Index Construction

Sven Fiergolla
1252732
s4svfier@uni-trier.de

4. Mai 2018

1 Einführung

Information Retrieval, zu deutsch Informationsrückgewinnung, ist ein Fachgebiet, welcher sich mit computergestütztem Suchen nach komplexen Inhalten befasst. Der häufigste Anwendungsbereich ist das Suchen von Dokumenten aus einer Sammlung, die für einen Anwender relevant sind. Würde dies auf der unveränderten Datenmenge geschehen, müsste bei jeder Suchanfrage, die komplette Menge an Rohdaten durchsucht werden. Dies ist ein sehr ressourcenintensiver Prozess, daher wird zunächst auf die Hardware sebst eingegangen. Anschließend werden einige Verfahren zum Erstellen eines Index vorgestellt und auf deren Vor- und Nachteile eingegangen.

2 Hardware Constraints

Typische Systemeigenschaften eines Desktop PC im Consumerbereich:

- *clock rate* 2-4 GHz, 4-8 Kerne
- *main memory* 4-32 Gb
- *disk space* $\leq$ 1 TB SSD oder $\geq$ 1 TB HDD
  - HDD (hard disk drive)
    - *average seek time* zwischen 2 und 10 ms
    - *transfer* 150 - 300 MB/s
  - SSD (solid state disk)
    - *average seek time* zwischen 0.08 und 0.16 ms
    - *transfer* Lesen: 545 MB/s, Schreiben: 525 MB/s

Die zu durchsuchende Datensammlung, häufig auch Korpus geannt, befindet sich auf der Festplatte, welche deutlich langsamer arbeitet als der Arbeitsspeicher oder die CPU. Daher stellt die Festplatte in diesem Fall ein Bottleneck[1] dar.

Um die intensive Nutzung der Festplatte zu vermeiden, wird eine alterntive Datenstruktur benötigt. Je nach Anforderung kann eine andere Struktur verwendet werden. Für klassisches „Document Retrival“ wird jedoch in der Regel ein so genannter Invertiert Index konstruiert (siehe Invertierter Index). Da mit sich der SSD eine neue Speichertechnologie verbreitet hat, welche deutlich bessere Zugriffszeiten hat als eine HDD, wird diese seperat betrachtet (siehe Indexierung mit Solid State Drives).

3 Index Construction

Um das ressourcenintensive Durchsuchen der Rohdaten zu vermeiden, existieren geeignete Datenstrukturen um Anfragen beantworten zu können, ohne alle Daten durchsuchen zu müssen. Eine einfache geeignete Struktur wäre eine „Term-Dokument-Matrix“, welche spaltenweise die Dokumente auflistet und zeilenweise das Vorkommen einzelner Worter. Auf einer solchen Matrix lässts sich mit einfachen boolschen Anfragen arbeiten, jedoch hat diese Datenstruktur auch einige Nachteile. Beipsielsweise wächst die Matrix zu stark an für große Sammlungen, es lassen sich keine Komplexeren Anfragen stelle und ein Ranking der Dokumente ist ebenfalls nicht möglich. Daher betrachten wir im Folgenden lediglich den invertierten Index als geeignete Datenstruktur.

Um aus einer Sammlung von Rohdaten (Abbildung 1) die relevanten Informationen zu gewinnen, müssen vorher einige Anpassungen getätigt werden. Besteht die Sammlung beispielsweise aus

[1] Wörtlich: „Flaschenhals“ oder „Engpass“. Gemeint ist ein Engpass beim Transport von Daten, der maßgeblichen Einfluss auf die Arbeitsgeschwindigkeit hat.

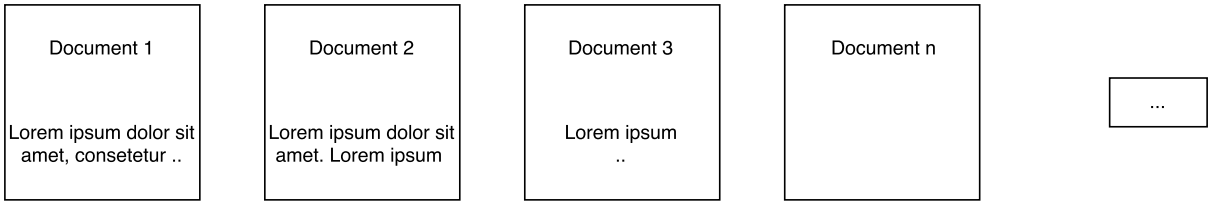

Abbildung 1: beispielhafter Dokumenten-Korpus

Webseiten, müssen die Informationen vor dem Indizieren aus der *.html*-Datei geparst und von html-Tags befreit werden. In der Regel wird ebenfalls Tokenization und Stemming auf den Text angewendet. Bei der Tokenisierung wird Dokument in kleinste Einheiten (meistens einzene Wörter) geteilt. Stemming erlaubt, dass verschiedene morphologische Varianten eines Wortes auf ihren gemeinsamen Wortstamm zurückgeführt werden (beispielsweise die Deklination von Wortes oder Wörter zu Wort und Konjugation von gesehen oder sah zu seh). Dies ermöglicht, dass bei einer Suchanfrage auch Dokumente gefunden werden können, in denen eine leichte Variation des gesuchten Wortes vorkommt, da beide auf den gleichen Term reduziert werden.

