Durch die Zähler wird gewährleistet, dass auch nach dem Entfernen, der Eintrag noch ungleich Null ist, insofern vorher mehrere Elemente die Stelle indiziert haben. Entsprechend wird das Löschen analog zum Einfügen umgesetzt. Lediglich werden die referenzierten Zähler dann nicht um Eins erhöht, sondern verringert. Ist ein Eintrag bereits auf Null, ist ein Fehler aufgetreten oder das Element war nicht Teil des Filters, weshalb der Vorgang abgebrochen werden muss. Im Beispiel der Abbildung 7 könnte das Element x entfernt werden, indem wieder die beiden Hashwerte gebildet und dann die so erhaltenen dritten und neunten Einträge jeweils um Eins verringert werden.

Der Counting Bloomfilter ist also wesentlich flexibler als die Standardvariante, indem auch nach dem Erstellen noch Anpassungen am Inhalt vorgenommen werden können. Dieser Vorteil geht allerdings auch mit mehr benötigtem Speicherplatz einher. Während der Standard Bloomfilter nur m Bit Speicherplatz benötigt, braucht der Counting Bloomfilter mit l-Bit-Zählern entsprechen $l \cdot m$ Bit Speicherplatz.

3.2 Variable-Increment Counting Blomfilter

Wiederum eine Erweiterung des Counting Bloomfilters ist der Variable-Increment Bloomfilter, der 2014 von Rottenstreich, Kanizo und Keslassy vorgestellt wurde [18]. Auch hier werden Zähler statt nur einzelner Bits verwendet, wodurch der Speicherbedarf entsprechend der Zählergröße analog zum Counting Bloomfilter vervielfacht wird. Die Schlüsselidee beim Variable-Increment Counting Bloomfilter ist, dass die Zähler mit bestimmten Werten erhöht werden. Diese werden mittels sogenannter $\tilde{B_h}$-Reihen, einer Abwandlung der B_h-Reihen [9], bestimmt. Die Summanden $s_i \in S = \{s_1, ..., s_o\}$ werden dabei so gewählt, dass man anhand der Summe, die sich aus ihnen ergibt, Rückschlüsse auf die mögliche Zusammensetzung ziehen kann, da diese voneinander disjunkt sind. Somit kann nicht nur anhand der Überprüfung auf Null darauf geschlossen werden, ob ein Element enthalten ist, sondern zudem

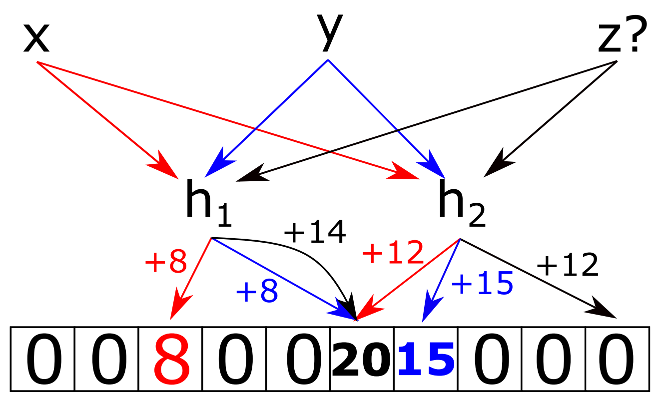

Abbildung 8: Funktionsweise des Variable-Increment Counting Bloomfilters

auch durch die Überprüfung der nur beschränkt möglichen Summanden.

In Abbildung 8 ist die Funktionsweise verdeutlicht. Nachdem Element x und y in den Filter eingetragen wurden, erfolgt die Prüfung von z. Hierfür werden jeweils die Hashwerte für die Position im Array (zwischen 0 und $m-1$) und für die Wahl des Summanden s_i (zwischen 1 und o) berechnet und die jeweils indizierte Stelle um s_i erhöht. Der sechste Wert im Array wird in Abbildung 8 jeweils von Element x und y indiziert und um acht beziehungsweise zwölf erhöht. Bei der Überprüfung des Elements z ergibt sich aus zwei Gründen die korrekte Aussage, dass es nicht in der Ursprungsmenge enthalten ist:

- Der erste Grund ist offensichtlich und genau der gleiche wie bei den beiden anderen bislang behandelten Varianten. Die von z indizierte, letzte Stelle des Arrays ist Null, weshalb eine Mitgliedschaft ausgeschlossen ist.
- Sollte an dieser Stelle jedoch bereits ebenfalls eine Zwölf stehen, würde dieses Kriterium nicht greifen. Trotzdem könnte man anhand des zweiten Grundes eine Mitgliedschaft ausschließen. An der vorderen indizierten Stelle steht 20 und der Summand von z ist 14. Es gibt in der potentiellen Menge an Summanden $S = \{8, 12, 14, 15\}$ jedoch keine Möglichkeit, wie die Summe 20 mit dem Summand 14 gebildet wird, da $20 - 14 = 6$. Somit ist offensichtlich, dass z nicht in der Ursprungsmenge gewesen sein kann.

