Bloom Filter

Jambura Anna, Pürstinger Kathrin,
Schnappauf Franziska, Thiele Coco

26. Februar 2026

Zusammenfassung

Bloom Filter sind probabilistische Datenstrukturen, die zur effizienten Lösung des Membership-Problems entwickelt wurden. Sie basieren auf einem Bit-Array und mehreren Hashfunktionen, wodurch eine sehr hohe Speicher- und Zeiteffizienz erreicht wird, jedoch mit einer geringen, konfigurierbaren Wahrscheinlichkeit für False-Positive-Ergebnisse. Die Fehlerrate hängt dabei von Parametern wie der Größe des Bit-Arrays, der Anzahl der Hashfunktionen und der Anzahl der gespeicherten Elemente ab und stellt einen Trade-off zwischen Genauigkeit und Speicherbedarf dar. Einschränkungen wie das fehlende Löschen von Elementen und die feste Größenplanung können durch Erweiterungen wie Counting Bloom Filter oder Scalable Bloom Filter verbessert werden, während der Cuckoo Filter eine alternative Lösung darstellt. Aufgrund ihrer Effizienz kommen Bloom Filter unter anderem in Bereichen wie Web-Caching und Datenbanksystemen zum Einsatz.

1 Grundlagen und Motivation

1.1 Das Membership-Problem

Seien ein beliebiges Element x und eine Menge S gegeben. Das Membership-Problem ist eine Bezeichnung für die Fragestellung: „Ist das Element x Teil der Menge S?“ Diese Frage tritt in vielen verschiedenen Bereichen und Anwendungen auf. Einige Beispiele dafür sind Datenbankenabfragen und URL-Caching in Web-Browsern. Klassische Ansätze, wie Listen, Hashtabellen oder Suchbäume liefern eine exakte Antwort auf die Frage, jedoch benötigen sie alle entsprechend viel Zeit und Speicherplatz. [1]

1.2 Lösungsansatz - Bloom Filter

Bloom Filter wurden 1970 von Burton H. Bloom entwickelt, um den hohen Ressourcenbedarf zu umgehen. Sie sind probabilistische Datenstrukturen, das bedeutet sie arbeiten mit Wahrscheinlichkeiten anstatt absoluter Sicherheit.

Dabei erlauben sie False-Positives in einem begrenzten Ausmaß. Ein Filter kann also fälschlicherweise melden, das Element x sei Teil der Menge S, auch wenn dies nicht der Fall ist. Umgekehrt sind False-Negatives jedoch ausgeschlossen. Wenn x tatsächlich ein Element von S ist, wird das der Filter immer korrekt erkennen. Mit anderen Worten: Ein vorhandenes Element wird nie als „nicht vorhanden“ gemeldet. [2]

1.3 Trade-off

Bloom Filter balancieren drei zentrale Faktoren. Neben der Reject-Time (Zeit zur Ablehnung von Nicht-Mitgliedern) und dem benötigten Speicherplatz, die auch in konventionellen Hashing-Methoden berücksichtigt werden müssen, wird hier auch die erlaubte Fehlerrate betrachtet. Der zentrale Trade-off ist dabei zwischen dem akzeptablen Anteil an False-Positives und der Speichereffizienz. Dieser ist bei der Implementierung eines Bloom Filters individuell konfigurierbar.

Durch die kontrollierte Fehlerwahrscheinlichkeit wird der Speicherbedarf bedeutend reduziert, da er nicht von der Länge der Daten abhängt, sondern immer gleich viele Bits pro Element beträgt. Je niedriger die Fehlerrate gewählt ist, desto mehr Bits pro Element werden benötigt. Bloom Filter sind besonders hilfreich, wenn die Mehrheit der Anfragen nicht-existente Elemente betrifft – hier liefern sie schnell ein definitives „Nein“ auf die Membership-Frage. [1]

2 Funktionsweise und Mathematische Grundlagen

2.1 Aufbau

Der Bloom Filter besteht auf einem m-stelligen Bitarray, welches initial mit Nullen befüllt wird. Weiters werden k unabhängige Hashfunktionen definiert. Diese verwendet man um die Elemente der gewünschten Menge zu hashen. Abhängig von ihrem Hashwert werden die Elemente dann an der entsprechenden Position im Array eingefügt. Um also jedes Element erfolgreich einzufügen, muss die Hashfunktion $\bmod m$ angewandt werden. Somit erreicht man die Indizes 0 bis $m-1$. [2]