3.1 Invertierter Index

Die wesentlichen Schritte zur Erstellung eines invertierten Index sind in Abbildung 2 dargestellt. Zuerst werden alle „Term-Dokument ID“ Paare gesammelt. Diese werden anschließend, dem Term nach, lexikographisch sortiert. Zuletzt werden die Dokument ID's für jeden Term in einer so genannten Postingslist zusammengefasst und Statistiken wie die Frequenz eines Terms berechnet. Die Postingslisten aller Terme bezeichnet man als invertierten Index, da die Zuordnung von Dokumenten auf deren Text invertiert wurde zu der Zuordnung zwischen einem Term und dessen Vorkommen. Bei einer Suchanfrage muss nun lediglich die Postingsliste zu dem gesuchten Term zurückgegeben werden. Aus Performancegründen werden häufig eindeutige Term ID's statt den Termen selbst genutzt. Für kleine Sammlungen kann dies im Arbeitsspeicher geschehen, für größere Sammlungen werden jedoch alternative Verfahren benötigt, auf die im Folgenden eingegangen wird.

In „Managing Gigabytes: Compressing and Indexing Documents and Images“[1] wird beispielhaft mit einer Modelsammlung gearbeitet, der Reuters-RCV1. Sie umfasst rund 800.000 Nachrichtenartikel, ist etwa 1 GB groß und besitzt 100.000.000 Terme. Die „Term-Dokument ID“ Paare dieser Sammlung auf einer Festplatte zu sortieren ist sehr ineffizient. Beim Sortieren kann von einer Komplexität von $O(n \cdot log_2(n))$ ausgegangen werden. Möchte man nun alle Terme der Reuters-RCV1 Sammlung sortieren und man von 2 Zugriffen auf die Festplatte beim Sortieren sowie einer durchschnittlichen Zugriffszeit auf die Festplatte von 5ms ausgeht, dauert es in etwa:

$$(100.000.000 \cdot log_2(100.000.000)) \cdot 2 \cdot (5 \cdot 10^{-3})$$

$$= 2.6575424759... \cdot 10^7 \text{ Sekunden}$$

$$= 307.59 \text{ Tage}$$

Diese Dauer resultiert vorallem aus der hohen Zugriffszeit einer mechanischen Festplatte, da bei jedem wahlfreiem Zugriff der Lesekopf neu positioniert werden muss. Das Lesen von sequenziellen Daten ist deutlich schneller. Ein erster Ansatz ist es, den wahlfreien Zugriff zu minimieren und die Daten hauptsächlich Blockweise zu lesen.

3.2 Blocked sort-based indexing (BSBI)

n = 0;
while *all documents have not been processed* **do**
    n = n + 1;
    block = ParseNextBlock();
    BSBI-INVERT(block);
    WriteBlockToDisk(block, f_n);
end
MergeBlocks($f_1, \cdots, f_n$; f_{merged});