Durch diese zusätzliche Kontrollmöglichkeit kann also die Falsch-Positiv-Rate verbessert werden. Da die Zähler je nach Wahl der Menge S hierbei aber auch größere Zahlen speichern können müssen, benötigen sie noch mehr Speicherplatz als der Counting Bloomfilter. Ebenso wie bei diesem ist das Löschen als neue Operation hier analog möglich, was eine weitere Verbesserung im Vergleich zur Standardvariante ist.

3.3 Deletable Blomfilter

Der Deletable Bloomfilter von Rothenberg et al. [17] ermöglicht wie die beiden zuvor vorgestellten Varianten ebenfalls das Löschen von Elementen aus dem Array, kann dies aber bereits mit weniger zusätzlichem Speicher bewerkstelligen. Dafür ist das Löschen ganz im probabilistischen Sinne des Bloomfilters nicht immer möglich, sondern nur zu einer bestimmten Wahrscheinlichkeit p_d. Als Ausgangslage für den Deletable Bloomfilter dient die Standardvariante. Diese wird in b $\frac{m}{b}$ Bit große Bereiche unterteilt und weiterhin zusätzlicher Speicher der Größe b Bit alloziert. Dieser Zusatzspeicher besteht zu Beginn genau wie das restliche Array nur aus Nullen und wird erst beim Eintragen der Elemente relevant. Hierbei wird immer geprüft, ob die durch eine der k Hashfunktionen indizierte Stelle bereits eine Eins enthält. Ist dies der Fall, wird das i-te Bit im Zusatzspeicher ebenfalls auf Eins gesetzt, wobei i die Nummer des Bereichs angibt, in dem die Stelle liegt.

In Abbildung 9 ist die Funktionsweise beispielhaft verdeutlicht. Element x und y werden in den Filter eingetragen. Zuerst setzt x den dritten Eintrag auf Eins und im Anschluss indiziert der Hashwert von y wieder die gleiche Stelle. Da dort nun schon eine Eins ist, wird der zweite Bereich im Zusatzspeicher markiert, da bei einer Bereichsgröße von zwei Bit der dritte Eintrag im zweiten solchen Bereich liegt. Die restlichen Stellen werden je nur einmal indiziert, weshalb die Einträge im Zusatzspeicher hier auf Null bleiben.

Während das Prüfen genau wie beim Standard Bloomfilter bleibt, läuft das Löschen wieder nahezu analog zum Einfügen ab. Nach dem Berechnen der Hashwerte werden die indizierten Stellen ermittelt und jeweils dazu, in welchem Bereich i sie liegen. Diese Stellen im Zusatzspeicher werden dann überprüft. Nun können wieder zwei Fälle auftreten:

- Ist das Bit im Zusatzspeicher eine Null, bedeutet dies, dass kein einziges Bit in diesem Bereich des Arrays doppelt indiziert wurde. Somit kann die Stelle im Bloomfilter problemlos auf Null gesetzt werden, da keine Gefahr besteht, ein anderes Element mit zu löschen.
- Ist das Bit im Zusatzspeicher hingegen eine Eins, kann die ursprünglich indizierte Stelle im Filter nicht auf Null gesetzt werden, da die Möglichkeit besteht, dass die Position durch mehrere Elemente indiziert wurde, wodurch das Löschen weitere unbekannte Elemente entfernen könnte.

Um ein Elemente aus dem Deletable Bloomfilter zu löschen, reicht es aus, wenn eine der k Stellen wieder auf Null gesetzt werden kann, da das Prüfen auch so schon korrekt funktioniert. Die Wahrscheinlichkeit dafür, dass dies möglich ist ergibt sich nach Rothenberg et al. [17] wie folgt:

$$p_d = (1 - (1 - p_c)^{m/r})^k \tag{6}$$

Mit der Wahrscheinlichkeit p_c, dass eine bestimmte Stelle im Filter mindestens zweimal durch verschiedene Elemente indiziert wurde:

$$p_c = 1 - p_0 - p_1 \tag{7}$$

Wobei p_0 die Wahrscheinlichkeit ist, dass eine bestimmte Stelle im Array gleich Null ist (siehe Formel (2)). Hingegen beschreibt p_1 die Wahrscheinlichkeit, dass nach dem Einfügen von n Elementen eine bestimmte Arraystelle genau nur einmal auf Eins gesetzt wurde:

$$p_1 = (kn)\left(\frac{1}{m}\right)\left(1 - \frac{1}{m}\right)^{kn-1} \tag{8}$$

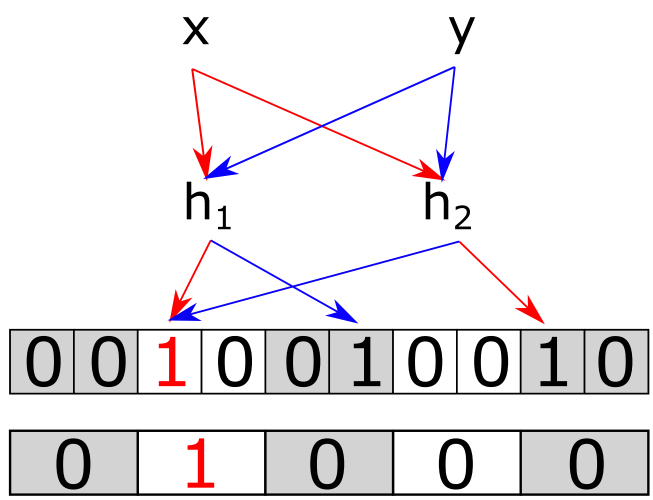

Abbildung 9: Funktionsweise des Deletable Bloomfilters, wobei die Färbung die Bereichsgrenzen darstellt