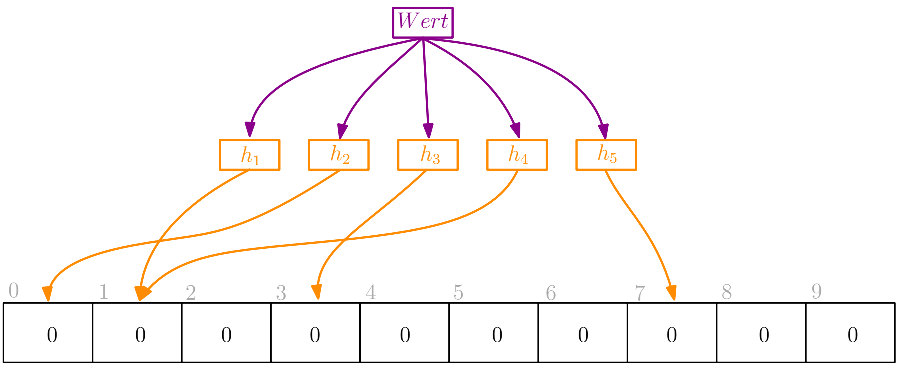

Abbildung 1: Visualisierung eines Bloom Filters

Da die Hashfunktionen keinem Sicherheitsstandard entsprechen, müssen keine kryptographischen Eigenschaften gelten. Kryptographische Eigenschaften bedeutet, minimale Eingabeänderungen müssen zu einer maximalen Änderung des Hashwerts führen. Die Eingabe darf nicht mittels der Hashfunktion wiederhergestellt werden können und zwei Eingaben haben fast unmöglich den selben Hashwert.
Für Bloom Filter verwendet man schnelle und einfache Hashfunktionen, da die Effizienz im Vordergrund steht.

2.2 Einfügen/Suchen

Einfügen
Eine Menge S wird nun wie folgt in einem Bloom Filter eingefügt:
Für jedes Element $x \in S$ werden die Hash Werte aller k Hash Funktionen berechnet. Nun wird an diesen Positionen im Array die 0 auf eine 1 gesetzt. Sollte an einer dieser Positionen bereits eine 1 stehen, wird dies ignoriert. Dieser Vorgang wird für alle n Elemente der Menge S wiederholt.

2.2.1 Beispiel Einfügen

Betrachte folgende Menge $S = \{2, 4, 9\}$ und einen Bloom Filter der Länge $m = 10$ mit $k = 3$ Hashfunktionen.
Als beispielhafte Hashfunktionen verwenden wir: $h_1(x) = x \bmod 10$ $h_2(x) = (2x+3) \bmod 10$ und $h_3(x) = (3x + 7) \bmod 10$.
Nun berechnen wir die Hashwerte für jedes Element der Menge S:

- Für $x = 2$:

$$\begin{aligned} h_1(2) &= 2 \bmod 10 = 2 \\ h_2(2) &= (2 \cdot 2 + 3) \bmod 10 = 7 \\ h_3(2) &= (3 \cdot 2 + 7) \bmod 10 = 3 \end{aligned}$$

- Für $x = 4$:

$$\begin{aligned} h_1(4) &= 4 \bmod 10 = 4 \\ h_2(4) &= (2 \cdot 4 + 3) \bmod 10 = 1 \\ h_3(4) &= (3 \cdot 4 + 7) \bmod 10 = 9 \end{aligned}$$

- Für $x = 9$:

$$\begin{aligned} h_1(9) &= 9 \bmod 10 = 9 \\ h_2(9) &= (2 \cdot 9 + 3) \bmod 10 = 1 \\ h_3(9) &= (3 \cdot 9 + 7) \bmod 10 = 4 \end{aligned}$$

Nun fügt man die Elemente in den Bloom Filter ein. Für das erste Element 2 werden die Positionen 2, 7 und 3 auf 1 gesetzt. Daraus resultiert der folgende Bloom Filter:

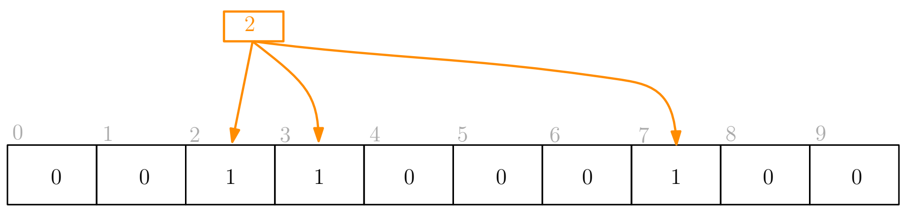

Abbildung 2: Bloom Filter nach Einfügen des Elements 2

Für das zweite Element 4 werden die Positionen 4, 1 und 9 auf 1 gesetzt.

