Index Construction and Advanced Queries

COMP90042 LECTURE 5, THE UNIVERSITY OF MELBOURNE

by

Matthias Petri

Tue 19/3/2019

Today

1. Inverted index construction.
2. Phrase searching and other advanced query types.

Inverted Index construction

Inverted Index construction

Static scenario:

1. Static collection of documents
2. Offline construction
3. Fixed at query time (no delete or append)

Real world scenario:

1. Document collection grows over time
2. Documents arrive at a given rate (1000 docs/second)
3. Documents should be searchable immediately
4. Might want to delete or update documents

Static construction

Constraints:

1. Static collection of documents (e.g. 1TB webcrawl)
2. Offline construction (only query after construction)
3. Fixed at query time (no modify or append)

Constraints and Properties:

1. Fixed amount of memory available
2. Dataset will not fit in memory
3. Final index small compared to dataset

Static construction - Algorithm

Algorithm:

Split document collection into blocks (X documents per block) and construct inverted indexes for each block. Merge blocks at the end.

```
def ConstructStatic(Collection):
    n = 0
    vocab = {} # map term to term_id
    # split collection into fixed size batches
    for batch in Collection:
        # invert each batch
        B = InvertBatch(batch,vocab)
        storeToDisk(B)
        n = n + 1
    # merge all inverted batches
    MergeAllInvertedBatches(B0,B1...,Bn)
```

Static construction - InvertBatch

Invert one batch:

1. Process documents in batch
2. Use global vocabulary to map terms to term ids
3. Create inverted index for all docs in batch
4. Can already compress postings lists (vbyte)

Stored on disk:

Term-ID	Compressed size	Compressed posting
5	52	<vbyte>
7	454	<vbyte>
9	432	<vbyte>
12	5324	<vbyte>

Static construction - MergeBatches

Merge batches:

1. Open all *n* files containing batches on disk
2. Read one term (or a few) from each file
3. Perform *n*-way merge, merging terms with same ids
4. Merge equivalent to appending bytes as document ids are increasing
5. Read the next terms

Note: In a way this is a form of map and reduce which is used for distributed index construction.

Static construction - MergeBatches

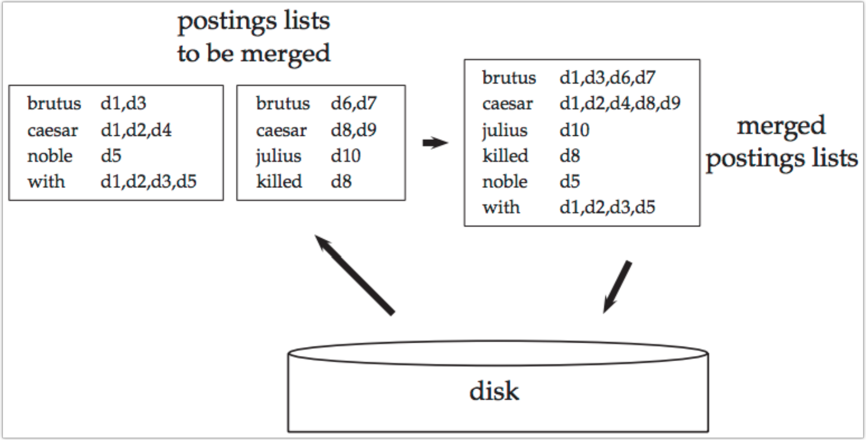

Figure 4.3., Intro to Information Retrieval, Manning et al.

From static to incremental indexing

Auxiliary Index

1. One static large static index on disk
2. As new documents arrive keep them in-memory in second index
3. If second index becomes too big, merge with static index
4. Query both indexes and merge results

From static to incremental indexing

Problem

1. After one year our index has $N = 10$ billion postings
2. At each time we have an inverted index in memory which is merged it has more than $n = 1$ million postings
3. Total merges = N/n. At each merge we look at all existing postings.
   Thus,

   $$totalcost = n + 2n + 3n + 4n + \ldots + (N/n)n = O(N^2/n)$$

   Example: $N = 10$ billion, $n = 1$ million =
   $N^2/n = 10B^2/1M = 100$ trillion I/Os

Incremental, logarithmic indexing

Idea

1. Use a logarithmic number ($\log N$) of indexes. At each level i, store index of size $2^i \times n$
2. Query all $\log N$ indexes at the same time and merge results
3. Construction cost: $N \log(N/n)$

Lets look at this "in action":
`http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html`

Incremental construction

Lucene (Open Source Search Engine):

1. Add over 650GB/hour worth of documents to the index on modern hardware
2. Incremental indexing as fast as batch indexing.

