

Index Construction

Sven Fiergolla
1252732
s4svfier@uni-trier.de

8. 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 selbst 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 (Stand 2018):

- *clock rate* 2-4 GHz, 4-8 Kerne
- *main memory* 4-32 Gb
- *disk space* ≤ 1 TB SSD oder ≥ 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 genannt, befindet sich auf der Festplatte, welche deutlich langsamer arbeitet als der Arbeitsspeicher oder die CPU. Daher stellt die

Festplatte in diesem Fall ein Bottleneck¹ dar. Um die intensive Nutzung der Festplatte zu vermeiden, wird eine alternative Datenstruktur benötigt. Je nach Anforderung kann eine andere Struktur verwendet werden. Für klassisches „Document Retrieval“ wird jedoch in der Regel ein so genannter Invertierter Index konstruiert (siehe Invertierter Index). Da sich mit der SSD eine neue Speichertechnologie verbreitet hat, welche deutlich bessere Zugriffszeiten hat als eine HDD, wird diese separat 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 Wörter. Auf einer solchen Matrix lässt sich mit einfachen boolschen Anfragen arbeiten, jedoch hat diese Datenstruktur auch einige Nachteile. Beispielsweise wächst die Matrix zu stark an für große Sammlungen, es lassen sich keine komplexeren Anfragen stellen 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 Webseiten, müssen die Informationen vor dem Indizieren aus der *.html*-Datei geparkt und von *html*-Tags befreit werden. In der Regel wird ebenfalls

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

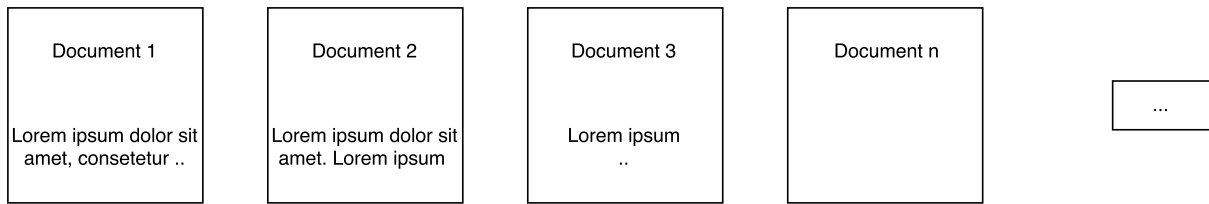


Abbildung 1: beispielhafter Dokumenten-Korpus

Tokenization und Stemming auf den Text angewendet. Bei der Tokenisierung wird Dokument in kleinste Einheiten (meistens einzelne 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 „Introduction to Information Retrieval“[1] wird beispielhaft mit einer Modellsammlung 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:

$$\begin{aligned} & (100.000.000 \cdot \log_2(100.000.000)) \cdot 2 \cdot (5 \cdot 10^{-3}) \\ &= 2.6575424759... \cdot 10^7 \text{ Sekunden} \\ &= 307.59 \text{ Tage} \end{aligned}$$

Diese Dauer resultiert vor allem 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

Eine Ansatz dies zu umgehen ist den Index Blockweise zu erstellen mit Hilfe des *block sort-based indexing Algorithmus* (siehe Algorithmus 1) oder auch *BSBI* auf welchen im Folgenden näher eingegangen wird.

```

1 n = 0;
2 while all documents have not been processed do
3     n = n + 1;
4     block = ParseNextBlock();
5     BSBI-INVERT(block);
6     WriteBlockToDisk(block, fn);
7 end
8 MergeBlocks(f1, ..., fn; fmerged);

```

Algorithm 1: BSBI Algorithmus

Der Algorithmus parst eine Menge von Dokumenten in „Term-Dokument ID“ Paare bis ein Block mit einer festen Größe voll ist (Zeile 4). Anschließend wird der Block sortiert, daher sollte der Block problemlos in den Arbeitsspeicher passen, da dieser wesentlich schneller arbeitet als die Festplatte und so ein effizienteres Sortieren ermöglicht wird.

