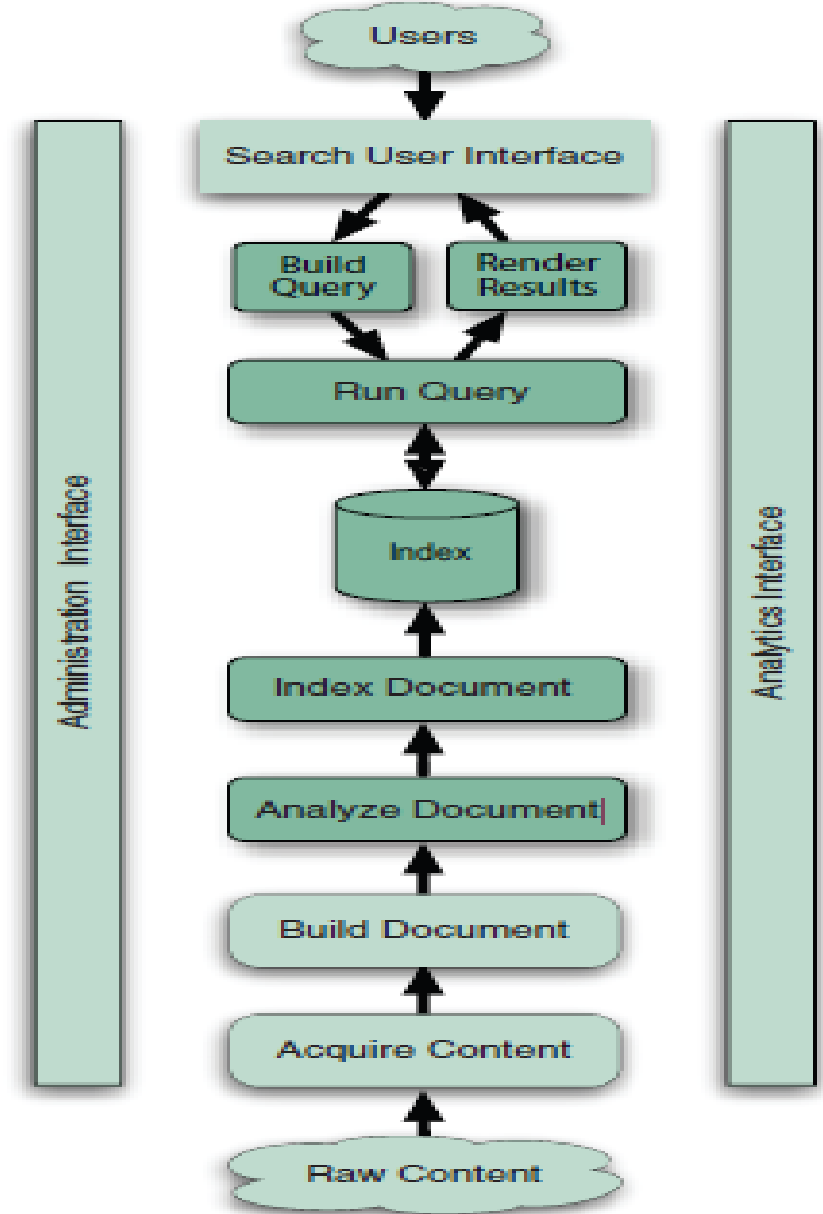




How search happens?

- Build the corpus of documents
- Create an index
- Build a search interface
- Fetch the results using prefix matching
- Rank and score the results to give the best x results.



Searching methods

- Grep method
- Term document incidence matrix(Boolean retrieval)
- Inverted index(with or without skip pointers/skip lists)
- Position postings(for phrase queries)

Term Document Incidence Matrix

Query : Brutus AND Caesar BUT NOT Calpurnia

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

1 if play contains
word, 0 otherwise

Answers to query

- Antony and Cleopatra, Act III, Scene ii

*Agrippa[Aside to DOMITIUS ENOBARBUS]: Why ,Enobarbus,
When Antony found Julius **Caesar** dead,
He cried almost to roaring; and he wept
When at Philippi he found **Brutus** slain*

- Hamlet ,Act III, Scene ii

*Lord Polonius : I did enact Julius **Caesar** I was killed I' the
Capitol;**Brutus** killed me*

With bigger collections!

- Consider $N = 1$ million documents, each about 1000 words
- Avg 6 bytes/word including spaces/punctuation = 6 GB of data in the documents
- Say there are 500K distinct terms among those
- 500K X 1M matrix has half trillion 0's and 1's
- But it has no more than 1 billion 1's (=1000 words X 1M doc)
- Extremely sparse
- What is a better representation? – We only record the 1 position.

What is an inverted index?

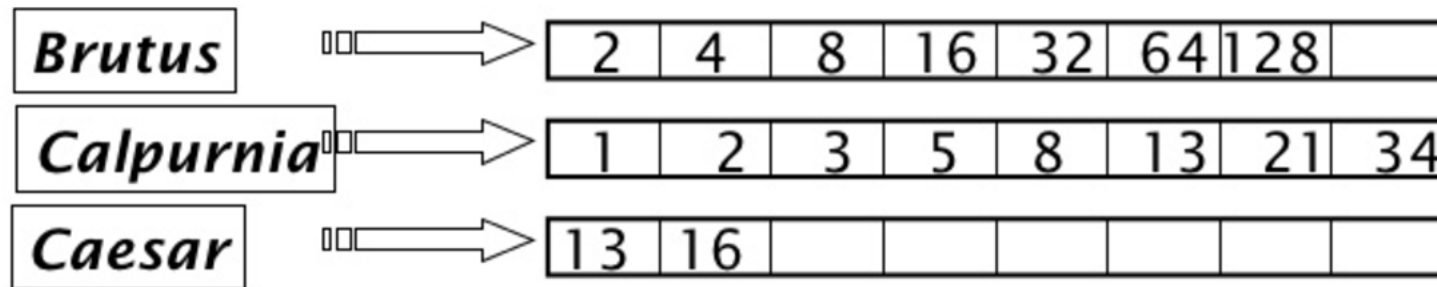
ID	Text
1	Baseball is played during summer months.
2	Summer is the time for picnics here.
3	Months later we found out why.
4	Why is summer so hot here
↑	Sample document data

Dictionary and posting lists →

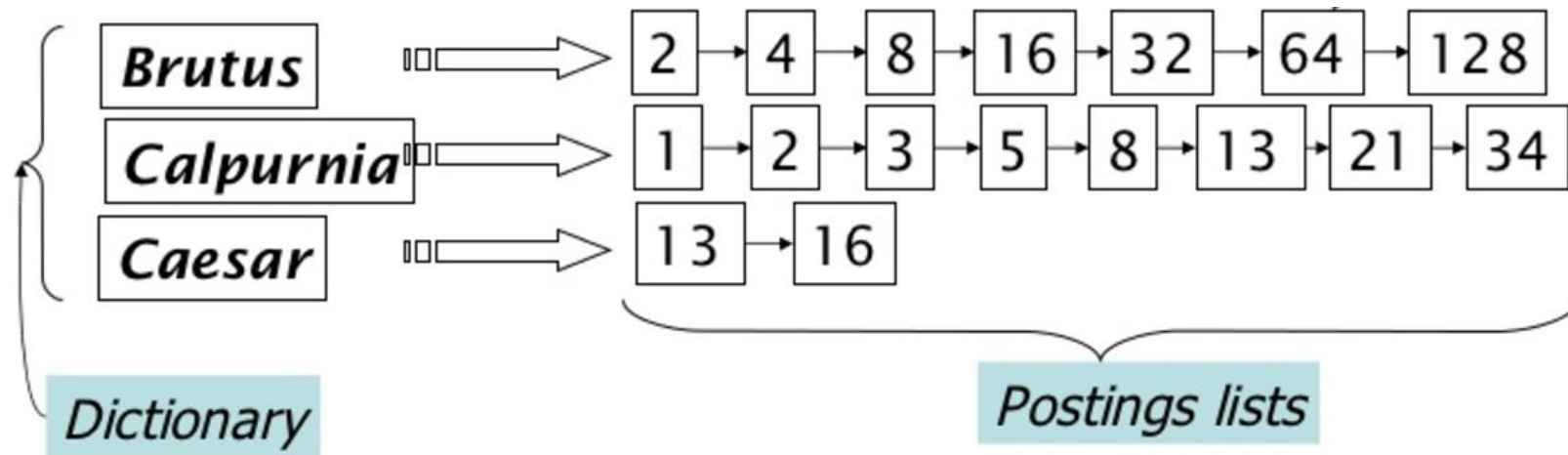
Term	Freq	Document ids
baseball	1	[1]
during	1	[1]
found	1	[3]
here	2	[2], [4]
hot	1	[4]
is	3	[1], [2], [4]
months	2	[1], [3]
summer	3	[1], [2], [4]
the	1	[2]
why	2	[3], [4]

Inverted index

- For each term T, we store the list of document id's containing the term T
- Array vs linked list?
- What happens if a document containing Caesar is added with doc id 14?