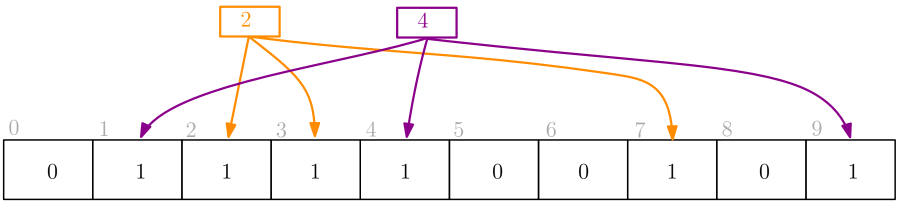

Abbildung 3: Bloom Filter nach Einfügen des Elements 4

Für das dritte Element 9 werden die Positionen 9, 1 und 4 auf 1 gesetzt. Da die Positionen 1, 4 und 9 bereits auf 1 gesetzt wurden, ändert sich der Bloom Filter nicht weiter.

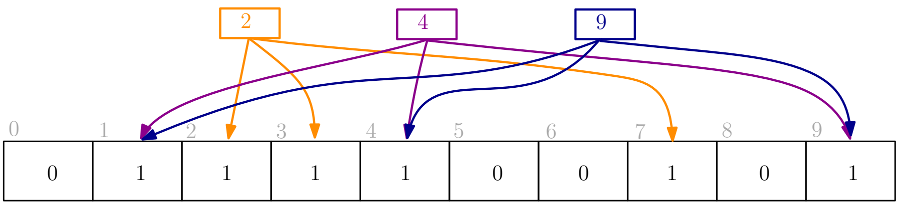

Abbildung 4: Bloom Filter nach Einfügen des Elements 9

Suchen

Um ein Element x in einem Bloom Filter zu suchen, werden dieselben Hashfunktionen wie beim Einfügen verwendet. Die Hashwerte werden berechnet und an den entsprechenden Positionen im Array geprüft. Wenn alle Positionen auf 1 gesetzt sind, so ist das Element wahrscheinlich in der Menge enthalten. Wenn mindestens eine Position auf 0 gesetzt ist, so ist das Element sicher nicht in der Menge enthalten. [2]

2.2.2 Beispiel Suchen

Betrachten wir den zuvor erstellten Bloom Filter und suchen nach dem Element 4. Berechnen wir die Hashwerte für 4:

$$h_1(4) = 4 \bmod 10 = 4$$
$$h_2(4) = (2 \cdot 4 + 3) \bmod 10 = 1$$
$$h_3(4) = (3 \cdot 4 + 7) \bmod 10 = 9$$

Nun prüfen wir die Positionen 4, 1 und 9 im Bloom Filter. Alle drei Positionen sind auf 1 gesetzt, daher ist das Element 4 wahrscheinlich in der Menge enthalten.

Betrachten wir nun das Element 5 und berechnen die Hashwerte:

$$
\begin{aligned}
h_1(5) &= 5 \bmod 10 = 5 \\
h_2(5) &= (2 \cdot 5 + 3) \bmod 10 = 3 \\
h_3(5) &= (3 \cdot 5 + 7) \bmod 10 = 2
\end{aligned}
$$

Nun prüfen wir die Positionen 5, 3 und 2 im Bloom Filter. Die Position 2 ist auf 0 gesetzt, daher ist das Element 5 sicher nicht in der Menge enthalten.

Ein wichtiger Aspekt des Bloom Filters ist, dass er fälschlicherweise angeben kann, dass ein Element in der Menge enthalten ist, obwohl es tatsächlich nicht vorhanden ist. Dies wird als *False Positive* bezeichnet. Wenn alle Positionen, die durch die Hashfunktionen eines Elements angegeben werden, auf 1 gesetzt sind, obwohl das Element nicht in der Menge enthalten ist, führt dies zu einem False Positive. Ein Beispiel hierfür wäre das Element 12:

$$
\begin{aligned}
h_1(12) &= 12 \bmod 10 = 2 \\
h_2(12) &= (2 \cdot 12 + 3) \bmod 10 = 7 \\
h_3(12) &= (3 \cdot 12 + 7) \bmod 10 = 3
\end{aligned}
$$

Die Positionen 2, 7 und 3 sind alle auf 1 gesetzt, obwohl das Element 12 nicht in der Menge enthalten ist. Daher würde der Bloom Filter fälschlicherweise angeben, dass 12 in der Menge enthalten ist.