Sind alle geprüften Bereiche im Zusatzspeicher als unsicher markiert, kann das Element also nicht gelöscht werden. Die Wahrscheinlichkeit dafür kann anhand der Formel 6 minimiert werden, indem beispielsweise die Bereichsgröße verkleinert wird und somit die Auflösung steigt. Dadurch erhöht sich aber auch wieder der Speicherbedarf. Prinzipiell ist der Deletable Bloomfilter trotzdem sparsamer als der Counting Blomfilter und der Variable-Increment Counting Blomfilter, da bei diesen je nach Zählergröße der Speicherbedarf vervielfacht wird, während er hier im schlimmsten Fall nicht einmal verdoppelt wird, da dies hieße einen Bereich pro Element zu haben.

3.4 Scalable Blomfilter

Neben der fehlenden Möglichkeit eingetragene Elemente zu löschen, ist die zweite große Schwachstelle des Standard Bloomfilters, dass vor dem Erstellen die Anzahl einzutragender Elemente bekannt sein muss, um entsprechend der Formeln (3), (4) und (5) die weiteren Parameter zu bestimmen. Ein späteres Wachstum ist nur möglich, wenn eine somit ebenfalls steigende Falsch-Positiv-Rate in Kauf genommen wird. Dieses Problem adressiert der Scalable Bloomfilter von Almeida et al. [2] und ermöglicht auch nach dem Erstellen ein beliebiges Wachstum.

Dabei bildet eine Abwandlung des Standard Bloomfilters die Grundlage. Bei dieser wird das Array in k Bereiche unterteilt. Jede der Hashfunktionen hat somit einen eigenen $\frac{m}{k}$ Bit großen Bereich, auf den nur sie abbildet. Der Unterschied zur Standardvariante liegt darin, dass so garantiert wird, dass k Stellen im Filter gesetzt werden. Beim Standard Bloomfilter besteht die Möglichkeit, dass mehrere Hashfunktionen zufällig den gleichen Eintrag indizieren.

Zu Beginn gleicht ein Scalable Bloomfilter also stark der Standardversion. Wird aber der maximale Füllstand n dieses Ausgangsfilters erreicht, wird ein weiterer erstellt und angehangen. Neue Elemente werden dann nur noch in diesen eingetragen. Dessen einzelne Falsch-Positiv-Rate ist geringer als die des vorherigen, um die gesamte Falsch-Positiv-Rate zu gewährleisten. Sei FPR_0 die Fehlerrate des Ausgangsfilters, dann sei $FPR_1 = r \cdot FPR_0$ oder

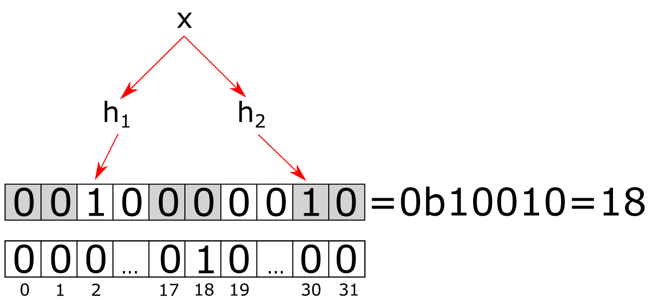

Abbildung 10: Funktionsweise des Floomfilters

allgemein $FPR_{i+1} = r \cdot FPR_i$ mit $0 < r < 1$. Dadurch erhöht sich für die jeweils neuen Filter ebenfalls der Speicherbedarf. Die gesamte Flasch-Positiv-Rate ergibt sich nach Almeida et al. [2] zu:

$$FPR \leq \frac{FPR_0}{1 - r} \tag{9}$$

Beim Prüfen werden der Reihe nach die Filter durchlaufen und auf Mitgliedschaft getestet. Sobald das Ergebnis in einem positiv ist, kann abgebrochen werden. Ist das Ergebnis hingegen immer negativ, wurde das Element sicher nicht eingespeichert.

3.5 Floomfilter

Während meiner Recherchen zu den Varianten von Bloomfiltern kam mir die Idee einer eigenen Variante, welche ich Floomfilter getauft habe. Im Folgenden erläutere ich dessen Funktionsweise und vergleiche die Leistungsfähigkeit mit der Standardversion. Der Code dazu befindet sich ebenfalls auf GitHub.

