Bonusaufgabe Bloom-Filter

Severin Peyer, Daniel Obrist

December 7, 2019

1 Idee des Bloom-Filters

Der Bloom-Filter ist eine pobabilistische Datenstruktur um schnell und speichereffizient zu bestimmen, ob ein Element in einem Set vorhanden ist. Er kann uns sagen, ob ein Element entweder definitiv nicht im Set ist oder sich mit einer gewissen Fehlerwahrscheinlichkeit im Set befinden kann.

Ein Bloom-Filter ist ein Array von m Bits, die anfangs alle auf 0/false gesetzt sind. Wird ein Element in das Set hinzugefügt, werden durch k unterschiedliche Hash-Funktionen Hashcodes berechnet. Diese Hashcodes bestimmmen, welche Stellen des Bloomfilters auf 1/true gesetzt werden. So entsteht eine Art Fingerabdruck aller hinzugefügten Daten. Beim Überprüfen ob ein Element enhalten ist werden ebenfalls die Hashcodes des Elements berechnet und mit dem Bloom-Filter verglichen. Wenn an einer der Positionen 0/false steht, ist das Element definitiv nicht im Set. Es können aber False-Positives entstehen, wenn per Zufall zwei Elemente die gleichen Hashcodes generieren. Je mehr Elemente zur Menge hinzugefügt wurden, desto grösser wird diese Fehlerwahrscheinlichkeit p, dass man aus Versehen eine positive Antwort bekommt, obwohl das Element gar nicht im Set enthalten ist.

Vorteile

- Zeit, um Elemente hinzuzufügen oder zu überprüfen, ob ein Element im Set ist konstant, O(k)
- Benötigen den eigentlichen Speicher/die Daten gar nicht
- Fehlerwahrscheinlichkeit kann festgelegt werden und durch mehrere Hash-Funktionen oder die Filtergrösse gesenkt werden

Nachteile

- Kann False-Positives liefern
- Eigentliche Daten müssen separat abgespeichert werden
- Elemente entfernen ist mühsam

2 Praxisbeispiele

Beispiele sind NoSQL-Datenbanken wie BigTable, Apache HBase und Apache Cassandra. Diese Datenbanken verwenden Bloom-Filter für das Nachschlagen eines Schlüssels in einer grösseren Tabelle. Der Bloom-Filter wird dabei der eigentlichen Suche vorgeschaltet um unnötige Suchvorgänge einzusparen. Nur wenn der Wert gemäss Bloom-Filter in der Tablle enthalten sein könnte, wird die aufwändige Suche in den Datenwerten angestossen.

3 Testen der Fehlerwahrscheinlichkeit

Um die Zuverlässigkeit des Bloom-Filters zu testen, verwenden wir nach dem Einlesen eine grosse Anzahl Testwerte. Dabei überprüfen wir in der Methode *testReliability()* und mit einer einfachen *contians()*-Abfrage, ob die Testwerte (von welchen der Bloom-Filter behauptet, dass sie in der Liste seien) auch tatsächlich in der Liste sind. Das Verhältnis von diesen False-Positives zu der Gesamtanzahl getesteten Wörtern gibt uns dabei die Zuverlässigkeit. Diese weicht dabei nie mehr als die voreingestellte Fehlerwahrscheinichekit p von 100 Prozent ab, da wir die Filtergrösse m und die Anzahl Hash-Funktionen k in Abhängigkeit der Anzahl zu speichernden Wörtern n anpassen. [1]

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

$$k = \frac{m}{n} \ln 2$$

[1] Quelle der Formeln: `https://en.wikipedia.org/wiki/Bloom_filter`

Figure 1: Konsolen-Ausgaben mit verschiedenen Fehlerwahrscheinlichkeiten

```
> Task :Application.main()
Bitte gewünschte Fehlerwahrscheinlichekeit p eingeben (z.B. 0.01 für 1%)
-> Achtung es funktioniert nur Punkt als Dezimaltrennzeichen!
0.01
Anzahl Wörter: 58110
gewünschte maximale Fehlerwahrscheinlichkeit: 1.0%
Anzahl Hash-Funktionen: 7
Anzahl Testwerte: 20000
Zuverlässigkeit: 99.5908625980225
```

```
> Task :Application.main()
Bitte gewünschte Fehlerwahrscheinlichekeit p eingeben (z.B. 0.01 für 1%)
-> Achtung es funktioniert nur Punkt als Dezimaltrennzeichen!
0.02
Anzahl Wörter: 58110
gewünschte maximale Fehlerwahrscheinlichkeit: 2.0%
Anzahl Hash-Funktionen: 6
Anzahl Testwerte: 20000
Zuverlässigkeit: 99.19179570768813
```

```
> Task :Application.main()
Bitte gewünschte Fehlerwahrscheinlichekeit p eingeben (z.B. 0.01 für 1%)
-> Achtung es funktioniert nur Punkt als Dezimaltrennzeichen!
0.05
Anzahl Wörter: 58110
gewünschte maximale Fehlerwahrscheinlichkeit: 5.0%
Anzahl Hash-Funktionen: 5
Anzahl Testwerte: 20000
Zuverlässigkeit: 98.013556137172
```