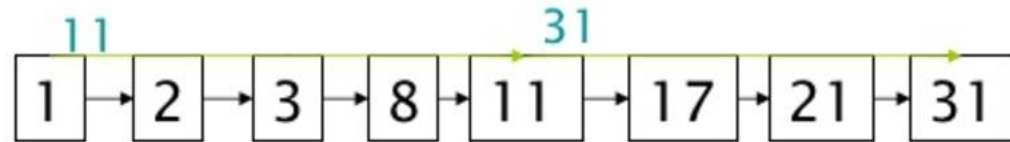
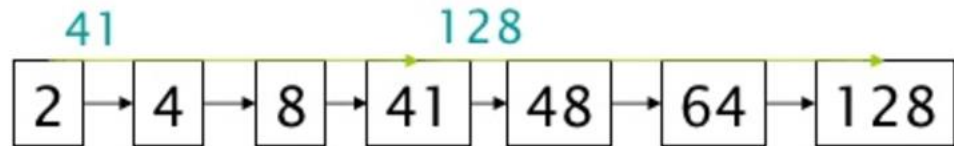


Inverted index



Skip pointers and inverted index

- Why?
- Where?
- How?

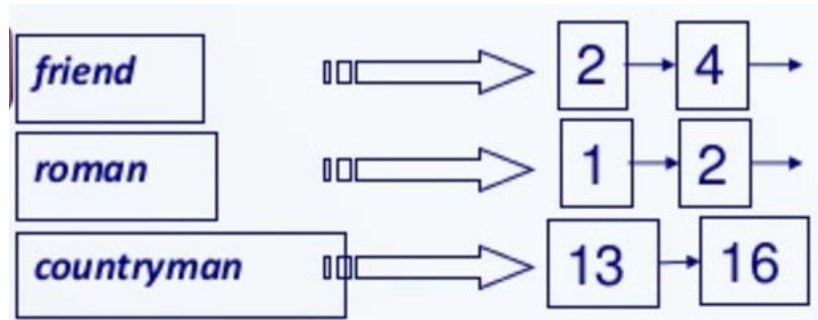


Inverted index construction

Friends, Romans,
Countrymen

Friends| Romans | Countrymen

Friend| Roman | Countryman



Documents to to be
indexed



Tokenizer



Linguistic
modules



Indexer

Indexer steps: Token sequence

- Sequence of (Modified token, Doc ID) pairs

- Doc1

I did enact Julius
Caesar I was killed
I' the
Captitol; Brutus
killed me

- Doc2

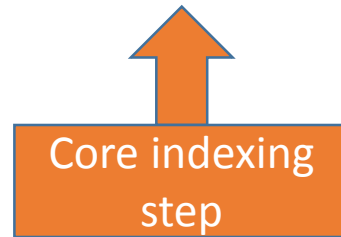
So let it be with
Caesar. The noble
Brutus hath told
you Caesar was
ambitious



Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
I'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer steps : Sort

- Sort by terms
 - And then DocId



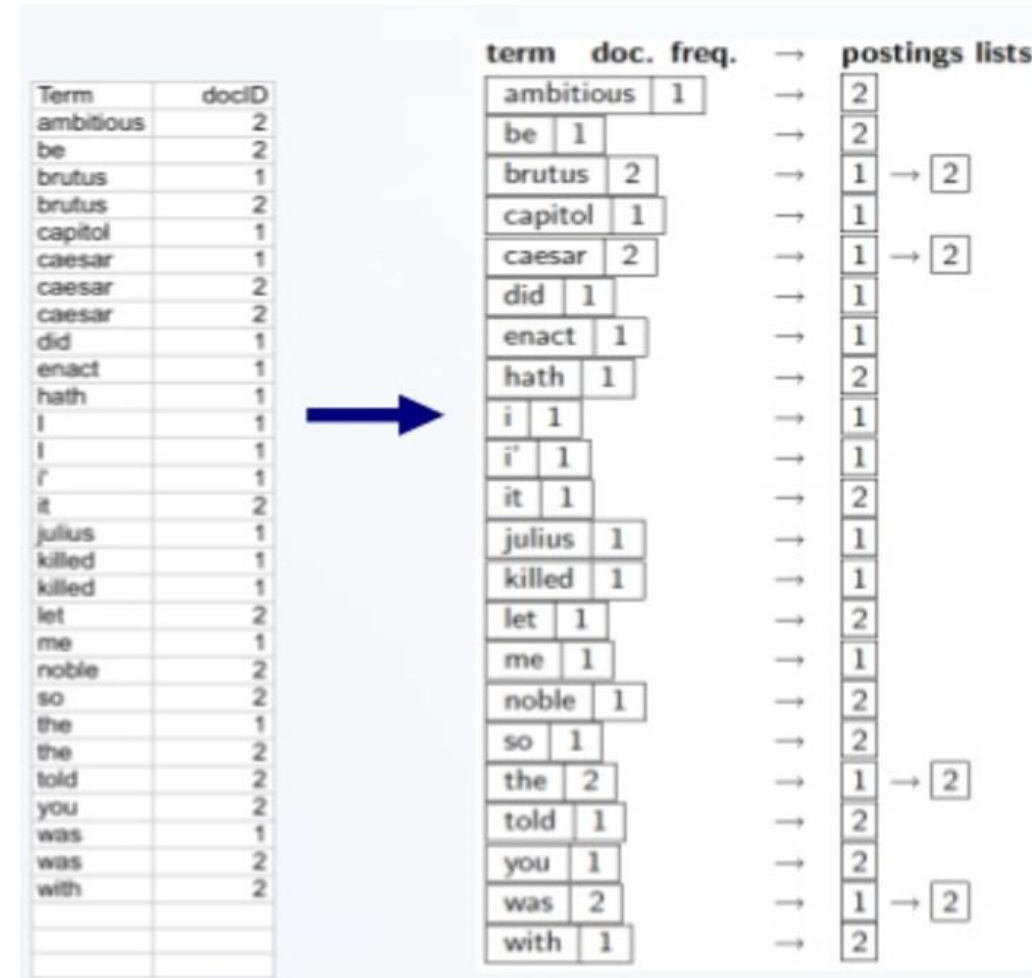
Term	docID	Term	docID
I	1	ambitious	2
did	1	be	2
enact	1	brutus	1
julius	1	brutus	2
caesar	1	capitol	1
I	1	caesar	1
was	1	caesar	2
killed	1	caesar	2
i'	1	did	1
the	1	enact	1
capitol	1	hath	1
brutus	1	I	1
killed	1	I	1
me	1	i'	1
so	2	it	2
let	2	julius	1
it	2	killed	1
be	2	killed	1
with	2	let	2
caesar	2	me	1
the	2	noble	2
noble	2	so	2
brutus	2	the	1
hath	2	the	2
told	2	told	2
you	2	you	2
caesar	2	was	1
was	2	was	2
ambitious	2	with	2

Indexer steps: Dictionary and Postings

Multiple terms entries in a single document are merged.

