Data-Intensive Distributed Computing
CS 451/651 (Fall 2025)

Text Processing I
(v1.01)

Week 9: October 30

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at http://lintool.github.io/bigdata-2025f/

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International
See https://creativecommons.org/licenses/by-nc-sa/4.0/ for details

Key Questions

For lexical retrieval, how do we assignment term weights?

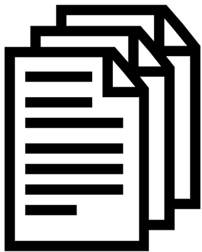
For lexical retrieval, how do we perform top-*k* retrieval efficiently?

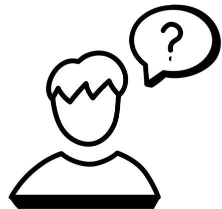
What's the problem we're trying to solve?

How to connect users with relevant information

search (information retrieval)…
… but also question answering, summarization, etc.
"information access"

… on text, images, videos, etc.
… for "everyday" searchers, domain experts, etc.



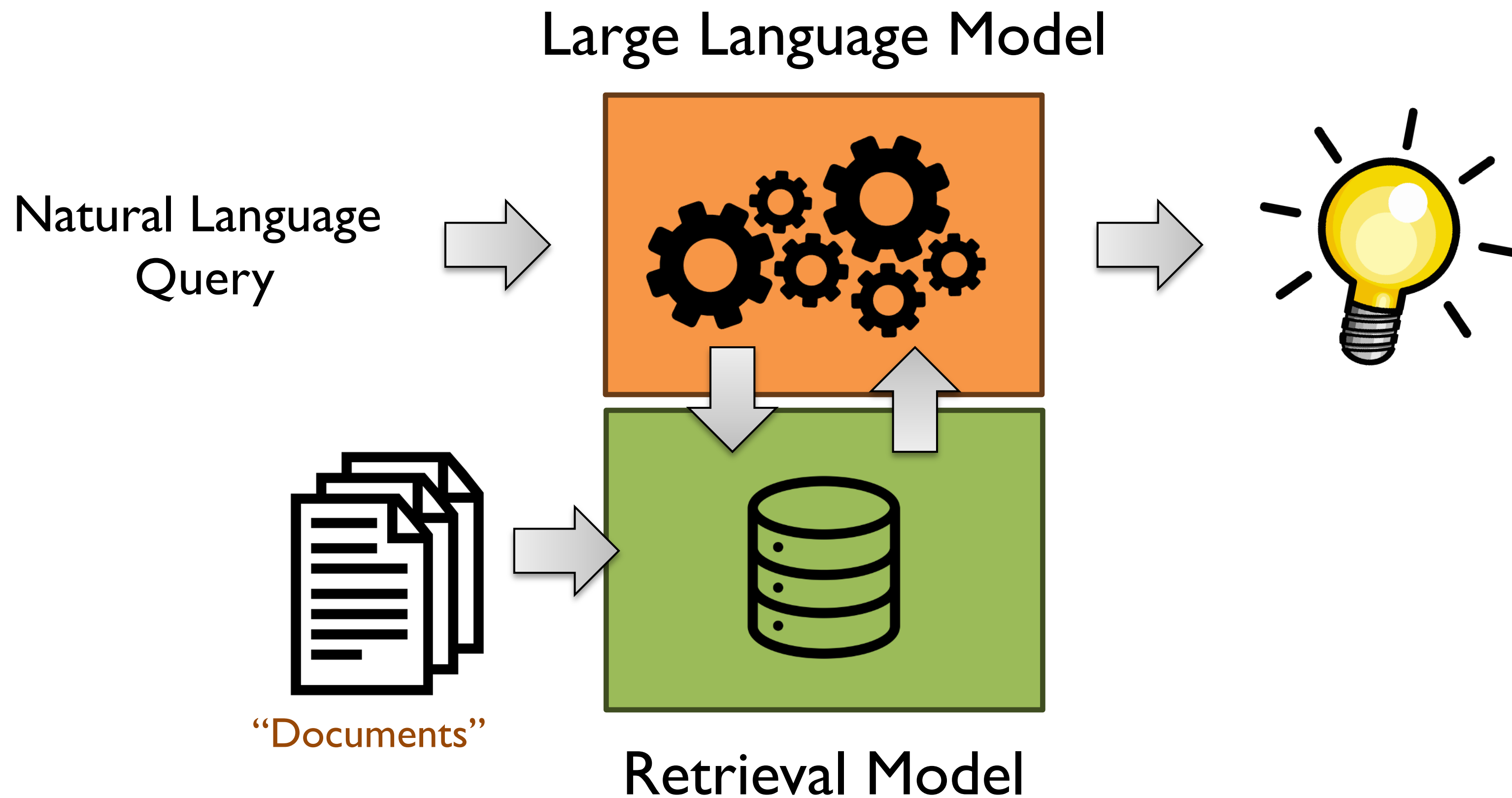


What's the problem we're trying to solve?

The "core" (text) retrieval problem

Given an information need expressed as a query q, the text retrieval task is to return a ranked list of k texts $\{d_1, d_2 \dots d_k\}$ from an arbitrarily large but finite collection of texts $C = \{d_i\}$ that maximizes a metric of interest, for example, precision, nDCG, AP, etc.

Retrieval-Augmented Generation



A Vector Space Model for Automatic Indexing

1. Document Space Configurations

Consider a document space consisting of documents D_i, each identified by one or more index terms T_j;

index terms are present. In that case, each document D_i is represented by a t-dimensional vector

$$D_i = (d_{i1}, d_{i2}, \ldots, d_{it}),$$