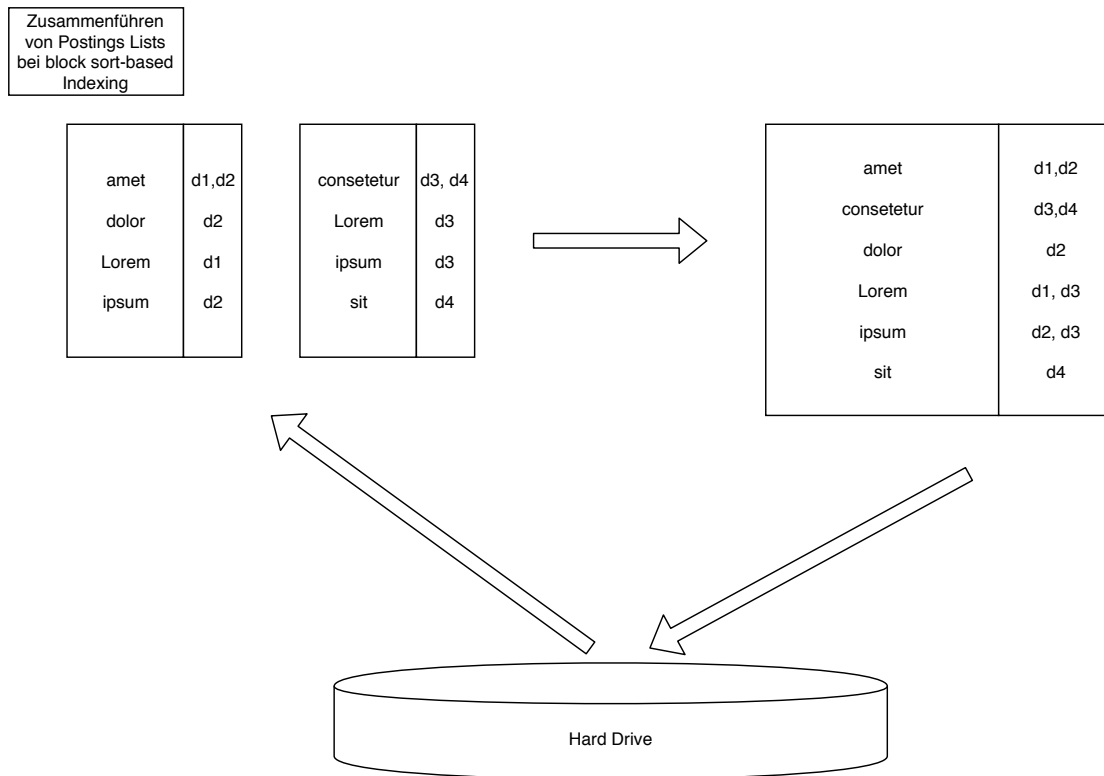


Abbildung 3: merging bei BSBI

den (Zeile 11). Wenn der verfügbare Arbeitsspeicher verbraucht ist, wird das Dictionary und die Postingslisten des Blocks persistiert und ein neuer Block geparst und bearbeitet (Zeile 17). Die Terme müssen sortiert im Dictionary vorliegen, da die Postingslisten dann in lexikographischer Ordnung persistiert werden können. Wurden alle Blöcke bearbeitet, werden die einzelnen Postingslisten zusammengeführt. Dies ist deutlich effizienter wenn sich die Postingslisten der einzelnen Blöcke bereits in lexikographischer Ordnung befinden.

todo fazit

3.4 Distributed indexing

BSBI und SPIMI eignen sich gut für Datensammlungen, die von einem Rechner verarbeitet werden können. Übersteigt der Umfang der Sammlung jedoch die Rechenleistung eines einzelnen Rechners, muss auf andere Weise herangegangen werden. Das Web umfasst ca. eine Millarden Webseiten und muss daher von einem Computercluster bearbeitet werden. Ein Computercluster oder Rechnernetz bezeichnet eine Menge von

vernetzten Computern. Solche Cluster bieten die Möglichkeit eine *MapReduce*-Architektur anzuwenden, eine generelle Architektur für verteiltes Arbeiten. So können aufwendige Probleme von vielen, günstigen Rechnern, auch *Nodes* genannt, gelöst werden. Ein einzelner Rechner wird zum *Master-Node*, welcher für das Verteilen der einzelnen Aufgaben verantwortlich ist. Da einzelne Nodes jederzeit ausfallen können, muss der Master-Node Aufgaben jederzeit neu zuweisen können.