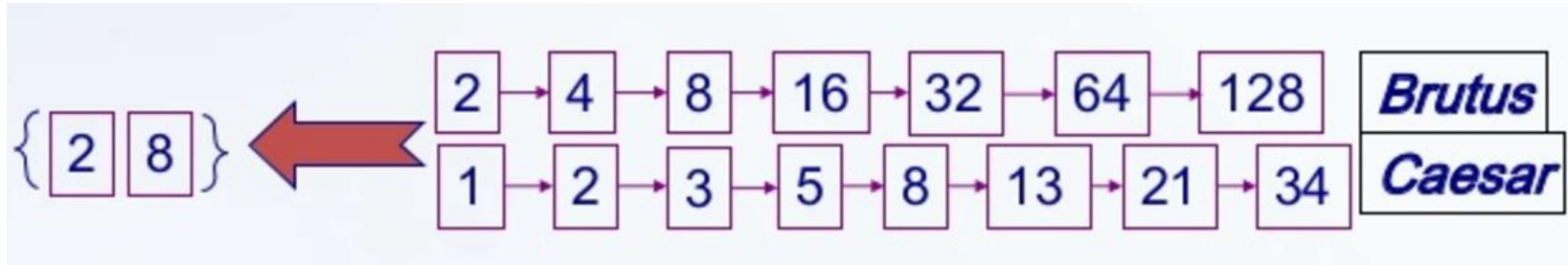
Split into Dictionary and postings

Doc. Frequency information is added



Merging two postings list

- Walk through the two postings simultaneously, in time linear in the total number of posting entries



Longer phrase queries

- Longer phrase queries can be processed by breaking them down.
- “Delhi university cut off 2016” can be broken on boolean queries on biwords
- Results can have false positives

Issues

- False positives
- Index blow up due to bigger dictionary, infeasible for more than biwords, big even for them
- Not practical

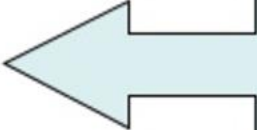
Positional index

- In the postings, store, for each term the position(s) in which tokens of it appear

```
<term, number of docs containing term;  
doc1: position1, position2 ... ;  
doc2: position1, position2 ... ;  
etc.>
```

Positional Index

<*be*: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, ...>



Which of docs *1,2,4,5*
could contain "*to be*
or not to be"?

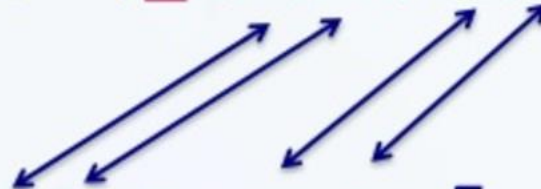
Merge process

– to:

- 2:1,17,74,222,551; 4:8,16,190,429,433; 7:13,23,191; ...

– be:

- 1:17,19; 4:17,191,291,430,434; 5:14,19,101; ...



A dictionary

- How do we store dictionary in memory efficiently?
- How do we quickly lookup at query time?

Dictionary data structures

- Two main choices – Hashes/Trees
- Some IR systems use hashes, some trees

Hashtable

- Each vocabulary term is hashed to an integer
- Faster lookup
- No easy way to find minor variants/No prefix search/Collision resolution
- Rehash everything if vocabulary keeps on growing



Issues

- Rebalancing for insert and delete operations
- Better option is to use a B-Tree in which every internal node has a number of children in the interval a - b
- Slower than a hash table
- Require standard ordering of characters and hence strings

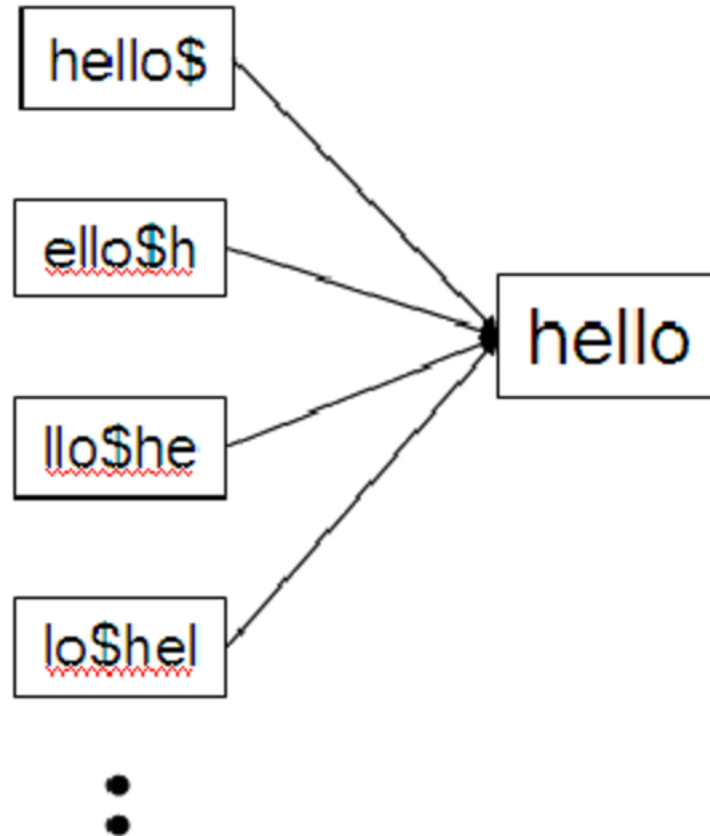
Wild card queries

- Find all docs containing any word beginning “**mon**”
- With B-tree find all words matching prefix mon.
- What about words ending with mon? – Maintain a reverse B-Tree
- Expensive when wild card is in the middle of the query
- Gives rise to **Permuterm** index

Permuterm Index

Queries:

- X lookup on X\$
- *X lookup on X\$*



Permuterm Query Processing

- Rotate wild card query to the right
- Use B-tree lookup as before
- Quadruple the size of lexicons

K-gram index

- Enumerate all k-grams, sequence of k characters occurring in any term
- Eg: from text “April is cruelest month” we get the 2-grams (bi-grams)

\$a,ap,pr,ri,il,l\$, \$i,is,s\$, \$t,th,he,e\$, \$c,cr,ru,
ue,el,le,es,st,t\$, \$m,mo,on,nt,h\$

- Maintain a second inverted index from bigrams to dictionary terms that match each bigram
- Query mon* can now be run as \$m AND mo AND on
- Filter moon - Post filtering steps
- Faster, space efficient than (compared to permuterm)

Wild card queries

- Expensive
- Sometimes hidden behind "Advanced query" section



Our engineers developed
this four you ! for

Spell Correction

Two basic principles:

- Of various alternative correct spellings of the word choose the nearest one
- When two correctly spelled queries are tied choose the one which is most common

Presenting to user:

- When query term is not in the dictionary
- When the search results to query terms return less than a particular number of expected results

Isolated word correction

- Edit distance
- Weighted Edit distance
- N-gram overlap

Edit distance

edit distance is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the minimum number of operations required to transform one string into the other.

Challenges

- To every mis-spelled query do we compute the edit distance to every dictionary term
- How do we cut the set of candidate dictionary terms
- Use n-gram overlap with this – Jaccard similarity
- This can also be used by itself for spell correction

$$|X \cap Y| / |X \cup Y|$$

Context Sensitive spell correction

- Text: I flew from Heathrow to Narita
- Consider the phrase query “flew form Heathrow”
- We’d like to respond
 - Did you mean “flew from Heathrow”
 - Because no docs matched the query phrase

Index construction

- How do we construct an index?
- What strategies do we use with limited main memory?

Hardware Basics

- Access to data in memory is much faster than access to data on disk.
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
- Block sizes: 8KB to 256 KB
- Servers used in IR systems now typically have several GB of main memory, sometimes tens of GB.
- Available disk space is several (2–3) orders of magnitude larger.
- Fault tolerance is very expensive: It's much cheaper to use many regular machines rather than one fault tolerant machine.

Hardware Assumptions

symbol	statistic	value
s	average seek time	5 ms = 5×10^{-3} s
b	transfer time per byte	$0.02 \mu\text{s} = 2 \times 10^{-8}$ s
	processor's clock rate	10^9 s^{-1}
p	low-level operation (e.g., compare & swap a word)	$0.01 \mu\text{s} = 10^{-8}$ s
	size of main memory	several GB
	size of disk space	1 TB or more

Using a larger collection : Reuters RCV1

symbol	statistic	value
N	documents	800,000
L	avg. # tokens per doc	200
M	terms (= word types)	400,000
	avg. # bytes per token (incl. spaces/punctuation)	6
	avg. # bytes per token (without spaces/punctuation)	4.5
	avg. # bytes per term	7.5
T	non-positional postings	100,000,000

How to scale?

Sorting by terms and then by document frequency is the core step for indexing.

In memory indexing does not scale

How can we construct index for large documents given the hardware constraints?

