Inverted Indexing
for Text Retrieval

Three components of the web search problem

- Gathering web content
  - Web crawling
- Construction of the inverted index
  - Indexing
- Ranking documents given a query
  - Retrieval

- First two steps are typically carried out off-line
- The retrieval step needs to be operated in real time

What is inverted index?

- First, what is index?

Index

Example of inverted index

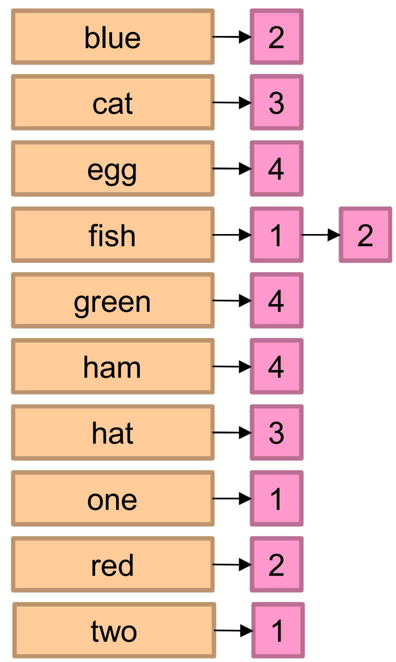
Doc 1
one fish, two fish

Doc 2
red fish, blue fish

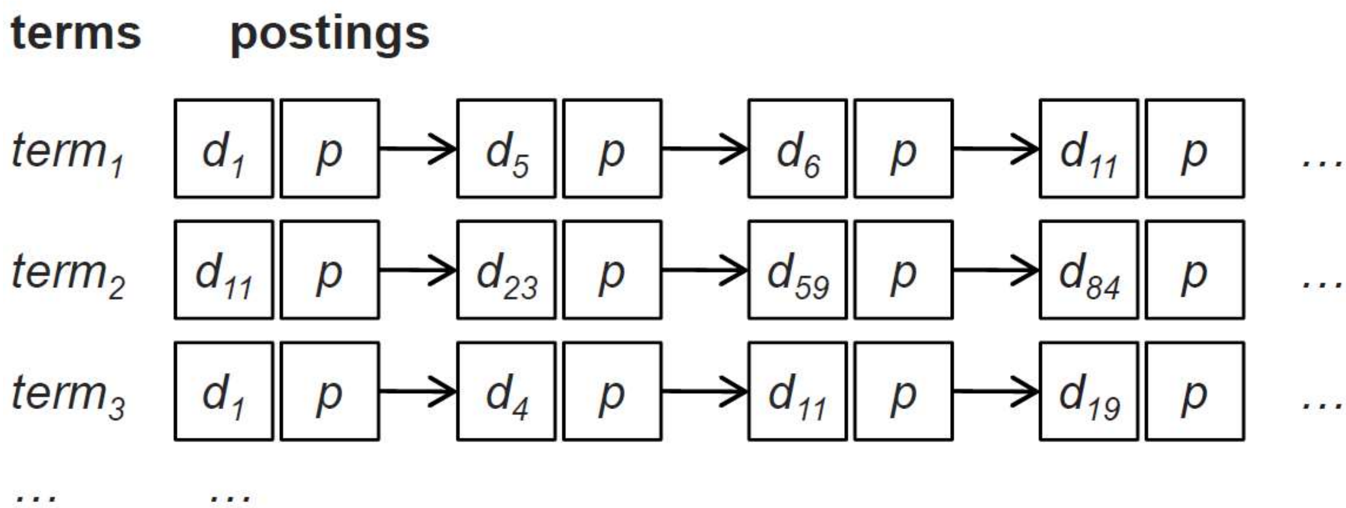
Doc 3
cat in the hat

Doc 4
green eggs and ham

	1	2	3	4
blue		1		
cat			1	
egg				1
fish	1	1		
green				1
ham				1
hat			1	
one	1			
red		1		
two	1			



More abstract view of inverted index



- An inverted index consists of posting lists
- A posting list is comprised of individual postings
  - Each posting consists of a document id and a payload
    - Payload example: the occurrence frequency of the term in the corresponding document
  - Generally, postings are sorted by document id

Baseline implementation of inverted indexing

```
class Mapper
    procedure Map(docid n, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(term t, posting ⟨n, H{t}⟩)
```

```
class Reducer
    procedure Reduce(term t, postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...])
        P ← new List
        for all posting ⟨a, f⟩ ∈ postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...] do
            P.Add(⟨a, f⟩)
        P.Sort()
        Emit(term t, postings P)
```