Wird ein Index auf einem solchen verteiltem System erstellt, ist auch der Index selbst geteilt (siehe distributed indexing mit MapReduce). Dies geschieht entweder nach Dokumenten oder nach Termen. Wird der Index nach Dokumenten geteilt, bearbeiten unterschiedliche Maschinen des Clusters unterschiedliche Abschnitte der Sammlung und erstellen einen Index für jeden Abschnitt, welche anschließend zusammengeführt werden. Häufig wird der Index jedoch auch nach Termen geteilt, diese Herangehensweise wird im Folgenden näher beschrieben. Da der Index verteilt erstellt wird, ist auch die Zuordnung zwischen Termen und ihren Term-ID's verteilt und daher komplexer. Eine ein-

fache Lösung ist es, die Zuordnung für häufige Terme im Voraus zu berechnen und auf alle Nodes zu verteilen. Für seltene Terme können die Terme statt Term-ID's verwendet werden.

Zu Beginn wird die Sammlung in n gleiche Teile unterteilt, die sogenannten *Splits*. Ein Ansatz ist es, die Anzahl der Splits gleich der Anzahl an Nodes zu wählen, so kann jedem Node genau ein Split zugeordnet werden. Alternativ sollte die Größe eines Splits so gewählt werden, dass die Arbeit gleich verteilt werden kann und ein Split von einem herkömmlichen Rechner in einer kurzen Zeit bearbeitet werden kann (16 oder 64MB pro Split). Während der ersten Phase von MapReduce, der *Map-Phase* wird jedem Node ein Split zugewiesen und geparkt. Die Ausgabe der r Nodes wird in j lokale Hilfsdateien gespeichert, welche auch *segment files* genannt werden. Sie beinhalten die Postings je einen von mehreren lexikographischen Abschnitt von Termen eines Splits, was es ermöglicht diese später sequenziell auszulesen. In der *Reduce-Phase* weist der Master-Node je einen der j lexikographischen Abschnitte einem Node zu, welcher alle r segment files dieses Abschnittes zusammenführt. Abschließend werden die verbundenen und sortierten Postingslisten persistiert (siehe „postings“ in Abbildung 6).

lid State Drives“ The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.

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.
- [3] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, Ke YiTree (Hong Kong University of Science and Technology) „[Indexing on So-](#)

Appendix

Single-pass in-memory indexing Algorithmus

```
1 SPIMI-Invert(token_stream)
2 outputFile = new File();
3 dictionary = new HashFile();
4 while free memory available do
5   token = next(TokenStream);
6   if term(token) ∉ dictionary then
7     PostingsList =
7       AddToDictionary(dictionary,
7         term(token));
8   else
9     PostingsList = GetPostingsList(dictionary,
9       term(token));
10  end
11  if full(PostingsList) then
12    PostingsList =
12      DoublePostingsList(dictionary,
12        term(token));
13  end
14  AddToPostingsList(PostingsList,
14    docID(token));
15 end
16 SortedTerms = SortTerms(dictionary);
17 WriteBlockToDisk(SortedTerms, dictionary,
  OutputFile);
```