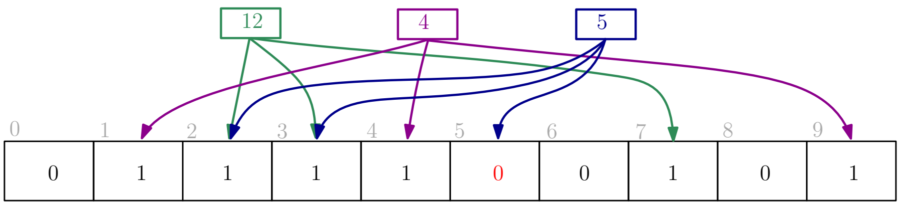

Abbildung 5: Suchen nach den Elementen 4, 5 und 12 im Bloom Filter

2.3 Formeln zur Evaluierung

2.3.1 False Positive Probability

Zur Erstellung des optimalen Bloom Filter ist es wichtig, die False Positive Probability (FPP) zu berechnen. Diese gibt an, wie wahrscheinlich es ist, dass der Bloom Filter fälschlicherweise angibt, dass ein Element in der Menge enthalten ist, obwohl es tatsächlich nicht vorhanden ist. Laut [2] entsteht die Formel zur Berechnung aus folgenden Komponenten:

Unter der Annahme, dass die Hashfunktionen unabhängig und gleichverteilt sind, ergibt sich die Wahrscheinlichekeit dass ein bestimmtes der m Bits nicht gesetzt ist durch:

$$1 - \frac{1}{m} \tag{1}$$

Weiters werden nun die k Hashfunktionen mitbetrachtet, immer noch für den Fall, dass ein bestimmtes Bit nicht gesetzt ist.

$$\left(1 - \frac{1}{m}\right)^k \tag{2}$$

Nun werden die n Elemente der Menge S betrachtet, welche in den Bloom Filter eingefügt werden.

$$\left(1 - \frac{1}{m}\right)^{kn} \tag{3}$$

Die Wahrscheinlichkeit, dass ein bestimmtes Bit auf 1 gesetzt ist, ergibt sich aus der Gegenwahrscheinlichkeit:

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \tag{4}$$

Da es bei Bloom Filtern um Membership-Tests geht, muss die Wahrscheinlichkeit berechnet werden, dass alle k Positionen eines Elements auf 1 gesetzt sind, obwohl das Element nicht in der Menge enthalten ist.

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \tag{5}$$

Aus der Formel lässt sich schließen, dass je **größer** m gewählt wird, desto **kleiner** wird die False Positive Probability. Je **größer** n gewählt wird, desto **größer** wird die False Positive Probability.

Da die False Positive Probability so klein wie möglich gehalten werden soll, ist auch die Wahl der Anzahl Hashfunktionen von großer Bedeutung. Setzt man die Formel für die False Positive Probability gleich 0 und löst sie nach k auf, erhält man die optimale Anzahl an Hashfunktionen:

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n} \tag{6}$$

3 Pseudocode und Implementierung

Ein Bloom Filter lässt sich mit drei grundlegenden Operationen beschreiben: Initialisierung, Einfügen und Abfragen.

3.1 Initialisierung

Bei der Initialisierung werden alle m Bits im Array auf 0 gesetzt und k Hash-Funktionen festgelegt. Je kleiner die gewünschte Fehlerrate sein soll, desto größer muss m gewählt werden.

Algorithm 1 Initialisierung eines Bloom Filters

```
Erzeuge Bit-Array B[0 ... m − 1] und setze alle Bits auf 0
Definiere Hash-Funktionen h_1, h_2, ..., h_k
```

3.2 Einfügen

Beim Einfügen wird für jedes Element x eine Schleife genau k-mal ausgeführt. In jeder Iteration wird mithilfe der jeweiligen Hash-Funktion h_i ein Index berechnet und das entsprechende Bit im Bitarray auf 1 gesetzt. Der Modulo-Operator stellt sicher, dass der berechnete Index immer innerhalb des gültigen Bereichs von 0 bis $m - 1$ liegt. [2]

Algorithm 2 Einfügen eines Elements x

```
for i = 1 to k do
    index ← h_i(x) mod m
    B[index] ← 1
end for
```

3.3 Abfragen

Für eine Abfrage werden dieselben k Hash-Werte berechnet und die entsprechenden Positionen im Array überprüft. Existiert mindestens ein Bit mit dem Wert 0, kann man mit absoluter Sicherheit sagen, dass das Element nicht enthalten ist – es gibt keine False Negatives. Sind hingegen alle k Bits gleich 1, gilt das Element als wahrscheinlich enthalten. Diese probabilistische Aussage ist das zentrale Merkmal des Bloom-Filters: Es sind False Positives möglich.

