Bloom Filter

Jambura Anna, Pürstinger Kathrin,
Schnappauf Franziska, Thiele Coco

24. Februar 2026

Zusammenfassung

passend auszufüllen

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. [Blo70]

1.2 Lösungsansatz - Bloomfilter

Bloomfilter 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. [TRL12]

1.3 Trade-off

Bloomfilter 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 Bloomfilters 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. Bloomfilter sind besonders hilfreich, wenn die Mehrheit der Anfragen nicht-existente Elemente betrifft – hier liefern sie schnell ein definitives „Nein“ auf die Membership-Frage. [Blo70]

2 Funktionsweise und Mathematische Grundlagen

2.1 Aufbau

Der Bloomfilter 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 Hasfunktion mod$m$ angewandt werden. Somit erreicht man die Indizes 0 bis $m-1$. [TRL12]

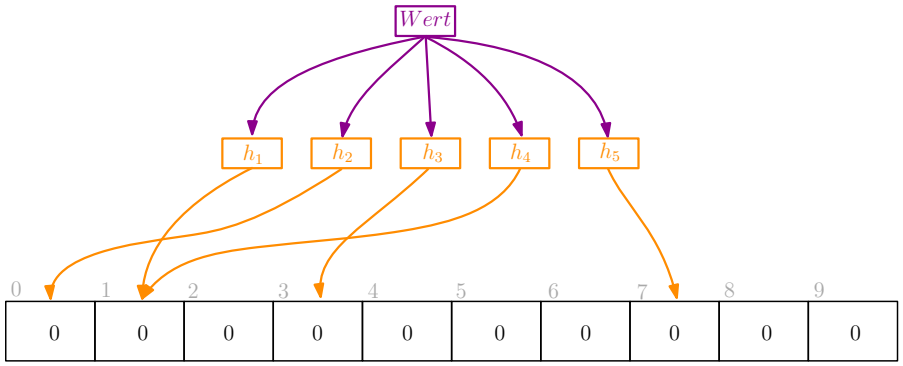

Abbildung 1: Visualisierung eines Bloomfilters

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 Bloomfilter 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 Bloomfilter 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 Bloomfilter 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 $x = 2$:

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

- Für $x = 4$:

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

- Für $x = 9$:

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

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

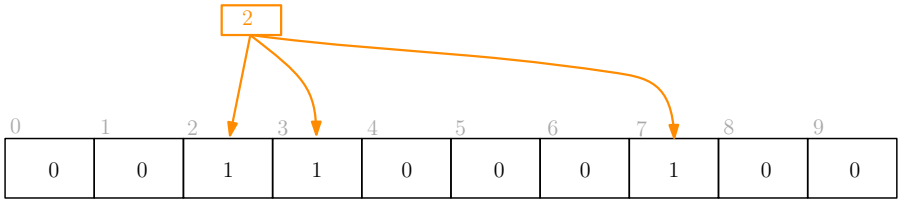

Abbildung 2: Bloomfilter nach Einfügen des Elements 2

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

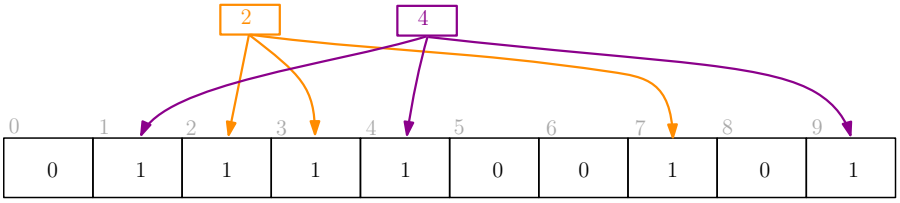

Abbildung 3: Bloomfilter nach Einfügen des Elements 4

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 Bloomfilter nicht weiter.

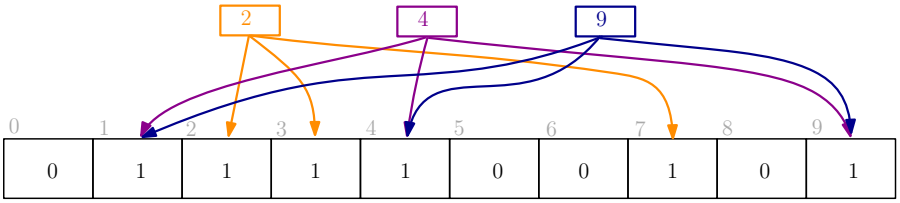

Abbildung 4: Bloomfilter nach Einfügen des Elements 9

Suchen

Um ein Element x in einem Bloomfilter 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. [TRL12]

2.2.2 Beispiel Suchen

Betrachten wir den zuvor erstellten Bloomfilter 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 Bloomfilter. 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:

$$h_1(5) = 5 \bmod 10 = 5$$
$$h_2(5) = (2 \cdot 5 + 3) \bmod 10 = 3$$
$$h_3(5) = (3 \cdot 5 + 7) \bmod 10 = 2$$