Algorithm 2: SPIMI-invert Algorithmus

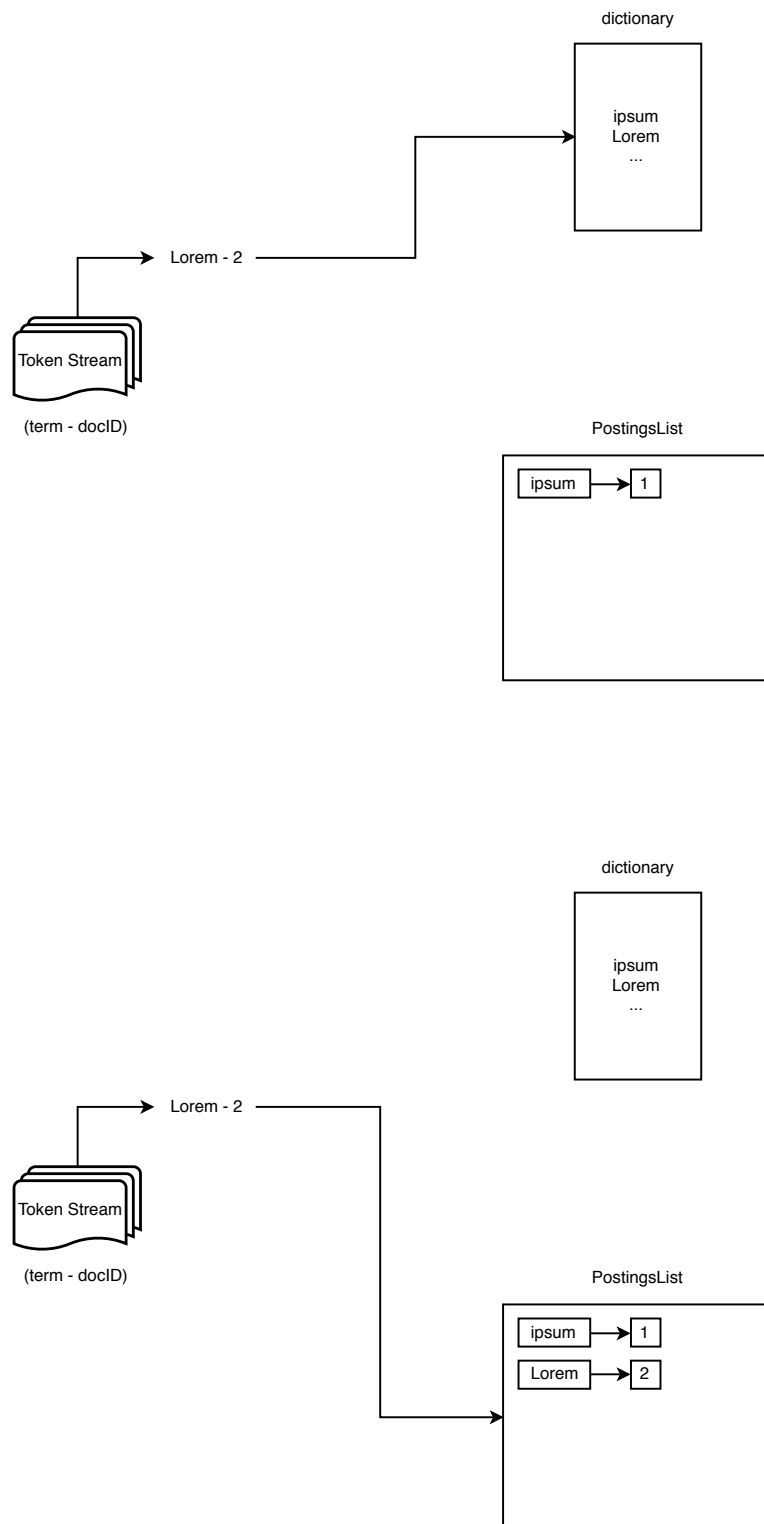


Abbildung 4: Erstes Auftreten eines Terms. Term wird Dictionary hinzugefügt und Postingsliste wird erstellt.