Lucene Code:

```
EnglishAnalyzer sa = new EnglishAnalyzer();
IndexWriterConfig iwc = new IndexWriterConfig(sa);
iwc.setSimilarity(new BM25Similarity());
IndexWriter writer = new IndexWriter(indexDir, iwc);

docContent = "This is a test";
Document document = new Document();
document.add(new Field(FIELD_BODY,docContent));
writer.addDocument(document);
```

Distributed Index Construction

- Search engines such as Elasticsearch support distributed indexes
- Break documents into shards which are spread across processing nodes
- Balance shards such as indexing and search is evenly distributed (hard problem!)
- Search: Query each shard and merge results (hard problem!)

Index Construction - Summary

- Block based processing of large collections
- Merge blocks to larger indexes
- Logarithmic merging reduces I/O costs
- Query multiple indexes at once
- When dealing with large amounts of data, careful engineering of construction algorithms is required to make things “work”

Phrase searching and advanced queries

Advanced queries

Users sometime want to look for:

- Exact phrases such as "the University of Melbourne"
- Arbitrary patterns in non-tokenized text such as "AGCTAGCAGAA" in genome databases
- Documents/sentences that are similar to an existing example.

Phrase queries

What is a phrase query?

1. Seek to identify documents that contain a specific phrase "the who"
2. Combination of individual known terms that occur sequentially in documents
3. Two or more terms possible - "the president of the united states"

Approaches

1. Inverted Index based
2. String matching indexes (Suffix Arrays)

Phrase queries - Inverted Index

Hack

Find the top-K most frequent phrases in your query log and add them as a token to your dictionary. Probably solves 90% of all incoming queries.

Positional Inverted Index

- In addition to document ids and frequencies, store the **positions** of each term in the document.
- Perform intersection to find phrases
- Index size dramatically increased

Positional Inverted Index

term t	f_t	Postings list for t (docids,freqs)
and	6	$\langle 1, 6, 7, 8, 9, 12 \rangle, \langle 1, 2, 3, 3, 1, 2 \rangle$
big	3	$\langle 2, 5, 42 \rangle, \langle 1, 1, 1 \rangle$
old	1	$\langle 32 \rangle, \langle 4 \rangle$
in	7	$\langle 2, 3, 5, 6, 8, 14, 25 \rangle, \langle 1, 1, 4, 1, 5, 3, 1 \rangle$
the	52	$\langle 1, 2, 3, 4, 5, 7, 8, 9, \ldots \rangle, \langle 10, 21, 10, 42, 12, 14, 12, 4, \ldots \rangle$
night	4	$\langle 1, 12, 13, 14 \rangle, \langle 2, 2, 1, 3 \rangle$
house	5	$\langle 6, 21, 32, 33, 43 \rangle, \langle 2, 3, 4, 2, 1 \rangle$
sleep	3	$\langle 1, 51, 53 \rangle, \langle 1, 2, 3 \rangle$
where	4	$\langle 1, 3, 4, 6 \rangle, \langle 1, 1, 2, 1 \rangle$

Positional Inverted Index

term t	f_t	Postings list for t (docids,freqs,positions)
and	6	$\langle 1, 6, 7, \ldots \rangle, \langle 1, 2, 3, \ldots \rangle, \langle \langle 5 \rangle, \langle 12, 43 \rangle, \langle 6, 45, 212 \rangle, \ldots \rangle$
big	3	$\langle 2, 5, 42 \rangle, \langle 1, 1, 1 \rangle, \langle \langle 8 \rangle, \langle 43 \rangle, \langle 65 \rangle \rangle$
old	1	$\langle 32 \rangle, \langle 4 \rangle, \langle \langle 6, 34, 56, 59 \rangle \rangle$
$\vdots$	$\vdots$	$\vdots$

Positional Index - Search

Phrase Search Algorithm

- Intersect document ids of phrase terms
- For the documents containing all terms, intersect position lists
- Perform ranking on result set

Problems

- Slow when phrase terms occur in many documents in the collection (“the who”)
- Slow when terms occur often in documents but not as phrase (“the president of the united states”)

Positional Index - Intersection

Small-vs-Small

To intersect *n* lists, pick the smallest and intersect with the second smallest. Use result of the intersection to keep intersecting.

5	6	13	53	234

2	4	6	12	53	434	545	656

1	3	4	6	9	10	11	12	13	14	18

1	2	3	4	6	8	9	10	13	14	15	17	19	···

Suffix Arrays

- Alternative search index which does not require tokenization
- Widely used in bioinformatics for exact string searches
- Gives you more runtime guarantees than an inverted index

Example: `https://www.ncbi.nlm.nih.gov/nuccore/?term=GTTCCTCAAG`

Classical String Matching

Problem

Given a string T and a pattern P over an alphabet Σ of constant size σ. Let $n = |T|$ be the length of T, and $m = |P|$ be the length of P and $n \gg m$.