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

Ein wichtiger Aspekt des Bloomfilters 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:

$$h_1(12) = 12 \bmod 10 = 2$$
$$h_2(12) = (2 \cdot 12 + 3) \bmod 10 = 7$$
$$h_3(12) = (3 \cdot 12 + 7) \bmod 10 = 3$$

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

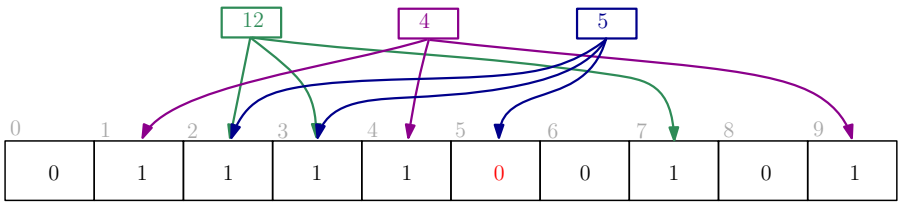

Abbildung 5: Suchen nach den Elementen 4, 5 und 12 im Bloomfilter

2.3 Formeln zur Evaluierung

2.3.1 False Positive Probability

Zur Erstellung des optimalen Bloomfilter ist es wichtig, die False Positive Probability (FPP) zu berechnen. Diese gibt an, wie wahrscheinlich es ist, dass der Bloomfilter fälschlicherweise angibt, dass ein Element in der Menge enthalten ist, obwohl es tatsächlich nicht vorhanden ist. Laut [TRL12] 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 Bloomfilter 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 Bloomfiltern 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 Anwendungsbeispiele

3.1 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 Bloomfilter zum Einsatz. Im Summary Cache Protokoll tauschen Proxies periodisch Bloomfilter untereinander aus, die den Inhalt ihres Caches zusammenfassen. Wenn nun ein Cache-Miss auftritt, werden die Bloomfilter jener anderen Proxies konsultiert, die ein positives Ergebnis versprechen und die Anfrage wird entsprechend weitergeleitet.

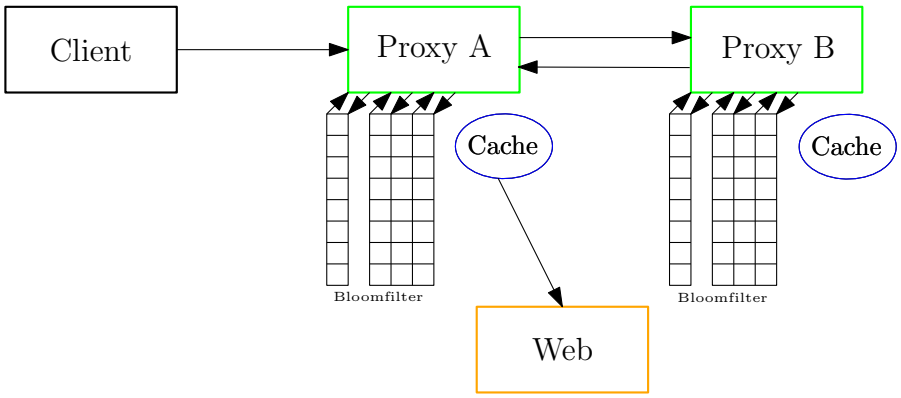

Abbildung 6: Web-Proxy-Caching mit Bloomfiltern

Hierbei können False-Positives auftreten, wobei es dann zu einer minimalen Verzögerung kommt. Die massive Reduktion des Netzwerkverkehrs durch den Bloomfilter überwiegt diesen Nachteil bei Weitem. Das Summary Cache Protokoll wird beispielsweise im Web-Proxy-Cache „Squid“ eingesetzt. [BM04]

3.2 Google Bigtable

Bloomfilter 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 Bloomfilter zum Einsatz, ändert sich dies drastisch. Für jede SSTable wird ein Bloomfilter 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. [CDG+08]

3.2.1 Beispiel Anfrage

Angenommen es wird eine Anfrage auf den Schlüssel X gestellt und auf der Festplatte liegen drei SSTables. Ohne Verwendung von Bloomfiltern müssten alle drei Tables abgerufen und durchsucht werden, also drei Festplattenzugriffe durchgeführt werden. Unter Einsatz von Bloomfiltern 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 Bloomfiltern.

3.3 Weitere Anwendungen

Heute kommen Bloomfilter in zahlreichen Systemen zum Einsatz. Google Chrome nutzt sie beispielsweise für Safe-Browsing zur Malware-Erkennung. [GKL10] Neben Google Bigtable setzen auch weitere Datenbanksysteme, wie Apache Cassandra auf die Vorteile von Bloomfiltern, um unnötige Festplattenzugriffe zu vermeiden. [LM10]

Literatur

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BM04] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[CDG+08] 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.

[GKL10] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. (un)safe browsing. Technical Report RR-8594, INRIA, 2010.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[TRL12] 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.