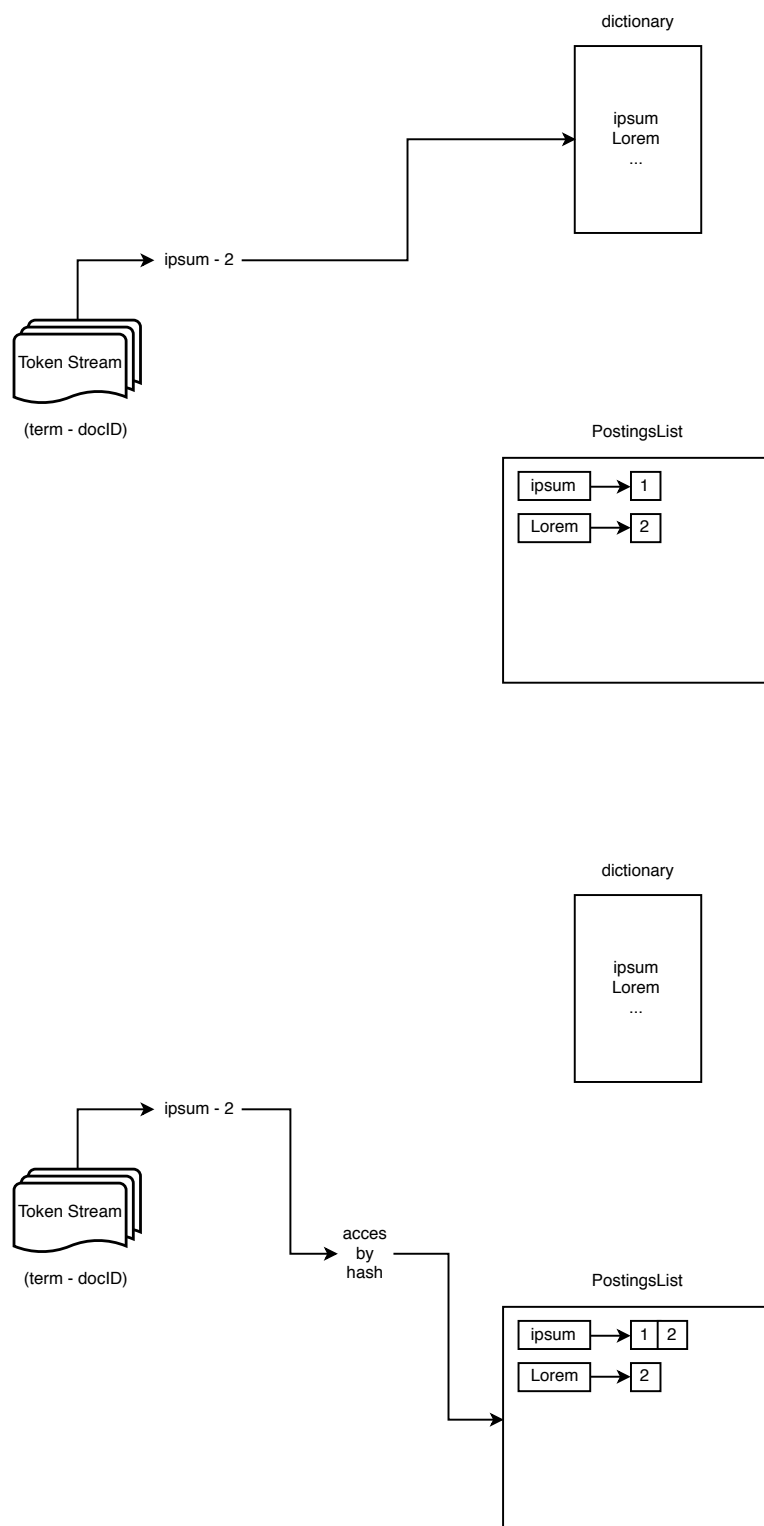


Abbildung 5: Wiederholtes Auftreten eines Terms. Term wird in Dictionary gefunden und per Hash der Postingsliste des Terms hinzuefügt.