Algorithm 3 Abfrage eines Elements x

```
for i = 1 to k do
    index ← h_i(x) mod m
    if B[index] = 0 then
        return FALSE
    end if
end for
return TRUE
```

4 Komplexitätsanalyse

4.1 Zeitkomplexität

Sowohl das Einfügen als auch das Abfragen eines Elements haben eine Zeitkomplexität von $\mathcal{O}(k)$, wobei k die Anzahl der Hash-Funktionen bezeichnet. Entscheidend ist dabei, dass diese Zeit *unabhängig* von der Anzahl n der bereits im Filter gespeicherten Elemente ist. Der Grund dafür liegt in der Struktur des Filters: Es werden keine Elemente explizit gespeichert, sondern lediglich Bits in einem Array der Größe m gesetzt oder gelesen. Egal ob sich 1.000 oder 100 Millionen Elemente im Filter befinden – die Abfragezeit bleibt konstant [1].

4.2 Speicherkomplexität

Die Speicherkomplexität beträgt $\mathcal{O}(m)$, wobei m die Größe des Bit-Arrays ist. Im Gegensatz zu klassischen Datenstrukturen hängt dieser Speicherbedarf *nicht* von der Größe der gespeicherten Elemente ab, sondern nur von zwei Faktoren: der Anzahl der zu speichernden Elemente n und der akzeptierten False-Positive-Rate ε [1].

Als praktische Faustregel gilt: Bei einer Fehlerrate von etwa 1 % benötigt ein Bloom Filter weniger als 10 Bits pro Element. Das ist bemerkenswert effizient – unabhängig davon, ob es sich bei den Elementen um kurze Zeichenketten oder lange URLs handelt [3].

4.3 Vergleich mit anderen Datenstrukturen

Tabelle 1 stellt die Komplexitätseigenschaften des Bloom Filters denen einer Hash-Tabelle mit verketteten Listen sowie eines balancierten Baums gegenüber.

Die **Hash-Tabelle mit verketteten Listen** erreicht im Durchschnitt $\mathcal{O}(1)$ für Einfüge- und Suchoperationen, kann im schlechtesten Fall jedoch auf $\mathcal{O}(n)$ anwachsen. Da jedes Element explizit gespeichert wird, beträgt der Speicherbedarf $\mathcal{O}(n)$ – typischerweise 64 Bits oder mehr pro Element (Nutzdaten plus Pointer). Der wesentliche Vorteil liegt

Eigenschaft	Bloom Filter	Hash-Tabelle mit Chaining	Balancierter Baum
Zeitkomplexität	$\mathcal{O}(k)$	$\varnothing\ \mathcal{O}(1)$, worst $\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Speicherkomplexität	$\mathcal{O}(m)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Genauigkeit	Probabilistisch	Exakt	Exakt

Tabelle 1: Vergleich ausgewählter Datenstrukturen

in der Exaktheit: Es gibt keine False Positives, und Elemente können jederzeit wieder abgerufen werden [4].

Der **balancierte Baum** hat eine Zeitkomplexität von $\mathcal{O}(\log n)$ für Suche und Einfügen. Die Laufzeit steigt mit wachsender Elementanzahl langsam an, da bei jedem Schritt etwa die Hälfte der verbleibenden Elemente verworfen wird. Der Speicherbedarf ist ebenfalls $\mathcal{O}(n)$. Der Vorteil liegt in der Möglichkeit, Elemente geordnet zu speichern, was zusätzliche Operationen wie Bereichsabfragen erlaubt [4].

4.4 Speichereffizienz in der Praxis

Um die Speicherersparnis greifbar zu machen, betrachten wir ein konkretes Beispiel: Für 100 Millionen URLs benötigt ein Bloom Filter bei einer Fehlerrate von 1 % rund 120 Megabyte. Eine Hash-Tabelle mit denselben Einträgen würde hingegen über ein Gigabyte beanspruchen. Das ist nicht nur ein quantitativer, sondern oft ein qualitativer Unterschied – nämlich der zwischen einem System, das auf einem Endgerät lauffähig ist, und einem, das einen dedizierten Server erfordert.