Wie der Deletable Blomfilter geht auch der Floomfilter von einem Standard Bloomfilter, der in b Bereiche der Größe $\frac{m}{b}$ Bit unterteilt wird, aus und erweitert diesen lediglich um einen Zusatzspeicher. Dessen Zweck ist jedoch ein anderer. Der Floomfilter adressiert nämlich nicht das Problem der fehlenden Lösch-Operation, sondern versucht lediglich die Falsch-Positiv-Rate zu verbessern. Die durchnummerierten Bereiche werden dazu beim Einfügen in Binärzahlen übertragen. So wird das i-te Bit eines Integers mittels OR auf Eins gesetzt, wenn ein Hashwert in den i-ten Bereich indiziert hat. Somit ergibt sich eine Zahl, die als Index für den Zusatzspeicher genutzt wird, um das Bit an dieser Stelle auf Eins zu setzen. Die Idee ist also, die Verteilung der Hashwerte zu speichern, um so beim Prüfen zusätzliches Wissen zu haben, anhand dessen die Frage der Mitgliedschaft beantwortet werden kann.

Dies wird in Abbildung 10 anhand des Einfügens von Element x verdeutlicht. Die Hashfunktionen bilden x auf zwei Stellen im Array ab, welches in $b = 5$ Bereiche mit jeweils zwei Bit unterteilt ist. Da die Eintragungen im zweiten und fünften Bereich erfolgen, ergibt sich als Hashwertverteilung die Dezimalzahl 18, weil binär das zweite und das fünfte Bit beginnend mit dem Least Significant Bit auf Eins gesetzt werden. Entsprechend wird im $2^5 = 32$ Bit großen Zusatzspeicher die Position mit Index 18 gesetzt.

Der Prüfprozess läuft wie folgt ab: Nachdem analog zum Standard Bloomfilter alle Hashwerte und so erhaltene Arraypositionen getestet wurden, ergibt sich wieder anhand der Verteilung der Werte

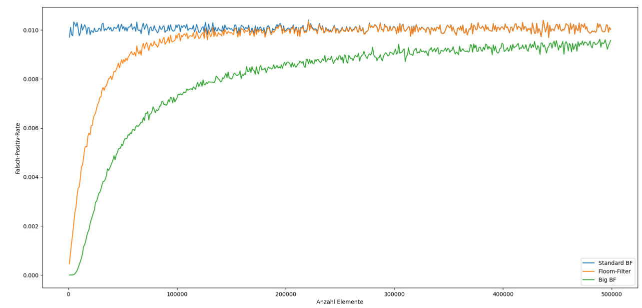

Abbildung 11: Vergleich des Floomfilters (gelb) mit dem Standard Bloomfilter gleicher Größe (grün) sowie der Standard Variante ursprünglicher Größe (blau) mit maximaler $FPR = 0,01$ und $b = 16$, Falsch-Positiv-Rate in Abhängigkeit der gesamt einzufügenden Elemente n

in den Bereichen eine Integerzahl, welche eine Stelle im Zusatzspeicher indiziert. Steht an dieser Position eine Null, kann das Element ebenfalls nicht Teil der Ausgangsmenge gewesen sein, da es genau diese Verteilung der Hashwerte sonst bereits hätte geben müssen, wodurch die Stelle auf Eins gesetzt sein müsste. Durch dieses Zusatzkriterium können also weitere Elemente ausgeschlossen und somit die Falsch-Positiv-Rate gesenkt werden.

Je nachdem wie groß man b wählt, steigt oder fällt die Auflösung der Hashwertverteilung und somit deren Aussagekraft. Zudem verändert sich die Größe des Zusatzspeichers. Dieser ist je 2^b Bit groß, wodurch eine höhere Auflösung zwar die Falsch-Positiv-Rate senkt, aber auch enormes Speicherwachstum bedeutet. Wählt man b hingegen klein, ist der Zusatzspeicher schnell komplett voller Einsen, wodurch er keinen Mehrwert bietet.

Um die Sinnhaftigkeit des Floomfilters zu testen, habe ich einen Standard Bloomfilter mit der Gesamtgröße des normalen Filters und des Zusatzspeichers erstellt und beide hinsichtlich der resultierenden Falsch-Positiv-Rate verglichen. Dies ist in Abbildung 11 für die Parameter $b = 16$ und $FPR = 0,01$ dargestellt. Die gelbe Kurve des Floomfilters verläuft zu Beginn deutlich unterhalb der blauen des Standard Bloomfilters, was zu erwarten war, da der Floomfilter diesen ja noch um den Zusatzspeicher erweitert und somit besser abschneidet. Ab circa 200 000 einzutragenden Elementen ist jedoch bereits kein Unterschied mehr sichtbar, was darauf schließen lässt, dass der Zusatzspeicher dann mit Einsen gefüllt ist. Vergleicht man das ganze jedoch noch mit der grünen Kurve des Standard Bloomfilters, der auf die Gesamtgröße des Floomfilters samt Zusatzspeicher (hier $2^{16} = 65536$ Bit) initialisiert wurde, wird klar, dass der Floomfilter zwar eine Verbesserung der Falsch-Positiv-Rate mit sich bringt, dies jedoch deutlich platzineffizienter als der Standard Bloomfilter der selben Größe.