Example

$T = abracadabrabarbara\$$, n=18

$P = bar$, m=3

Problem: String search

- Does P occur in T? (Existence query)
- How often does P occur in T? (Count query)
- Where does P occur in T? (Locate query)

Simple Solutions

- Check for each i in $i \in \{0, ..., n-m-1\}$ if $T[i..i+m-1] = P$.

```
for (size_t i=0; i<n; ++i){
  bool match=true;
  for (size_t j=0; j<m && match; ++j)
    match = (i+j < n && T[i+j] == P[j]);
  if (match)
    return true;
}
return false;
```

- Time complexity: $\mathcal{O}(n \cdot m)$ comparisons

Improved solution

- Knuth, Morris, and Pratt precomputed a table of size m which allows to shift the pattern by possibly more than one position in case of a mismatch and get complexity: $\mathcal{O}(n + m)$
- This solution is optimal in the online scenario, in which we are not allowed to pre-process T (online scenario), but not in ...

our scenario

We are allowed to **pre-compute an index structure** I for T and use I for the string search.

- I should be small
- Time complexity of matching independent of n

Suffix Arrays - Idea

- Store all suffixes of the text
- Sort them in lexicographical order
- Each occurrence of P is a prefix of a suffix

Suffix Arrays (1)

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- First sort suffixes of T. (quicksort: $\mathcal{O}(n^2 \log n)$, best algorithms: $\mathcal{O}(n)$)
- Storing all suffixes takes $n^2 \log \sigma$ bits space. Only store starting positions of suffixes in SA ($n \log n$ bits).
- Question: How fast can we search using T and SA?

Suffix Arrays (2)

- The suffixes are *ordered* in SA. We can use *binary search*!
- Start with the empty string ϵ which matches all prefixes (i.e. the interval $[sp_0..ep_0] = [0..n-1]$) of suffixes in SA.
- Then use binary search to determine the interval $SA[sp_j..ep_j]$ in $SA[sp_{j-1}..ep_{j-1}]$ so that all suffixes start with $P[0..j-1]$ for all $j \in [1..m]$.
- P occurs in T if $[sp_m..ep_m]$ is not empty.
- If P occurs the count query can be answered by $ep_m - sp_m + 1$.
- Time complexity: $\mathcal{O}(m \cdot \log n)$, space $\mathcal{O}(n \log n + |T|)$

Suffix Arrays - Example

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for *bar*.

Suffix Arrays - Example

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for *bar*.
- Step 1: *b* interval [9..12]

Suffix Arrays - Example

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for *bar*.
- Step 1: *b* interval [9..12]
- Step 2: *ba* interval [9..10]

Suffix Arrays - Example

i	$SA[i]$	$T[SA[i]..n-1]T[0..SA[i]-1]$
0	18	$abracadabrabarbara
1	17	a$abracadabrabarbar
2	10	abarbara$abracadabr
3	7	abrabarbara$abracad
4	0	abracadabrabarbara$
5	3	acadabrabarbara$abr
6	5	adabrabarbara$abrac
7	15	ara$abracadabrabarb
8	12	arbara$abracadabrab
9	14	bara$abracadabrabar
10	11	barbara$abracadabra
11	8	brabarbara$abracada
12	1	bracadabrabarbara$a
13	4	cadabrabarbara$abra
14	6	dabrabarbara$abraca
15	16	ra$abracadabrabarba
16	9	rabarbara$abracadab
17	2	racadabrabarbara$ab
18	13	rbara$abracadabraba

- Search for *bar*.
- Step 1: *b* interval [9..12]
- Step 2: *ba* interval [9..10]
- Step 2: *bar* interval [9..10]

Suffix Arrays - State-of-the-Art

- Compressed Suffix Arrays (CSA) use space equivalent to the **compressed** size of the text. Example: 1GB text, 8GB Suffix Array, Compressed text (gzip) 200MB, CSA 150MB
- Can be constructed, parallel, space-efficiently
- Can be used to index TBs of data
- Can solve queries that are hard to solve using an inverted index
- Widely used in bioinformatics where the concept of tokens does not exist

More advanced queries

Many more advanced queries exist

- Wildcard/misspelling queries (Sydney vs. Sidney: query S?dney)
- Regular expression queries (“[Jj]ohn.*”@smith.com???”)
- Proximity queries (“president” close to “obama”)

Often specialized indexing structures are required to solve those kind of queries efficiently. Generally **much slower** than regular queries.

Phrase search - Summary

- Inverted indexes with positional information and intersection
- Use substantially more space than regular inverted index
- Suffix array is an alternative index structure to inverted indexes
- Complex queries require specialized indexes

Further Reading

Reading:

- Manning, Christopher D; Raghavan, Prabhakar; Schütze, Hinrich; Introduction to information retrieval, Cambridge University Press 2008. (Chapter 4)

Additional References:

- `http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html`
- Simon J. Puglisi, William F. Smyth, Andrew Turpin: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. 39(2): 4 (2007)
- Paolo Ferragina, Rodrigo González, Gonzalo Navarro, Rossano Venturini: Compressed text indexes: From theory to practice. ACM JEA 13 (2008)