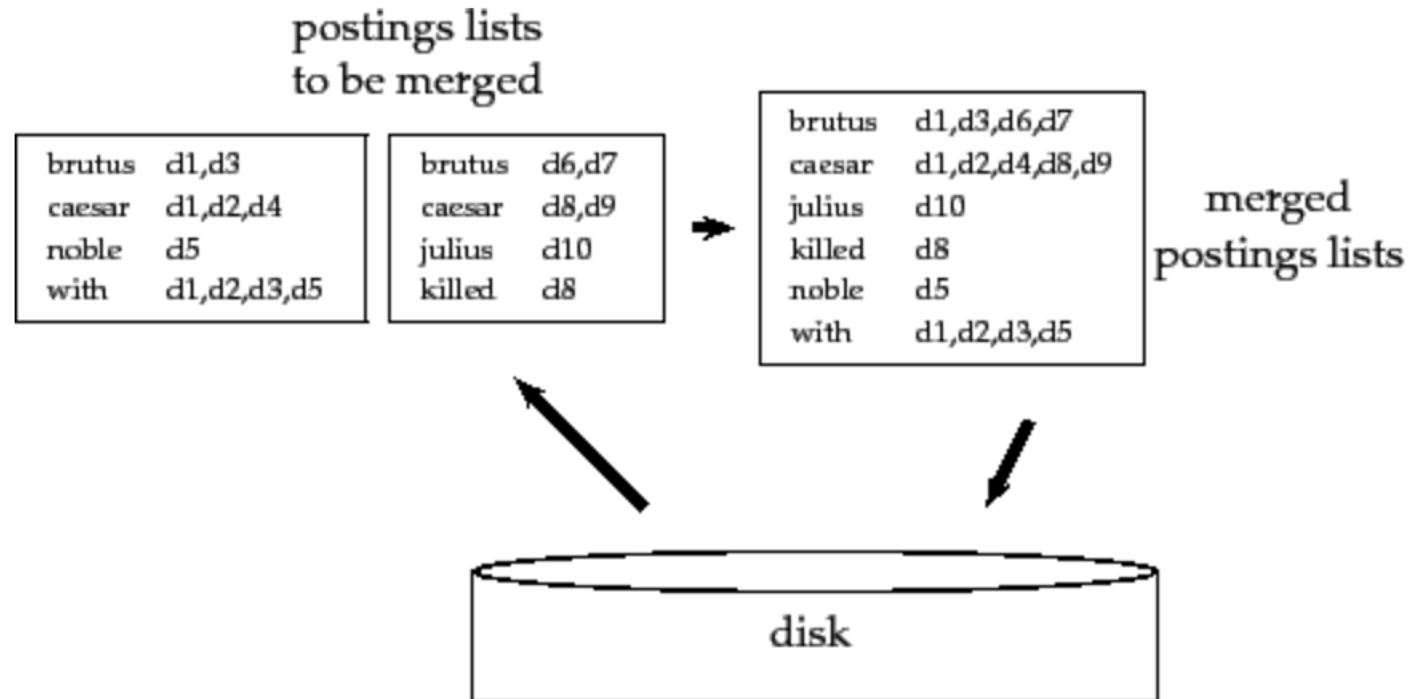
Algorithms

Blocked sort based indexing – BSBI

BSBINDEXCONSTRUCTION()

```
1   $n \leftarrow 0$   
2  while (all documents have not been processed)  
3  do  $n \leftarrow n + 1$   
4     $block \leftarrow \text{PARSENEXTBLOCK}()$   
5    BSBI-INVERT( $block$ )  
6    WRITEBLOCKTODISK( $block, f_n$ )  
7  MERGEBLOCKS( $f_1, \dots, f_n; f_{merged}$ )
```

Merging



Single pass in-memory indexing : SPIMI

```
SPIMI-INVERT(token_stream)
1  output_file = NEWFILE()
2  dictionary = NEWHASH()
3  while (free memory available)
4  do token  $\leftarrow$  next(token_stream)
5      if term(token)  $\notin$  dictionary
6          then postings_list = ADDTODICTIONARY(dictionary, term(token))
7          else postings_list = GETPOSTINGSLIST(dictionary, term(token))
8          if full(postings_list)
9              then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10         ADDTOPOSTINGSLIST(postings_list, docID(token))
11 sorted_terms  $\leftarrow$  SORTTERMS(dictionary)
12 WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13 return output_file
```

Parallel Indexing

- Using map and reduce functionality
- Use same machine for both mapper and reducer function as well other tasks

map:

- d2 : "Caesar died". d1 : "Caesar came, Caesar conquered". →
<Caesar, d2>, <died,d2>, <Caesar,d1>, <came,d1>, <Caesar,d1>, <conquered, d1>

reduce:

- <Caesar, d2>, <died,d2>, <Caesar,d1>, <came,d1>, <Caesar,d1>, <conquered, d1>
→ (<Caesar,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <conquered,(d1:1)>)

Dynamic Indexing

- Maintain “big” main index
- New docs go into “small” auxiliary index
- Search across both, merge results
- Deletions
- Invalidation bit-vector for deleted docs
- Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index

Logarithmic Merge

LMERGEADDTOKEN(*indexes*, Z_0 , *token*)

```
1   $Z_0 \leftarrow \text{MERGE}(Z_0, \{token\})$ 
2  if  $|Z_0| = n$ 
3    then for  $i \leftarrow 0$  to  $\infty$ 
4      do if  $l_i \in \text{indexes}$ 
5        then  $Z_{i+1} \leftarrow \text{MERGE}(l_i, Z_i)$ 
6          ( $Z_{i+1}$  is a temporary index on disk.)
7           $\text{indexes} \leftarrow \text{indexes} - \{l_i\}$ 
8        else  $l_i \leftarrow Z_i$  ( $Z_i$  becomes the permanent index  $l_i$ .)
9           $\text{indexes} \leftarrow \text{indexes} \cup \{l_i\}$ 
10         BREAK
11        $Z_0 \leftarrow \emptyset$ 
```

LOGARITHMICMERGE()

```
1   $Z_0 \leftarrow \emptyset$  ( $Z_0$  is the in-memory index.)
2   $\text{indexes} \leftarrow \emptyset$ 
3  while true
4  do LMERGEADDTOKEN(indexes,  $Z_0$ , GETNEXTTOKEN())
```

- Maintain a series of indexes, each twice as large as the previous one.
- Keep smallest (Z_0) in memory
- Larger ones (l_0, l_1, \dots) on disk
- If Z_0 gets too big ($> n$), write to disk as l_0
- or merge with l_0 (if l_0 already exists) as Z_1
- Either write merge Z_1 to disk as l_1 (if no l_1)
- Or merge with l_1 to form Z_2

Why compression?

- Use less disk space
 - Saves a little money
- Keep more stuff in memory
 - Increases speed
- Increase speed of data transfer from disk to memory
 - [read compressed data | decompress] is faster than [read uncompressed data]
 - Premise: Decompression algorithms are fast
 - True of the decompression algorithms we use

Why compression for inverted indexes?

- Dictionary
 - Make it small enough to keep in main memory
 - Make it so small that you can keep some postings lists in main memory too
- Postings file(s)
 - Reduce disk space needed
 - Decrease time needed to read postings lists from disk
 - Large search engines keep a significant part of the postings in memory.
 - Compression lets you keep more in memory
- We will devise various IR-specific compression schemes

Recall Reuters RCV1

• symbol	statistic	value
• N	documents	800,000
• L	avg. # tokens per doc	200
• M	terms (= word types)	~400,000
•	avg. # bytes per token (incl. spaces/punct.)	6
•	avg. # bytes per token (without spaces/punct.)	4.5
•	avg. # bytes per term	7.5
•	non-positional postings	100,000,000

Index parameters vs. what we index

size of	word types (terms)			non-positional postings			positional postings		
	dictionary			non-positional index			positional index		
	Size (K)	$\Delta\%$	cumul %	Size (K)	$\Delta\%$	cumul %	Size (K)	$\Delta\%$	cumul %
Unfiltered	484			109,971			197,879		
No numbers	474	-2	-2	100,680	-8	-8	179,158	-9	-9
Case folding	392	-17	-19	96,969	-3	-12	179,158	0	-9
30 stopwords	391	-0	-19	83,390	-14	-24	121,858	-31	-38
150 stopwords	391	-0	-19	67,002	-30	-39	94,517	-47	-52
stemming	322	-17	-33	63,812	-4	-42	94,517	0	-52

Lossless vs. lossy compression

- Lossless compression: All information is preserved.
 - What we mostly do in IR.
- Lossy compression: Discard some information
- Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.