Dieser enorme Vorteil hat allerdings seinen Preis: Ein Bloom Filter beantwortet ausschließlich die Frage *„Ist das Element möglicherweise in der Menge?“* Er kann weder Elemente aufzählen noch löschen, noch gibt er die Elemente selbst zurück [2].

5 Probleme von Bloom Filtern und Lösungen

5.1 Das Löschen von Elementen

Der klassische Bloom Filter besitzt unter anderem die Einschränkung, dass er das Löschen von Elementen nicht unterstützt. Möchte man ein Element entfernen, liegt es zunächst nahe, die entsprechenden Bits im Bit-Array wieder auf 0 zu setzen. Genau hier entsteht jedoch ein fundamentales Problem. Mehrere Elemente können auf dieselbe Position im Bit-Array hashen. Wird ein Bit zurückgesetzt, entfernt man daher nicht nur das gewünschte Element, sondern gleichzeitig auch alle anderen Elemente, die an dieser Position gespeichert wurden. Das eigentliche Element ist zwar entfernt, aber andere Elemente gelten nun ebenfalls als nicht mehr vorhanden, obwohl sie eigentlich noch im Filter sein sollten.

Eine Lösung für dieses Problem ist der Counting Bloom Filter.[5] Das Grundprinzip beim Einfügen bleibt dabei gleich wie beim klassischen Bloom Filter. Der Unterschied besteht darin, dass man an jeder Position nicht nur ein einzelnes Bit speichert, sondern einen kleinen Zähler. Dieser Zähler wird beim Einfügen eines Elements um 1 erhöht und beim Löschen wieder um 1 verringert. Typischerweise sind diese Zähler 4 Bit groß und können somit Werte von 0 bis 15 speichern. Dadurch wird es möglich, Elemente sicher zu löschen, ohne andere Einträge unbeabsichtigt zu beeinflussen.

Allerdings hat der Counting Bloom Filter auch Nachteile. Der Speicherverbrauch ist deutlich höher, da statt eines einzelnen Bits nun 4 Bits pro Position benötigt werden. Bei gleicher Genauigkeit benötigt ein Counting Bloom Filter somit ungefähr das Drei- bis Vierfache an Speicher im Vergleich zum klassischen Bloom Filter. Außerdem können die Zähler überlaufen. Wenn mehr als 15 Elemente auf dieselbe Position hashen, reichen 4 Bits nicht mehr aus. Man könnte größere Zähler verwenden, allerdings würde das den Speicherbedarf weiter erhöhen. Der Counting Bloom Filter eignet sich daher besonders dann, wenn häufig gelöscht werden muss - man bezahlt diese Möglichkeit jedoch mit einem deutlich höheren Speicherverbrauch. An Verbesserungen wird zwar gearbeitet, doch eine genauere Betrachtung würde den Rahmen dieser Arbeit sprengen.[6]

5.2 Größenplanung

Ein weiteres grundlegendes Problem klassischer Bloom Filter ist die Größenplanung. In der Regel muss man vorher festlegen, wie groß der Filter sein soll. Ist er zu klein dimensioniert, steigt die Fehlerwahrscheinlichkeit stark an. Die Bits werden sehr schnell gesetzt und die False-Positive-Rate nimmt deutlich zu. Ist der Filter hingegen zu groß gewählt, wird Speicherplatz verschwendet, da möglicherweise Kapazitäten reserviert werden, die nie vollständig genutzt werden.

Der Scalable Bloom Filter bietet hier eine Lösung durch dynamisches Wachstum.[5] Er besteht aus mehreren klassischen Bloom Filtern, die nacheinander erstellt werden. Sobald ein Filter eine bestimmte Auslastung erreicht, wird ein neuer, größerer Filter mit einer strengeren Fehlerrate hinzugefügt. Auf diese Weise bleibt die Gesamtfehlerwahrscheinlichkeit über alle Filter hinweg kontrollierbar. Selbst wenn mehrere Filter hinzukommen, bleibt die kombinierte Fehlerrate in akzeptablen Grenzen. Der große Vorteil ist, dass der Filter beliebig wachsen kann, ohne komplett neu aufgebaut werden zu müssen. Ein Nachteil ist jedoch, dass Abfragen mit jedem zusätzlichen Filter etwas langsamer werden, da mehrere Filter überprüft werden müssen. Neben Counting- und Scalable-Varianten gibt es noch viele weitere spezielle Varianten von Bloom Filtern, die jedoch den Rahmen dieser Arbeit überschreiten würden.[7]