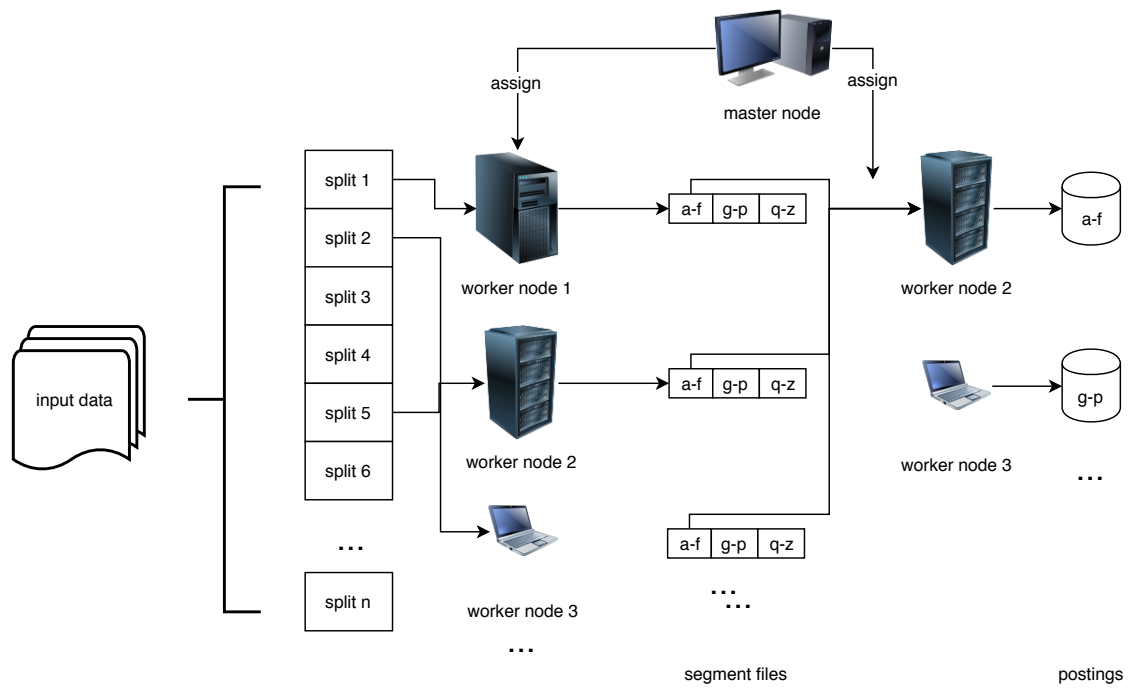


Abbildung 6: distributed indexing mit MapReduce

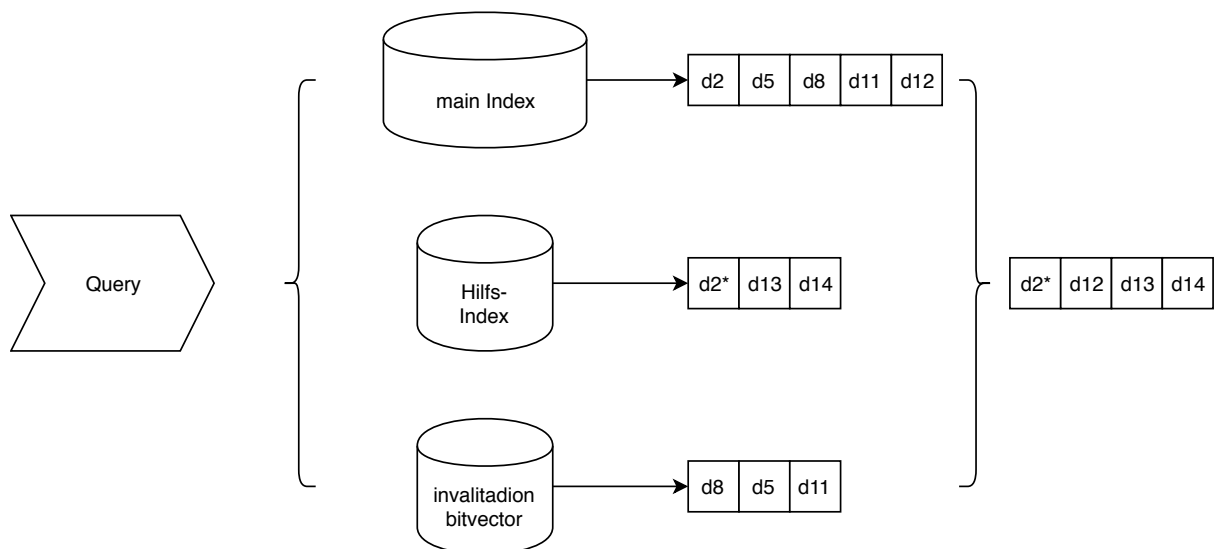


Abbildung 7: Query auf einem dynamic index