Algorithm 1: BSBI Algorithmus

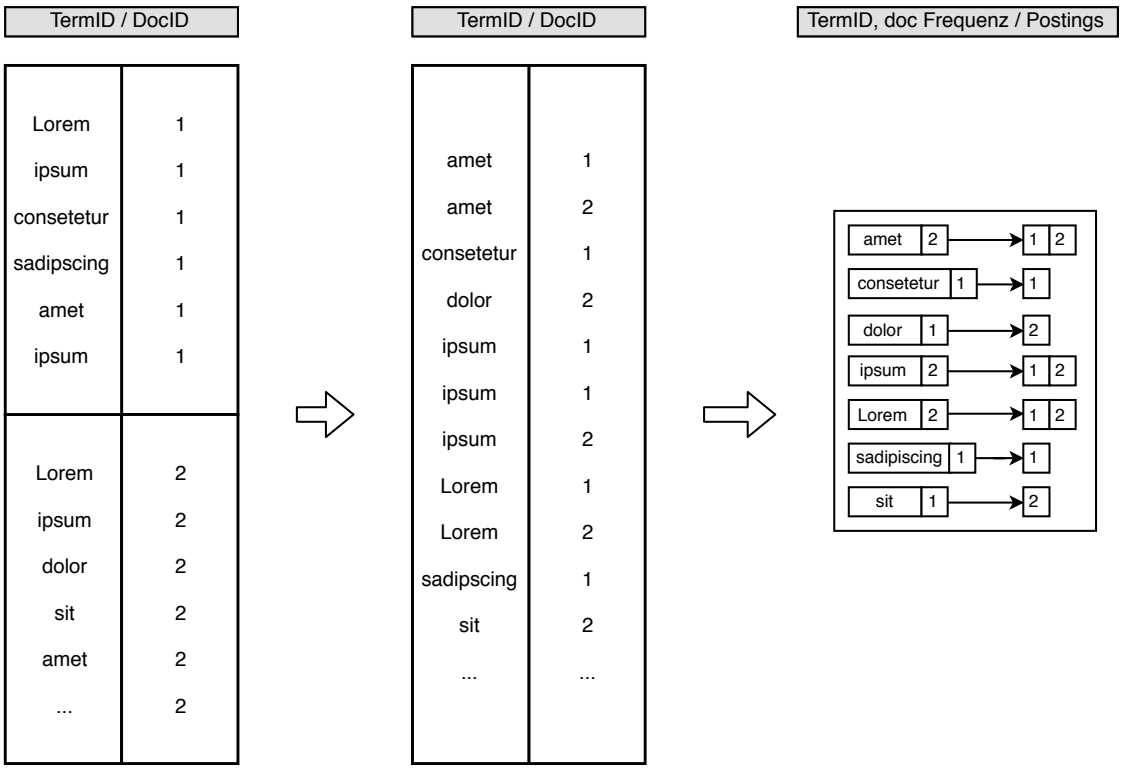

Abbildung 2: Invertierter Index

```
outputFile = new File();
dictionary = new HashFile();
while free memory available do
    token = next(TokenStream);
    if term(token) ∉ dictionary then
        PostingsList =
          AddToDictionary(dictionary,
          term(token));
    else
        PostingsList = GetPostingsList(dictionary,
          term(token));
    end
    if full(PostingsList) then
        PostingsList =
          DoublePostingsList(dictionary,
          term(token));
    end
    AddToPostingsList(PostingsList,
      docID(token));
    SortedTerms = SortTerms(dictionary);
    WriteBlockToDisk(SortedTerms, dictionary,
      OutputFile);
end
```

Algorithm 2: SPIMI-invert Algorithmus

3.3 Single-pass in-memory indexing (SPIMI)

3.4 Distributed indexing

3.5 Dynamic indexing

3.6 andere Indexierungsverfahren

4 Indexierung mit Solid State Drives

5 Fazit

Literatur

[1] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze „Introduction to Information Retrieval“ Cambridge University Press 2008, pp. 1-18 and 67-84.

[2] Ian H. Witten, Alistair Moffat, Timothy C. Bell „Managing Gigabytes: Compressing and Indexing Documents and Images“ Morgan Kaufman Publishers 1999, pp. 223-261.

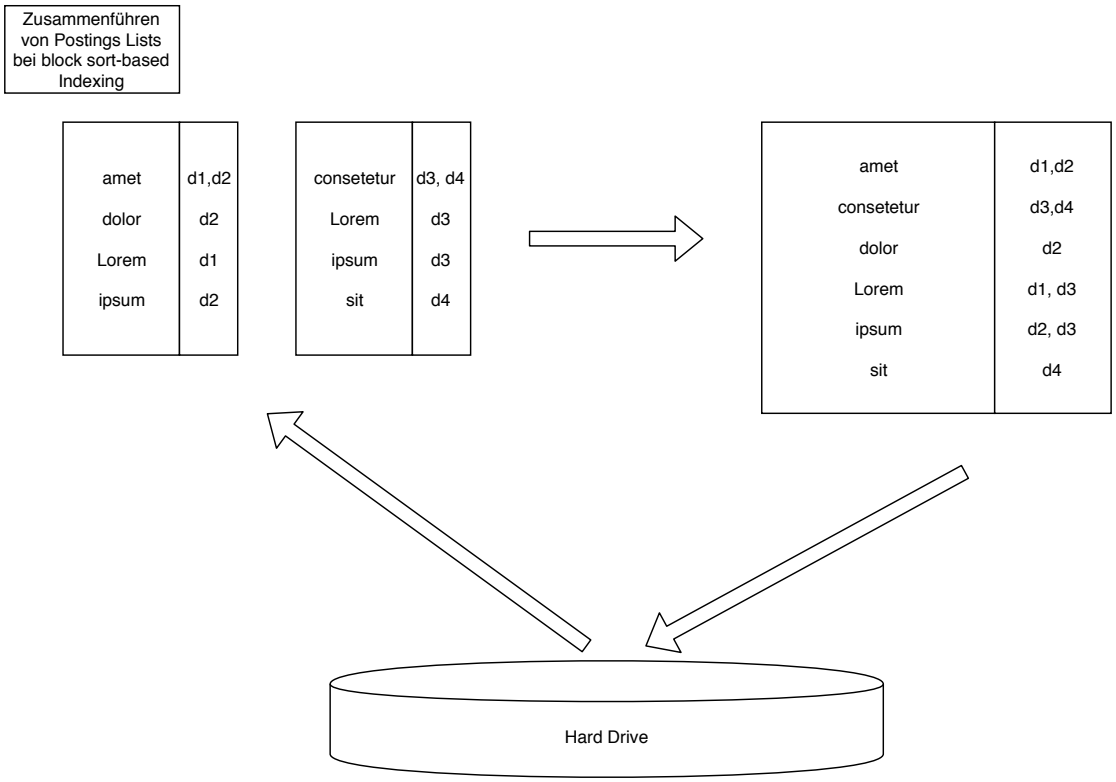

Abbildung 3: merging bei BSBI

[3] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, Ke YiTree (Hong Kong University of Science and Technology) „Indexing on Solid State Drives“ The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.