6 Cuckoo Filter

Eine alternative Datenstruktur stellt der Cuckoo Filter dar. Hierbei handelt es sich nicht merh wirklich um einen Bloom Filter, dennoch verfolgt er dasselbe Ziel: speichereffiziente Mengenabfragen bei geringen Fehlerraten. Der Cuckoo Filter basiert nicht auf einem Bit-Array, sondern auf einer Hash-Tabelle mit kleinen Fächern, sogenannten Buckets. In diesen Buckets werden Fingerabdrücke, sogenannte Fingerprints, gespeichert. Das sind kurze, eindeutige Kennungen der Elemente mit nur wenigen Bits Länge.

Beim Einfügen eines Elements wird mithilfe einer Hash-Funktion berechnet, in welches Fach es gehört. Jedes Element besitzt dabei genau zwei mögliche Buckets, in denen es abgelegt werden kann. Ist in einem dieser Buckets noch Platz vorhanden, wird der Fingerprint dort gespeichert. Sind jedoch beide Buckets belegt, greift das sogenannte Cuckoo-Prinzip. Hier verdrängt das neue Element einen bestehenden Eintrag aus einem der beiden Buckets. Das verdrängte Element muss sich anschließend einen neuen Platz in seinem alternativen Bucket suchen. Dieser Prozess kann sich fortsetzen, bis schließlich alle Elemente einen Platz gefunden haben.

Der Cuckoo Filter bringt sowohl Vorteile als auch Nachteile mit sich. Ein großer Vorteil ist, dass Elemente problemlos gelöscht werden können, da die Fingerprints direkt gespeichert sind und gezielt entfernt werden können. Außerdem sind Abfragen sehr schnell, da nur zwei Buckets geprüft werden müssen. Ein Nachteil zeigt sich bei sehr hoher Auslastung der Hash-Tabelle. In solchen Fällen kann die Verdrängungskette sehr lang werden, ohne dass ein freier Platz gefunden wird. Dann muss die gesamte Struktur vergrößert werden. Studien zeigen jedoch, dass Cuckoo Filter in vielen realen Anwendungen praktisch besser abschneiden als klassische Bloom Filter.[8] Klassische Bloom Filter sind dennoch besonders sinnvoll, wenn sehr große Datenmengen verarbeitet werden, der verfügbare Speicher knapp oder teuer ist, kleine Fehlerraten akzeptiert werden können und sie als Vorfilter von aufwendigen oder rechenintensiven Operationen eingesetzt werden.[4]

7 Anwendungsbeispiele

7.1 Web-Proxy-Caching

In verteilten Netzwerken arbeiten mehrere Proxy-Server zusammen und tauschen sich untereinander aus. Bei einer Anfrage nach einer Webseite sucht ein Proxy zunächst im eigenen Cache, ob er diese bereits gespeichert hat. Wenn das nicht der Fall ist, spricht man von einem Cache-Miss und es wird geprüft, ob sich die Webseite im Cache eines anderen Proxys befindet. Wird sie hier gefunden, wird die Anfrage an den entsprechenden Proxy weitergeleitet, anstatt die Seite direkt aus dem Web zu laden.

Damit dieses System funktioniert, muss jeder Proxy über den Inhalt der Caches aller anderen Proxies Bescheid wissen. Um den enormen Netzwerkverkehr, der beim wiederholten Austausch der kompletten URL-Listen entstehen würde, zu vermeiden, kommen hier Bloom Filter zum Einsatz. Im Summary Cache Protokoll tauschen Proxies periodisch Bloom Filter untereinander aus, die den Inhalt ihres Caches zusammenfassen. Wenn nun ein Cache-Miss auftritt, werden die Bloom Filter jener anderen Proxies konsultiert, die ein positives Ergebnis versprechen und die Anfrage wird entsprechend weitergeleitet.