Vocabulary vs. collection size

- How big is the term vocabulary?
 - That is, how many distinct words are there?
- Can we assume an upper bound?
 - Not really: At least $70^{20} = 10^{37}$ different words of length 20
- In practice, the vocabulary will keep growing with the collection size

Vocabulary vs. collection size

- Heaps' law: $M = kT^b$
- M is the size of the vocabulary, T is the number of tokens in the collection
- Typical values: $30 \leq k \leq 100$ and $b \approx 0.5$
- In a log-log plot of vocabulary size M vs. T , Heaps' law predicts a line with slope about $\frac{1}{2}$
 - It is the simplest possible relationship between the two in log-log space
 - An empirical finding (“empirical law”)

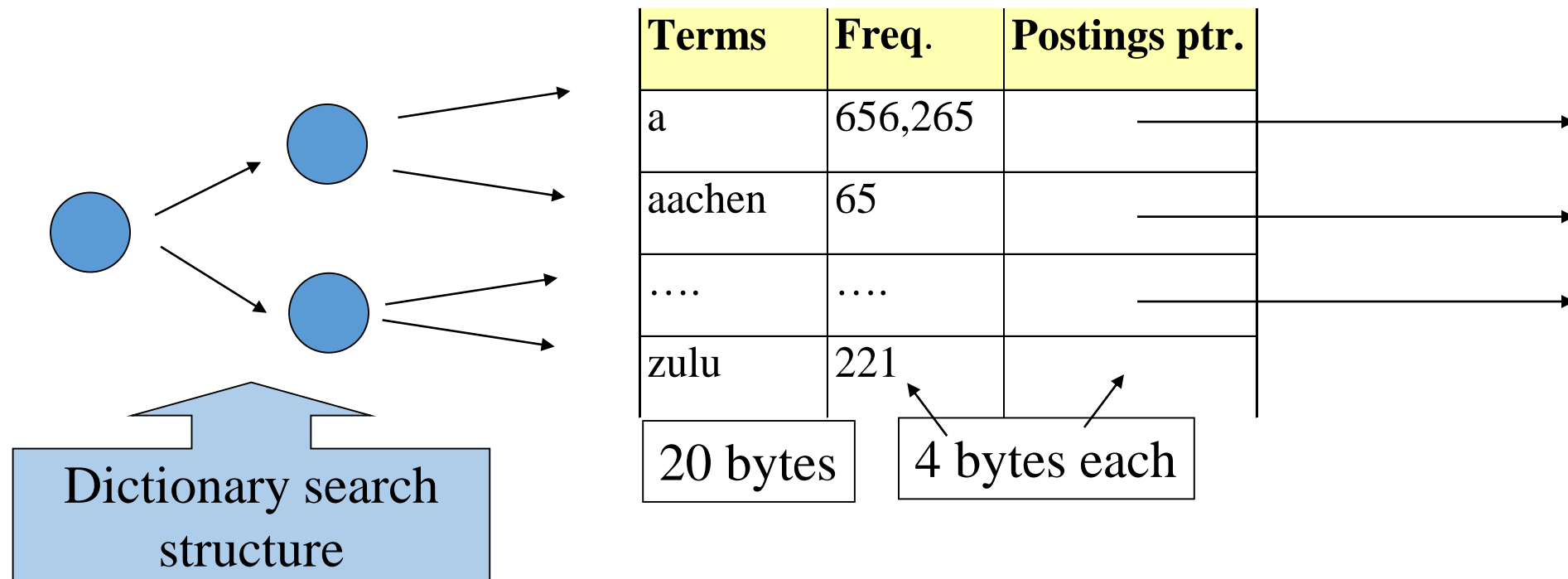
DICTIONARY COMPRESSION

Why compress the dictionary?

- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn't in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary is important

Dictionary storage - first cut

- Array of fixed-width entries
 - ~400,000 terms; 28 bytes/term = 11.2 MB.

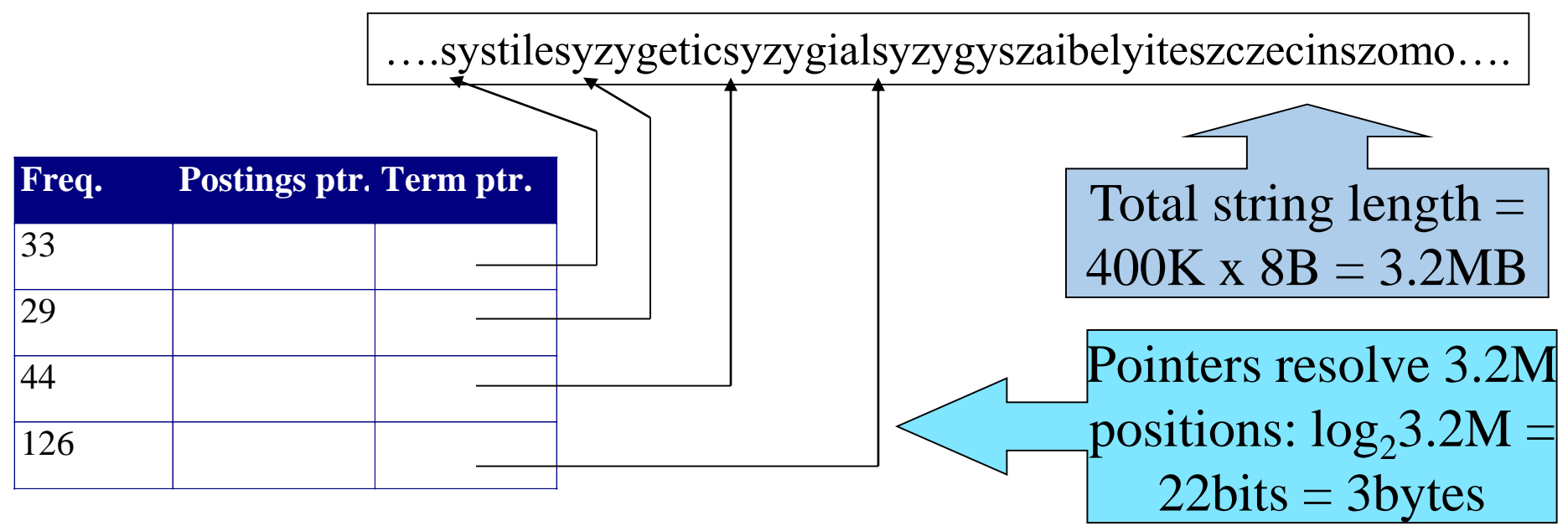


Fixed-width terms are wasteful

- Most of the bytes in the **Term** column are wasted – we allot 20 bytes for 1 letter terms.
 - And we still can't handle *supercalifragilisticexpialidocious* or *hydrochlorofluorocarbons*.
- Written English averages ~4.5 characters/word.
- Ave. dictionary word in English: ~8 characters
 - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.

Compressing the term list: Dictionary-as-a-String

- Store dictionary as a (long) string of characters:
 - Pointer to next word shows end of current word
 - Hope to save up to 60% of dictionary space.



Space for dictionary as a string

- 4 bytes per term for Freq.
 - 4 bytes per term for pointer to Postings.
 - 3 bytes per term pointer
 - Avg. 8 bytes per term in term string
 - 400K terms x 19 \Rightarrow 7.6 MB (against 11.2MB for fixed width)
- } Now avg. 11
} bytes/term,
} not 20.

RCV1 dictionary compression summary

Technique	Size in MB
Fixed width	11.2
Dictionary-as-String with pointers to every term	7.6
Also, blocking $k = 4$	7.1
Also, Blocking + front coding	5.9

POSTINGS COMPRESSION

Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key requirement: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800,000 \approx 20$ bits per docID.
- Our goal: use far fewer than 20 bits per docID.

Postings: two conflicting forces

- A term like ***arachnocentric*** occurs in maybe one doc out of a million – we would like to store this posting using $\log_2 1M \sim 20$ bits.
- A term like ***the*** occurs in virtually every doc, so 20 bits/posting is too expensive.
 - Prefer 0/1 bitmap vector in this case

Postings file entry

- We store the list of docs containing a term in increasing order of docID.
 - **computer**: 33,47,154,159,202 ...
- Consequence: it suffices to store *gaps*.
 - 33,14,107,5,43 ...
- Hope: most gaps can be encoded/stored with far fewer than 20 bits.

Three postings entries

	encoding	postings list					
THE	docIDs	...	283042	283043	283044	283045	...
	gaps		1	1	1		...
COMPUTER	docIDs	...	283047	283154	283159	283202	...
	gaps		107	5	43		...
ARACHNOCENTRIC	docIDs	252000	500100				
	gaps	252000	248100				

RCV1 compression

Data structure	Size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocking, $k = 4$	7.1
with blocking & front coding	5.9
collection (text, xml markup etc)	3,600.0
collection (text)	960.0
Term-doc incidence matrix	40,000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20 bits)	250.0
postings, variable byte encoded	116.0
postings, γ -encoded	101.0

- What if there was a ~~software~~ JAVA library to just this?

Meet Lucene

Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java

It is the core api used in many other open source search applications like – Solr, Nutch, Tika, Mahout as well as Elastic Search

Doug Cutting originally wrote Lucene in 1999.