Ich habe diese Tests mit vielen Kombinationen der Parameter b, n und FPR durchgeführt und konnte keine signifikante Verbesserung durch den Floomfilter im Vergleich zur Standardvariante gleicher Größe feststellen. Die Ergebnisse habe ich in der Datei $floom.csv$ beziehungsweise $VergleichFloom.xlsx$ zusammengefasst. Diese sind ebenfalls auf GitHub zu finden. Der Floomfilter verfehlt also klar sein Ziel, weshalb eine Verwendung nicht zu empfehlen ist.

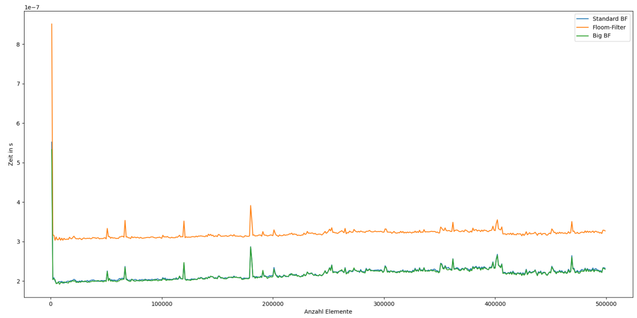

Abbildung 12: Vergleich des Floomfilters (gelb) mit dem Standard Bloomfilter gleicher Größe (grün) sowie der Standardvariante ursprünglicher Größe (blau) mit maximaler $FPR = 0,05$ und $b = 8$, Zeit zum Einfügen in s in Abhängigkeit der gesamt einzufügenden Elemente n

Das erste Bit des Zusatzspeichers ist außerdem verschwendet, da immer mindestens k Stellen im Array indiziert werden, die im schlimmsten Fall alle im ersten Bereich liegen, wodurch mindestens das erste Element indiziert wird, aber nie das mit Index Null. Da sich dieses Bit aufrgrund der verbreiteten Computerarchitekturen aber nur schwer nutzen lässt, habe ich dies in meiner Implementierung ignoriert.

Ein weiterer Aspekt, der gegen die Verwendung des Floomfilters spricht, ist dessen Laufzeit. Durch die zusätzlichen Operationen, um die Hashwertverteilung zu speichern und im Zusatzspeicher zu setzen beziehungsweise zu prüfen, benötigt er länger als die Standard Bloomfilter, obwohl sich die Komplexität nicht ändert, da die Operationen offensichtlich primär weiterhin nur von k abhängen. Dies ist in Abbildung 12 für die Parameter $FPR = 0,05$ und $b = 8$ verdeutlicht. Während die blaue und grüne Kurve der Standard Bloomfilter nahezu identisch übereinander liegen, was die in 2.2 diskutierte Unabhängigkeit von n verdeutlicht, sieht man die gelbe Kurve des Floomfilters deutlich darüber.

4 VERGLEICH ZU ANDEREN STRUKTUREN

Durch ihre Funktionsweise unterscheiden sich Bloomfilter stark von vielen anderen Datenstrukturen, die die Elemente selbst speichern, wie Linked Lists. Trotzdem gibt es vergleichbare Strukturen, die in Konkurrenz zu Bloomfiltern stehen. Im Rahmen dieser Arbeit, werde ich diese jedoch nur erwähnen und auf eine umfangreiche Analyse verzichten.

Bereits 2005 veröffentlichten Pagh, Pagh und Rao [14] eine alternative Datenstruktur unter dem Titel: „Ein optimaler Bloomfilter-Ersatz“. Diese bietet die Möglichkeit, Elemente zu löschen, den Speicher effizienter auszunutzen und eine Komplexität von $O(1)$ für das Prüfen.

Eine weitere Alternative bietet der 2014 von Fan et al. [7] veröffentlichte Cuckoofilter. Auch diese Datenstruktur ermöglicht das Löschen, verbessert die Prüfzeit und verspricht eine effizientere Speichernutzung, sofern die Fehlerrate unter 3% liegt. Dafür werden die Hashwerte der Elemente mittels *cuckoo hashing* gespeichert, statt diese nur als Indizes zu verwenden.

5 ANWENDUNGEN

In diesem Abschnitt werde ich einige der Anwendungsmöglichkeiten des Bloomfilters und seiner Varianten vorstellen. Dabei unterscheide ich wie Pal und Sardana [19] die Kategorien Netzwerkanwendungen, Sicherheitsanwendungen und Weitere Anwendungen.

5.1 Netzwerkanwendungen

5.1.1 Routing. Mittels Bloomfiltern können Ressourcenanfragen innerhalb von Netzwerken effizient an ihr Ziel weitergeleitet werden. Dazu speichert jeder Knoten in erster Ebene für alle seine Nachbarn jeweils die in ihnen erreichbaren Ressourcen in einem eigenen Filter. In den weiteren Ebenen werden die vom jeweiligen Nachbar aus erreichbaren Ressourcen in je einen weiteren Filter eingefügt. Diese Ebenenanzahl wird auch Tiefe genannt. So wird der Speicherort der angefragten Ressource ermittelt und die Anfrage entsprechend weitergeleitet. Falsch-Positive wirken sich hierbei so aus, dass die Anfrage einen Knoten erreicht, der sie nicht bedienen kann. In solchen Fällen kann die Nachricht einfach verworfen oder auch ein deterministischer Algorithmus, der jedoch auch aufwendiger ist, genutzt werden, um den tatsächlichen Ort sicher zu bestimmen [16].