Illustration of the baseline algorithm

Map

Doc 1: one fish, two fish			Doc 2: red fish, blue fish			Doc 3: cat in the hat		
one	1	1	red	2	1	cat	3	1
two	1	1	blue	2	1	hat	3	1
fish	1	2	fish	2	2			

Shuffle and Sort: aggregate values by keys

Reduce

cat	3	1		
fish	1	2	2	2
one	1	1		
red	2	1		

blue	2	1
hat	3	1
two	1	1

Inverted Indexing: Pseudo-Code

class MAPPER
    procedure MAP(docid n, doc d)
        $H \leftarrow$ new ASSOCIATIVEARRAY
        for all term $t \in$ doc d **do**
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ **do**
            EMIT(term t, posting $\langle n, H\{t\}\rangle$)

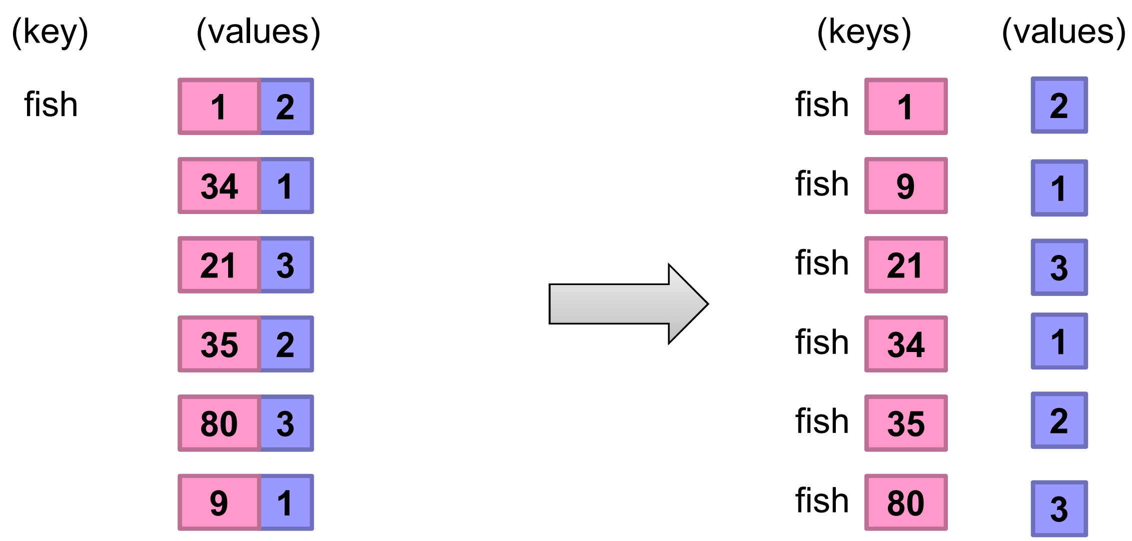
class REDUCER
    procedure REDUCE(term t, postings $[\langle a_1, f_1\rangle, \langle a_2, f_2\rangle \ldots]$)
        $P \leftarrow$ new LIST
        for all posting $\langle a, f\rangle \in$ postings $[\langle a_1, f_1\rangle, \langle a_2, f_2\rangle \ldots]$ **do**
            APPEND(P, $\langle a, f\rangle$)
        SORT(P)
        EMIT(term t, postings P)

What's the problem?

Scalability issue of the baseline implementation

- Initial implementation: terms as keys, postings as values
  - Reducers must buffer all postings associated with key (to sort)
  - What if we run out of memory to buffer postings?

Another try



- Value-to-key conversion

Revised implementation

class MAPPER
    method MAP(docid n, doc d)
        $H \leftarrow$ new ASSOCIATIVEARRAY
        for all term $t \in$ doc d **do**
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ **do**
            EMIT(tuple $\langle t, n\rangle$, tf $H\{t\}$)

class REDUCER
    method INITIALIZE
        $t_{prev} \leftarrow \emptyset$
        $P \leftarrow$ new POSTINGSLIST
    method REDUCE(tuple $\langle t, n\rangle$, tf $[f]$)
        if $t \neq t_{prev} \wedge t_{prev} \neq \emptyset$ **then**
            EMIT(term t, postings P)
            P.RESET()
        P.ADD($\langle n, f\rangle$)
        $t_{prev} \leftarrow t$
    method CLOSE
        EMIT(term t, postings P)