Doug is also the creator of Hadoop

<http://lucene.apache.org/>



API Sample I (Indexing)

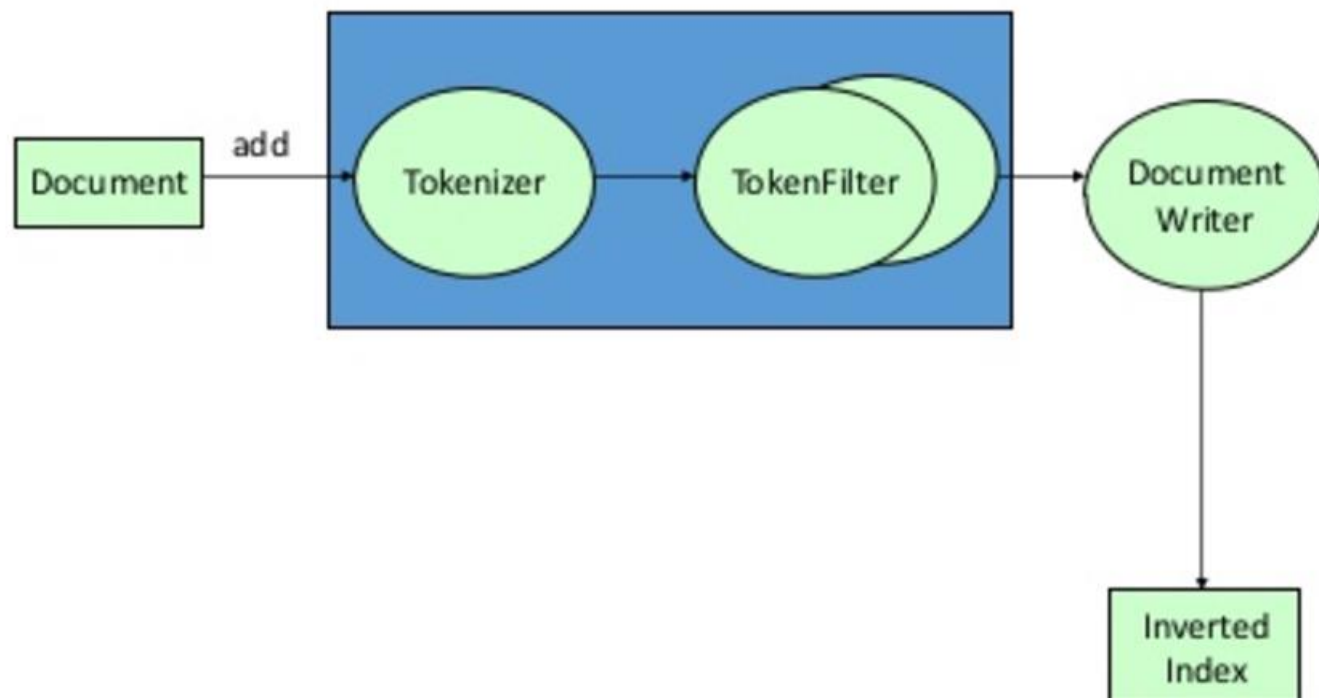
```
private IndexWriter writer;

public Indexer(String indexDir) throws IOException {
    Directory dir = FSDirectory.open(new File(indexDir));
    writer = new IndexWriter(dir, new StandardAnalyzer(Version.LUCENE_CURRENT), true,
        IndexWriter.MaxFieldLength.UNLIMITED);
}

public void close() throws IOException {
    writer.close();
}

public void index(String dataDir, FileFilter filter) throws Exception {
    File[] files = new File(dataDir).listFiles();
    for (File f: files) {
        Document doc = new Document();
        doc.add(new Field("contents", new FileReader(f)));
        doc.add(new Field("filename", f.getName(), Field.Store.YES, Field.Index.NOT_ANALYZED));
        writer.addDocument(doc);
    }
}
```

Indexing Pipeline (Simplified)



Analysis Basic Types

"The quick brown fox jumped over the lazy dogs"

WhitespaceAnalyzer :

[The] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

SimpleAnalyzer :

[the] [quick] [brown] [fox] [jumped] [over] [the] [lazy] [dogs]

StopAnalyzer :

[quick] [brown] [fox] [jumped] [over] [lazy] [dogs]

StandardAnalyzer:

[quick] [brown] [fox] [jumped] [over] [lazy] [dogs]

"XY&Z Corporation - xyz@example.com"

WhitespaceAnalyzer:

[XY&Z] [Corporation] [-] [xyz@example.com]

SimpleAnalyzer:

[xy] [z] [corporation] [xyz] [example] [com]

StopAnalyzer:

[xy] [z] [corporation] [xyz] [example] [com]

StandardAnalyzer:

[xy&z] [corporation] [xyz@example.com]

API Sample II (Searching)

```
public void search(String indexDir, String q) throws IOException, ParseException {  
    Directory dir = FSDirectory.open(new File(indexDir));  
    IndexSearcher is = new IndexSearcher(dir, true);  
  
    QueryParser parser = new QueryParser("contents",  
                                           new StandardAnalyzer(Version.LUCENE_CURRENT));  
    Query query = parser.parse(q);  
    TopDocs hits = is.search(query, 10);  
    System.err.println("Found " + hits.totalHits + " document(s)");  
  
    for (int i=0; i<hits.scoreDocs.length; i++) {  
        ScoreDoc scoreDoc = hits.scoreDocs[i];  
        Document doc = is.doc(scoreDoc.doc);  
        System.out.println(doc.get("filename"));  
    }  
  
    is.close();  
}
```



Elastic search is a search engine based on lucene and is the most popular enterprise search engine followed by Apache Solr. It provides a distributed, [multitenant](#)-capable full-text search engine with an [HTTP](#) web interface and schema-free [JSON](#) documents

Creator : Shay Banon ,**Compass** his first creation led to Elastic Search.

https://www.elastic.co/guide/en/elasticsearch/reference/current/_installation.html

Features at a glance

Extremely elegant and powerful REST API

- Almost all search engine features are accessible over plain HTTP
- JSON formatted query and results
- Can test/experiment/debug with simple tools like curl

Schema less Data model

- Allows great flexibility for application designer
- Can index arbitrary documents right away with no schema metadata
- Can also tweak type/field mappings for indexes as needed

Fully distributed and Highly available

- Tunable index level write paths (index) and read-path (query) distribution policies
- Indexes scale horizontally as new nodes are added

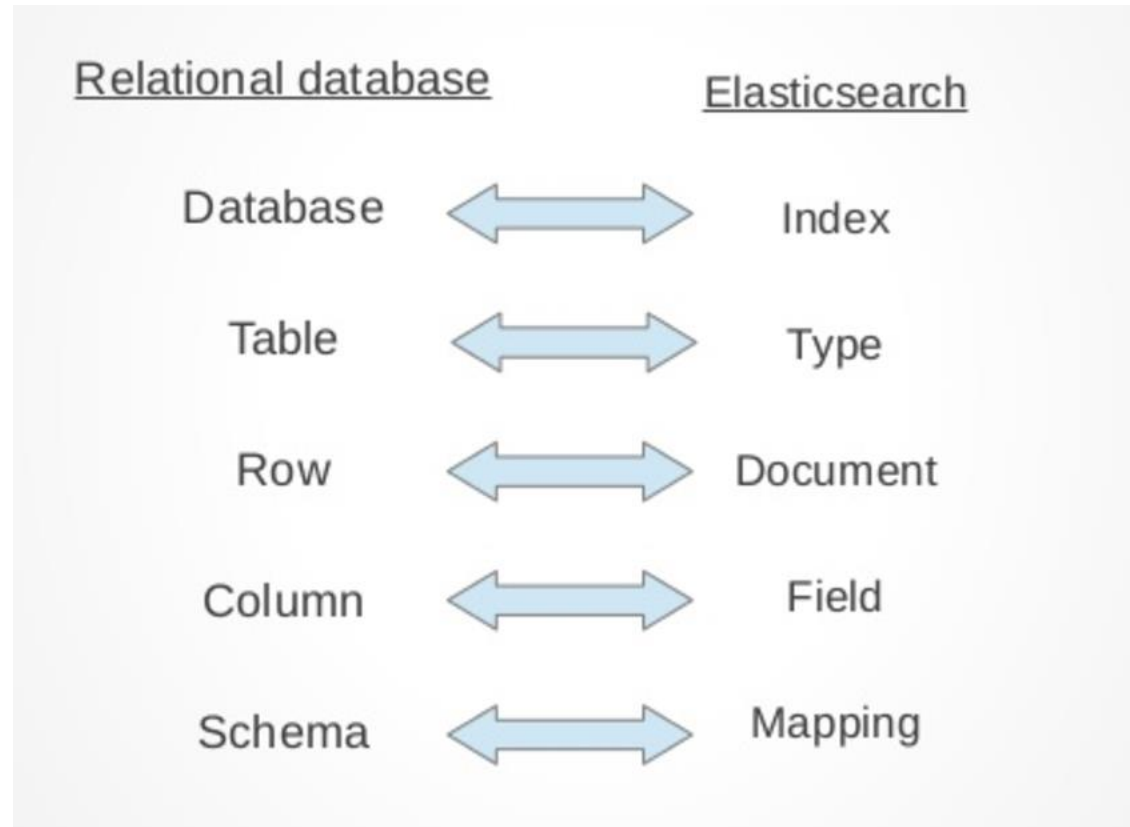
Advanced Search features

- Power query DSL, multi-language support, built in tokenizers, filters , analysers
- Full text search, auto-complete, facets, realtime search analytics