5.1.2 Loop Prevention. Bloomfilter eignen sich aber nicht nur, um Ziele in Netzwerken zu ermitteln, sondern auch um zu verhindern, dass die Pakete auf ihrem Weg dahin nicht in Schleifen stecken bleiben. Dafür kann jedem Paket ein Bloomfilter fester Größe im Header hinzugefügt werden, der zu Beginn voller Nullen ist. Jedes Netzwerkinterface, das nun im Verlauf passiert wird, trägt beim Weiterleiten mittels OR dann seine Bloommaske in den Filter ein. Diese ist analog zum Hashwert eines einzutragenden Elements, muss aber nur einmal berechnet werden und wird dann immer wieder verwendet. Ausgangspunkt dafür kann beispielsweise die MAC-Adresse sein. Ändert sich beim Eintragen in den Filter keine der Stellen, da bereits alle gesetzt sind, liegt die Vermutung nahe, dass das Paket dieses Interface bereits passiert hat, weshalb es verworfen werden sollte. Entsprechend der Falsch-Positiv-Rate kann es aber auch eine Falschaussage sein, weshalb die Parameterwahl der Filter entsprechend geeignet sein sollte, um diese unnötigen Verluste zu verringern [21].

5.1.3 IP-Routenverfolgung. Nahezu analog zur Loop Prevention können Bloomfilter genutzt werden, um den Weg eines Pakets im Netzwerk nachzuvollziehen. So werden allen Headern ebenfalls Bloomfilter hinzugefügt, in die an jedem Interface der einmalig berechnete Hashwert der jeweiligen IP-Adresse eingetragen wird. Somit entsteht kein zeitlicher Mehraufwand, da dieser Prozess auch in Hardware umgesetzt werden kann. Um den Ursprung des Pakets zu ermitteln, prüft nun der Empfängerknoten, weche IP-Adresse seiner Nachbarn im Filter enthalten ist und leitet die Anfrage an den entsprechenden Knoten weiter, der als Vorgänger ausgemacht werden konnte. Dieser Prozess wird so lange wiederholt bis kein Nachbar mehr gefunden werden kann. Dann ist davon auszugehen, dass der Ursprung erreicht ist [11].

5.1.4 Verhindern von DDoS-Angriffen. Ein einfacher Ansatz gegen DDoS-Attacken ist das Blacklisten von IP-Adressen oder ganzen

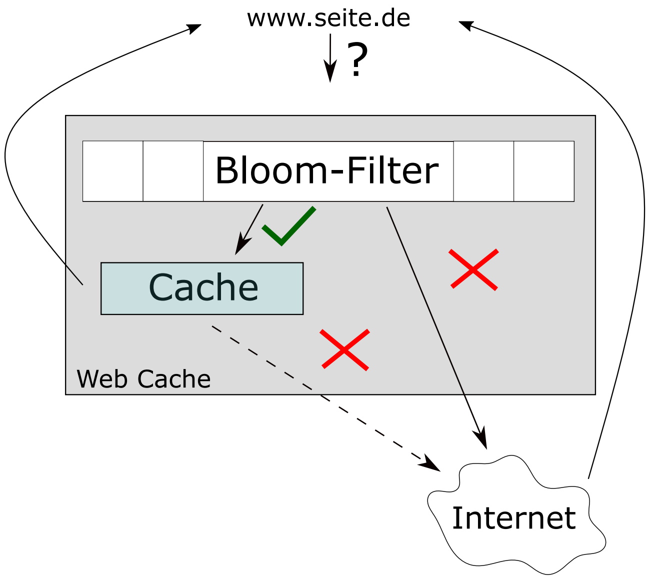

Abbildung 13: Funktionsweise eines Web Caches mittels Bloomfiltern

Bereichen. Ein Angreifer kann beispielsweise über die eben vorgestellte IP-Routenverfolgung ermittelt und seine Anfragen anhand der IP-Adresse identifiziert und abgeblockt werden, sodass sie den bereitgestellten Dienst nicht überlasten. Zum Speichern dieser Blacklist bieten sich ebenfalls wieder Bloomfilter an, da in ihnen die Adressen schnell und platzsparend gespeichert und ebenso überprüft werden können. Analog können Bloomfilter auch für eine Whitelist verwendet werden. Da IP-Adressen meist nach bestimmten Zeiten wieder freigegeben werden, bieten sich hier Varianten des Bloomfilters an, die Lösch-Operationen erlauben [15].

5.1.5 Web Caches. Genau wie die IP-Adressen können auch URLs in Web Caches in Bloomfiltern gespeichert werden, um schnell überprüfen zu können, ob die angefragte Ressource im Cache vorliegt oder extern angefordert werden muss. Wie in Abbildung 13 dargestellt, wirken sich falsch-positive Aussagen hier so aus, dass die URL im Cache gesucht, jedoch nicht gefunden wird und die Seite dann trotzdem aus dem Internet angefragt werden muss [20].