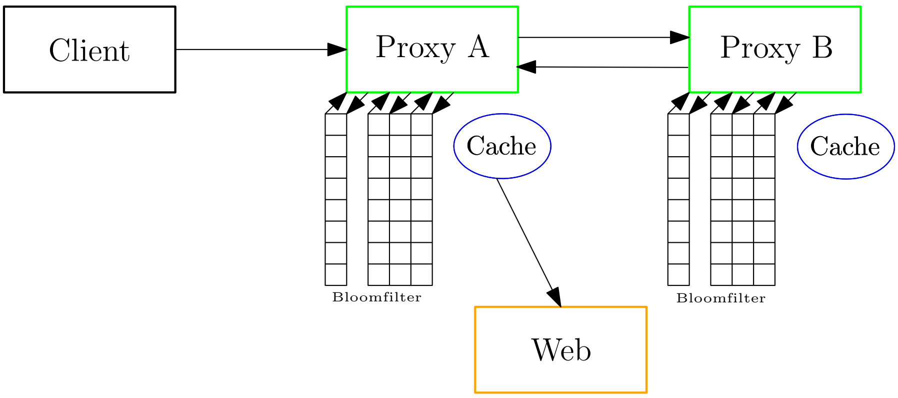

Abbildung 6: Web-Proxy-Caching mit Bloom Filtern

Hierbei können False-Positives auftreten, wobei es dann zu einer minimalen Verzögerung kommt. Die massive Reduktion des Netzwerkverkehrs durch den Bloom Filter überwiegt diesen Nachteil bei Weitem. Das Summary Cache Protokoll wird beispielsweise im Web-Proxy-Cache „Squid“ eingesetzt. [4]

7.2 Google Bigtable

Bloom Filter werden oft in Datenbanksystemen verwendet, wobei Google Bigtable ein bekanntes Beispiel hierfür ist. Bigtable speichert die Daten auf der Festplatte in Sorted-String-Tables (SSTables). Wenn eine Leseoperation durchgeführt werden soll, müssen potenziell mehrere dieser Tables durchsucht werden, bis die gewünschten Daten gefunden werden. Da jede Table auf der Festplatte liegt, verursacht jeder Zugriff auf eine SSTable auch einen teuren Festplattenzugriff. Besonders problematisch im Bezug auf die benötigten Ressourcen wird dies bei Abfragen nach nicht-existenten Daten.

Kommen jetzt die Bloom Filter zum Einsatz, ändert sich dies drastisch. Für jede SSTable wird ein Bloom Filter im Hauptspeicher gehalten, der Auskunft über deren Inhalt gibt. Vor einem Festplattenzugriff wird also der Filter befragt, ob die gesuchten Daten in der Table enthalten sind. Bei einem positiven Ergebnis wird der Zugriff durchgeführt, ansonsten kann er eingespart werden. [9]

7.2.1 Beispiel Anfrage

Angenommen es wird eine Anfrage auf den Schlüssel X gestellt und auf der Festplatte liegen drei SSTables. Ohne Verwendung von Bloom Filtern müssten alle drei Tables abgerufen und durchsucht werden, also drei Festplattenzugriffe durchgeführt werden. Unter Einsatz von Bloom Filtern werden jedoch zuerst diese konsultiert. Die ersten beiden Filter könnten melden, dass Schlüssel X jeweils nicht in SSTable 1 bzw. SSTable 2 liegt, sie können also beide übersprungen werden. Filter 3 sagt jetzt, dass sich X in Table 3 befinden könnte – dieser Zugriff wird durchgeführt. Demzufolge wurde nur ein Festplattenzugriff durchgeführt, bis der gesuchte Schlüssel X gefunden wurde, das bedeutet eine Ersparnis von zwei Zugriffen durch die Verwendung von Bloom Filtern.

7.3 Weitere Anwendungen

Heute kommen Bloom Filter in zahlreichen Systemen zum Einsatz. Google Chrome nutzt sie beispielsweise für Safe-Browsing zur Malware-Erkennung. [10] Neben Google Bigtable setzen auch weitere Datenbanksysteme, wie Apache Cassandra auf die Vorteile von Bloom Filtern, um unnötige Festplattenzugriffe zu vermeiden. [11]

Literatur

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

[2] Sasu Tarkome, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2012.

[3] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8 (3):281–293, 2000.

[4] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[5] Paul A. Gagniuc, Ionel-Bujorel Păvăloiu, and Maria-Iuliana Dascălu. Bloom filters at fifty: From probabilistic foundations to modern engineering and applications. *Algorithms*, 18:1–15, 2025.

[6] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Proceedings of the 14th conference on Annual European Symposium on Algorithms*, pages 684–695. Springer, 2006.

[7] Paulo S. Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[8] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies*, pages 179–190. ACM, 2014.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26 (2), 2008.

[10] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. (un)safe browsing. Technical Report RR-8594, INRIA, 2010.

[11] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.