Terminologies of elastic search

- Clusters – One master node
- Nodes
- Index
- Shards
- Documents
- Indexes
- Mappings
- Types
- Query
- Analysis

W.r.t RDBMS

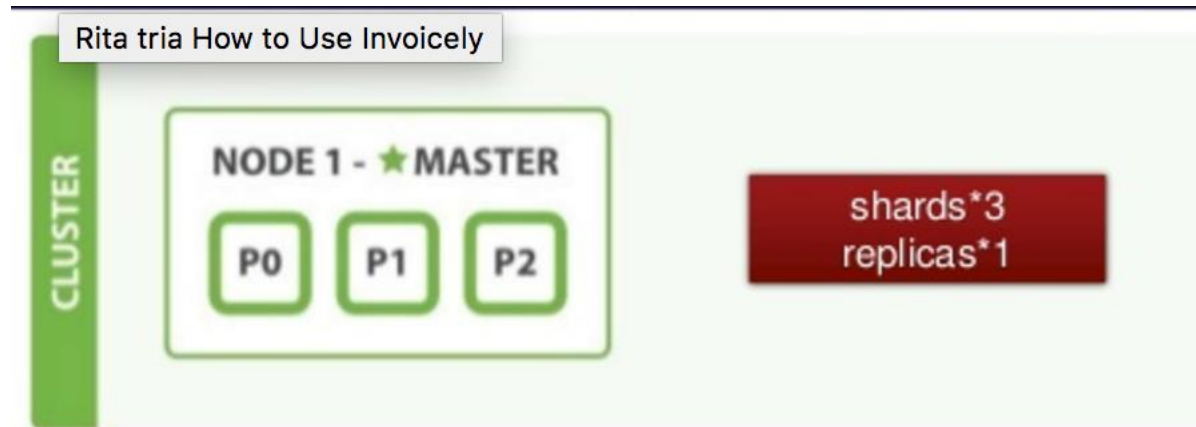


Shards and Replicas

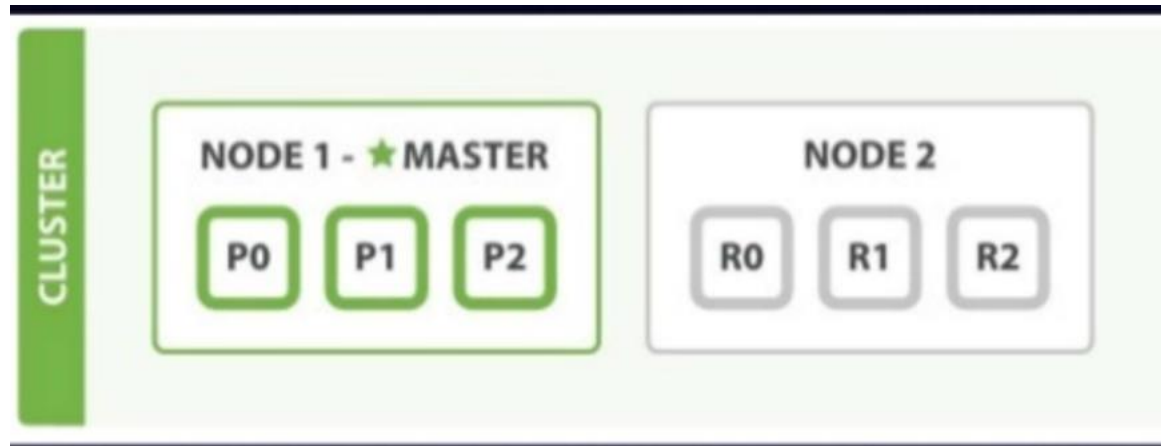
Empty Cluster



Create Index



2 nodes



3 nodes



3 nodes



Cluster after killing one node



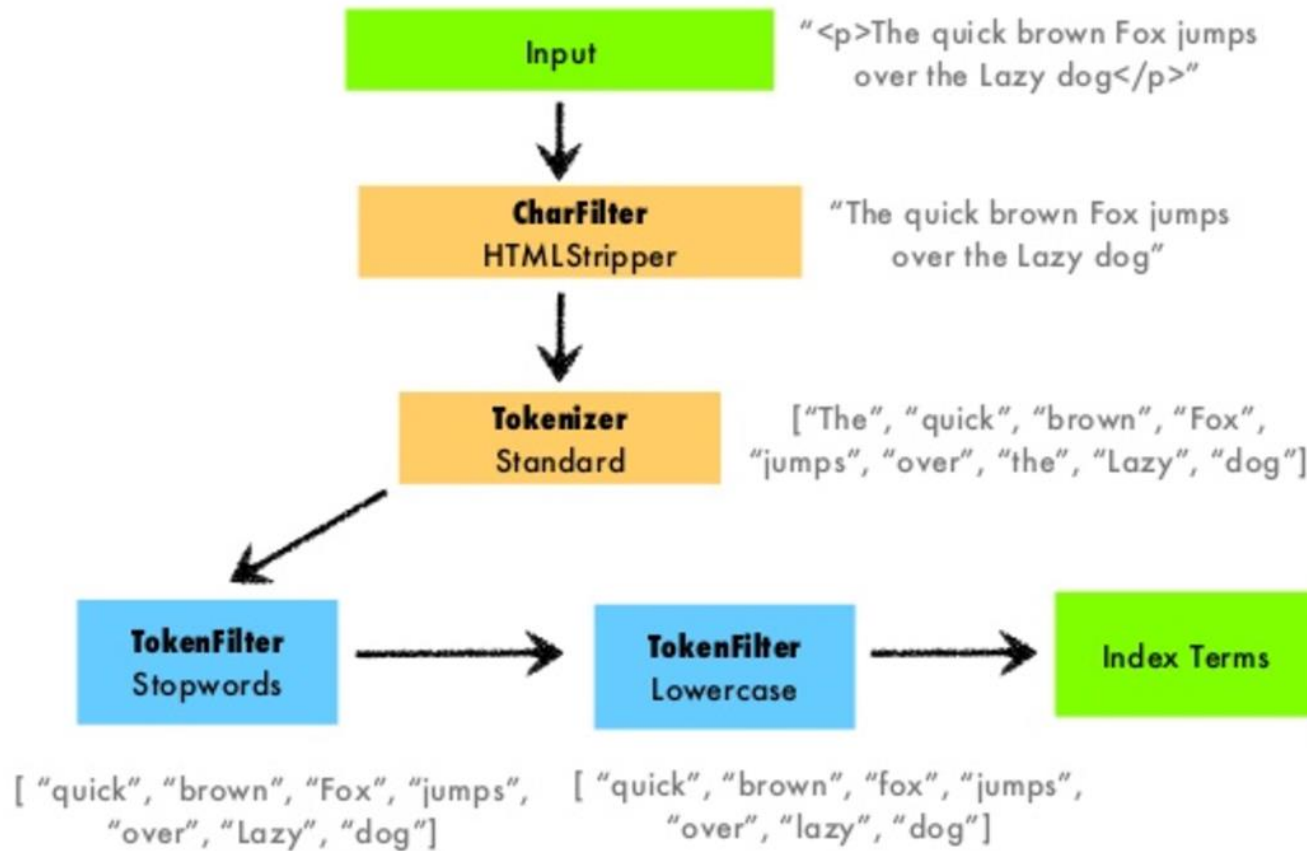
Analyzers

The process of breaking down a raw document text into terms that will be indexed in a single Lucene index