5.2 Sicherheitsanwendungen

5.2.1 Intrusion Detection. Auch im Bereich der Sicherheitsanwendungen gibt es für Bloomfilter Einsatzmöglichkeiten. So auch in Intrusion Detection Systemen. Hierfür werden bekannte Schadstrings, nach denen gesucht werden soll, in verschieden große Stücke unterteilt und je die Stücke gleicher Länge in den selben Filter gespeichert. Eintreffende Daten können dann ebenfalls in entsprechend große Teile zerlegt und diese auf Mitgliedschaft in den jeweiligen Filtern geprüft werden. Da falsch-negative Aussagen nicht möglich sind, werden alle bekannten Signaturen erkannt und als gefährlich markiert. Andersherum können durch die Falsch-Positiv-Rate aber auch unkritische Pakete als Gefährdung markiert werden. Darum werden alle so gekennzeichneten Pakete im Anschluss noch an

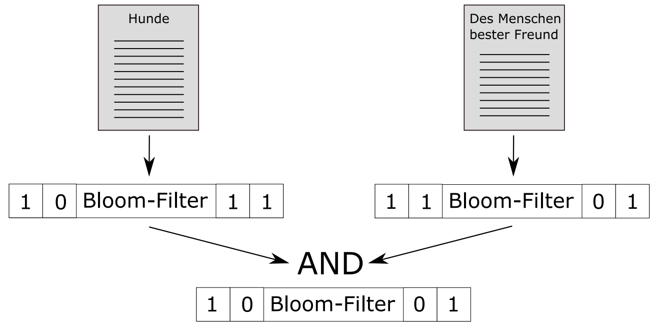

Abbildung 14: Analyse der Ähnlichkeit zweier Webseiten mittels Bloomfiltern

einen deterministischen Algorithmus weitergegeben, um sichere Aussagen treffen zu können [5].

5.2.2 Encrypted Search. Verschlüsseltes Suchen ermöglicht die sichere, kodierte Speicherung von Ressourcen auf Servern, ohne dass diese die Dateien entschlüsseln können müssen. Somit sind die Server und die auf ihnen gespeicherten Daten weniger anfällig für Diebstahl oder andere Hackerangriffe. Der Client, der eine Ressource auf dem Server speichern möchte, überträgt diese zunächst in einen Bloomfilter, verschlüsselt dann die Datei und sendet beides an den Server. Durch den zugehörigen Filter ist es nun immer noch möglich, die Ressourcen nach Schlagwörtern zu durchsuchen, ohne dass dem Server der eigentliche Inhalt bekannt ist. Ist die gesuchte Datei gefunden, wird sie an den Client gesendet, der diese wieder mit seinem Schlüssel dekodieren kann [19].

5.3 Weitere Anwendungen

Anhand der vorgestellten Anwendungsmöglichkeiten kann man sich leicht zahlreiche weitere Einsatzmöglichkeiten für Bloomfilter herleiten. Prinzipiell sind Bloomfilter immer dann geeignet, wenn große Mengen von Elementen schnell auf Mitgliedschaft überprüft werden müssen und seltene Falschaussagen verkraftbar sind.

5.3.1 Rechtschreibprüfung. Diese Anforderungen sind beispielsweise bei Rechtschreibprüfungen erfüllt, wo alle Wörter einer Sprache in einen großen Filter gespeichert werden können, um im Anschluss die in einem Dokument geschriebenen Wörter auf Mitgliedschaft zu testen. Durch die Falsch-Positiv-Rate werden so einige Fehler nicht gefunden, weshalb das Verfahren für bessere Ergebnisse um deterministische Algorithmen ergänzt werden müsste. Schnelle Analysen großer Texte sind für den Überblick aber möglich [12].

5.3.2 Ähnlichkeitsanalyse. Eine weitere vorstellbare Anwendung ist die Ähnlichkeitsanalyse von Dokumenten oder Webseiten. Hierfür wird der Inhalt in Bloomfiltern gespeichert und diese dann wie in Abbildung 14 mittels AND zusammengeführt. Bleiben viele Einsen übrig, liegt die Vermutung nahe, dass es eine inhaltliche Ähnlichkeit der beiden Seiten gibt, da gleiche Worte verwendet wurden [10].

6 ZUSAMMENFASSUNG

In dieser Seminararbeit habe ich die Funktionsweise des Bloomfilters wie er 1970 von Burton H. Bloom [4] vorgestellt wurde erläutert und bin auf dessen Stärken aber auch Schwächen eingegangen (siehe Abschnitt 2). Weiterhin habe ich Abwandlungen der Standardvariante vorgestellt, welche die Hauptprobleme der fehlenden Lösch-Operation und der eingeschränkten Erweiterbarkeit behandeln (siehe Abschnitt 3). Dabei habe ich ebenfalls eine selbst entwickelte Variante vorgestellt, deren Code zusammen mit einer Implementierung des Standard Bloomfilters auf GitHub unter folgendem Link zu finden ist: https://github.com/Florian2501/Seminar. Dieser Floomfilter bietet im Verhältnis zum benötigten Speicherplatz jedoch keine Verbesserung und sollte nicht verwendet werden (siehe Abschnitt 10). In Abschnitt 4 bin ich kurz auf alternative Datenstrukturen eingegangen und habe abschließend zahlreiche Anwendungsmöglichkeiten in Netzwerken, Sicherheitsanwendungen und weiteren Kontexten vorgestellt (siehe Abschnitt 5), was die Relevanz von Bloomfiltern noch einmal unterstreicht.