- Charfilter
- Tokenizer
- Tokenfilter

ES ships with sever common analyzer,you can also create a custom analyser

Analyzers example



Playing with elastic Search API's

- Search

- curl -XGET http://localhost:9200/my_index/test/_search
- curl -XGET http://localhost:9200/my_index/_search
- curl -XPUT http://localhost:9200/_search

- Meta Data

- curl -XPUT http://localhost:9200/my_index/_status

- Documents:

- curl -XPUT http://localhost:9200/my_index/test/1
- curl -XGET http://localhost:9200/my_index/test/1
- curl -XDELETE http://localhost:9200/my_index/test/1

Playing with elastic Search API's

```
Curl -XPUT http://localhost:9200/my\_index/test/1 -d  
{  
  "name": "joeywen",  
  "value": 100  
}
```

```
{ -  
  "_index": "my_index",  
  "_type": "test",  
  "_id": "1",  
  "_version": 1,  
  "_shards": { -  
    "total": 2,  
    "successful": 2,  
    "failed": 0  
  },  
  "created": true  
}
```

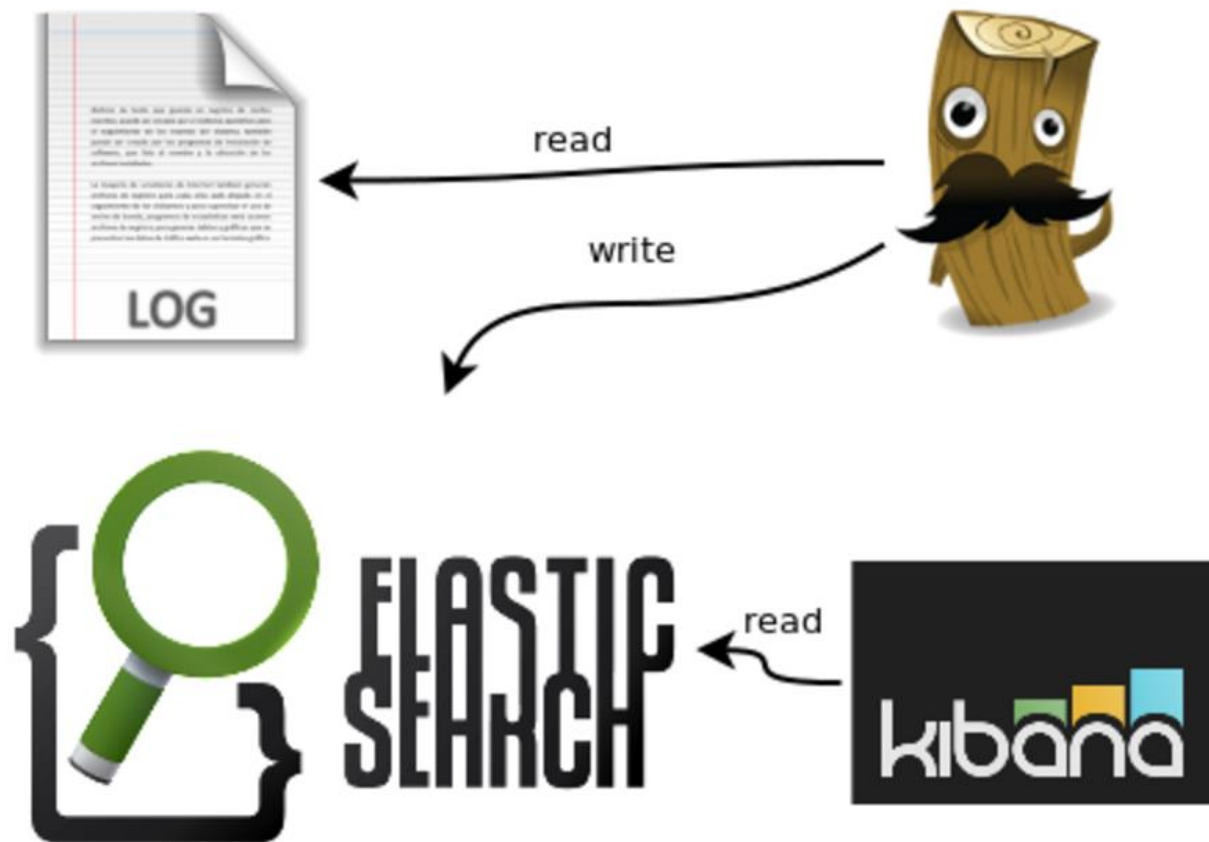

How search works?

- Search requests hit a node.
- Node broadcasts to every shard in the index
- Each shard performs its query
- Each shard returns metadata about results
- Node merges results and scores them
- Node requests documents from shard
- Results merged, sorted and returned to client

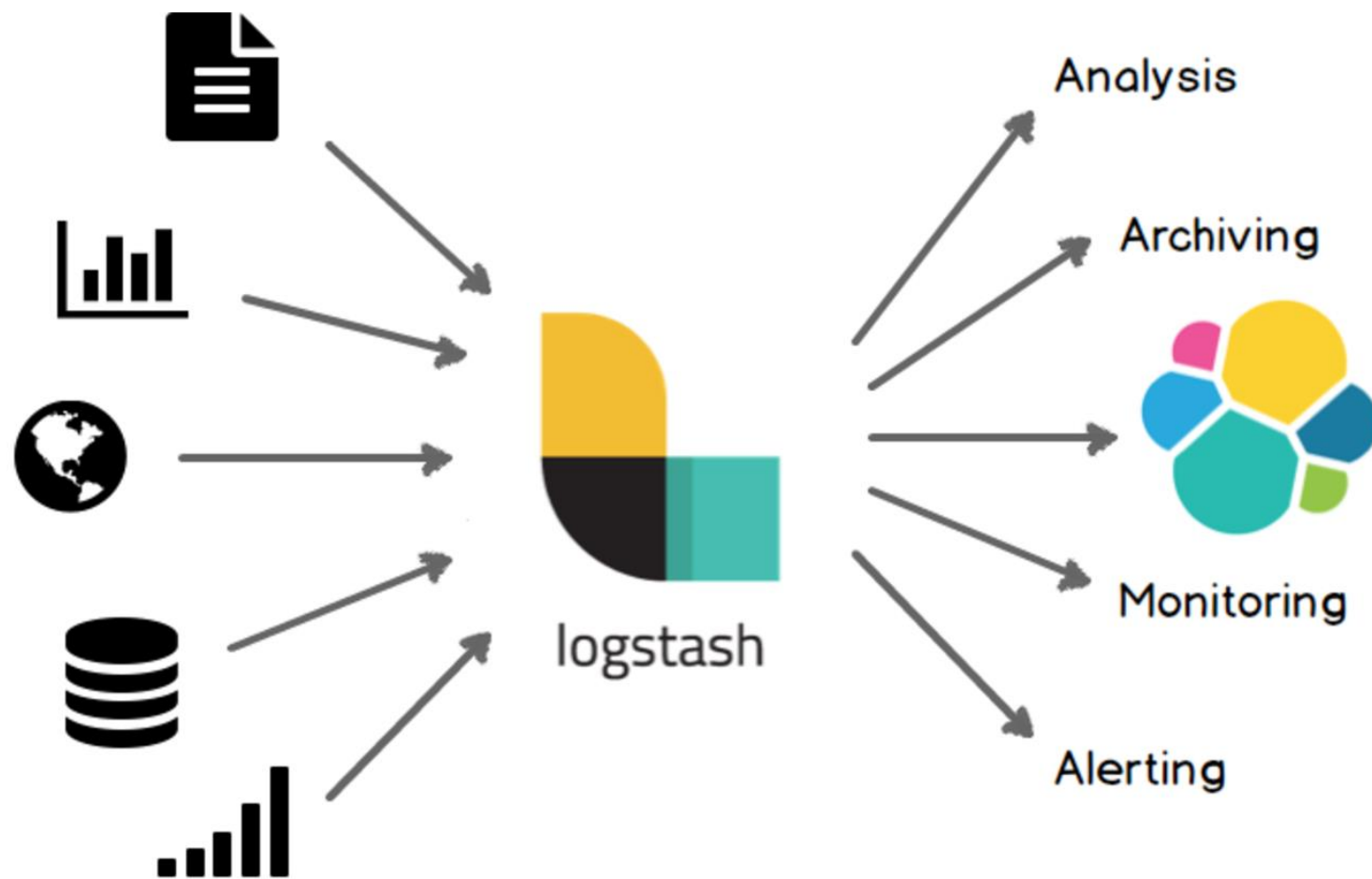
ELK Stack



ELK Stack



Logstash



Hands on