Wie man anhand meiner Quellen, deren Erscheinungsjahre sich breit über die letzten Jahrzehnte streuen, gut sehen kann, bleiben Bloomfilter bis heute konstant Forschungsgegenstand und es werden ständig neue Varianten entwickelt. Ich denke diese Entwicklung wird weiter anhalten. Insbesondere im Kontext von Ubiquitous Computing könnten Bloomfilter interessant werden, da auch hierbei große Datenmengen anfallen, wobei die Geschwindigkeit der Verarbeitung je nach Anwendungsfall gegenüber einer kleinen Fehlerrate überwiegen könnte. Da auch das Thema Datensicherheit in unserer digitalisierten Welt eine immer größere Rolle einnimmt, sollte auch in diesem Anwendungsfeld weiter geforscht werden.

LITERATUR

[1] 12.05.2022. Bloom Filter | Brilliant Math & Science Wiki. https://brilliant.org/wiki/bloom-filter/

[2] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom Filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261. https://doi.org/10.1016/j.ipl.2006.10.007

[3] Austin Appleby. 2008. Murmurhash 2.0. https://github.com/aappleby/smhasher

[4] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[5] Dharmapurikar S., Krishnamurthy P., Sproull T., and Lockwood J. 2003. Deep packet inspection using parallel Bloom filters. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.* 44–51. https://doi.org/10.1109/CONECT.2003.1231477

[6] Fabio Grandi. 2018. On the analysis of Bloom filters. *Inform. Process. Lett.* 129 (2018), 35–39. https://doi.org/10.1016/j.ipl.2017.09.004

[7] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[8] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 1998. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '98)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/285237.285287

[9] S. W. Graham. 1996. *Bh sequences.* Birkhäuser Boston, Boston, MA, 431–449. https://doi.org/10.1007/978-1-4612-4086-0_23

[10] Navendu Jain, Mike Dahlin, and Renu Tewar. 2005. Using Bloom Filters to Refine Web Search Results. In *Eighth International Workshop on the Web and Databases (WebDB '05)*. https://www.microsoft.com/en-us/research/publication/using-bloom-filters-refine-web-search-results/

[11] Laufer Rafael P., Velloso Pedro B., Cunha Daniel de O., Moraes Igor M., Bicudo Marco D.D., Moreira Marcelo D.D., and Duarte Otto Carlos M.B. 2007. Towards Stateless Single-Packet IP Traceback. In *32nd IEEE Conference on Local Computer Networks (LCN 2007)*. 548–555. https://doi.org/10.1109/LCN.2007.15

[12] Selvakumar Murugan, Tamil Arasan Bakthavatchalam, and Malaikannan Sankarasubbu. 2020. SymSpell and LSTM based Spell-Checkers for Tamil. (2020).

[13] Sabuzima Nayak and Ripon Patgiri. 2021. RobustBF: A High Accuracy and Memory Efficient 2D Bloom Filter. https://arxiv.org/pdf/2106.04365

[14] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. 2005. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05)*. Society for Industrial and Applied Mathematics, USA, 823–829.

[15] Ripon Patgiri, Sabuzima Nayak, and Samir Borgohain. 2018. Preventing DDoS using Bloom Filter: A Survey. *ICST Transactions on Scalable Information Systems* 5, 19 (2018), 155865. https://doi.org/10.4108/eai.19-6-2018.155865

[16] Rhea S.C. and Kubiatowicz J. 2002. Probabilistic location and routing. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 3. 1248–1257 vol.3. https://doi.org/10.1109/INFCOM.2002.1019375

[17] Christian Rothenberg, Carlos Macapuna, Fabio Verdi, and Mauricio Magalhaes. 2010. The deletable Bloom filter: a new member of the Bloom family. *IEEE Communications Letters* 14, 6 (2010), 557–559. https://doi.org/10.1109/LCOMM.2010.06.100344

[18] Rottenstreich Ori, Kanizo Yossi, and Keslassy Isaac. 2014. The Variable-Increment Counting Bloom Filter. *IEEE/ACM Transactions on Networking* 22, 4 (2014), 1092–1105. https://doi.org/10.1109/TNET.2013.2272604

[19] Saibal Kumar Pal and Puneet Sardana. 2012. BLOOM FILTERS & THEIR APPLICATIONS. *International Journal of Computer Applications Technology and Research* 1 (2012), 25–29.

[20] Jia Wang. 1999. A Survey of Web Caching Schemes for the Internet. *SIGCOMM Comput. Commun. Rev.* 29, 5 (1999), 36–46. https://doi.org/10.1145/505696.505701

[21] Whitaker A. and Wetherall D. 2002. Forwarding without loops in Icarus. In *2002 IEEE Open Architectures and Network Programming Proceedings. OPENARCH 2002 (Cat. No.02EX571)*. 63–75. https://doi.org/10.1109/OPNARC.2002.1019229