d_{ij} representing the weight of the jth term.

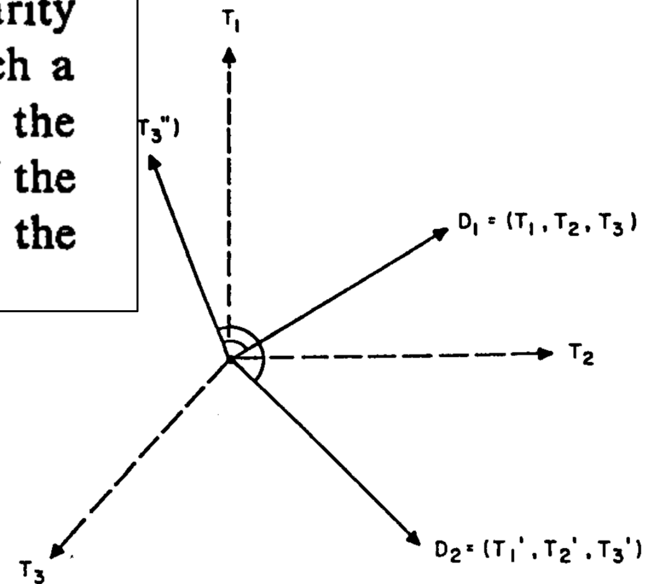
Given the index vectors for two documents, it is possible to compute a similarity coefficient between them, $s(D_i, D_j)$, which reflects the degree of similarity in the corresponding terms and term weights. Such a similarity measure might be the inner product of the two vectors, or alternatively an inverse function of the angle between the corresponding vector pairs; when the

of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

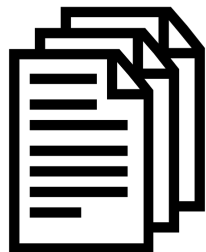
This study was supported in part by the National Science Foundation under grant GN 43505. Authors' addresses: G. Salton and A. Wong, Department of Computer Science, Cornell University, Ithaca, NY 14850; C. S. Yang, Department of Computer Science, The University of Iowa, Iowa City, IA, 52240.

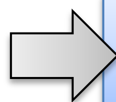
[1] Although we speak of documents and index terms, the present development applies to any set of entities identified by weighted property vectors.

[2] Retrieval performance is often measured by parameters such as *recall* and *precision*, reflecting the ratio of relevant items actually retrieved and of retrieved items actually relevant. The question concerning optimum space configurations may then be more conventionally expressed in terms of the relationship between document indexing, on the one hand, and retrieval performance, on the other.

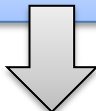


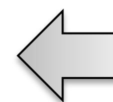
Salton et al. (1975) A Vector Space Model for Automatic Indexing. *Communications of the ACM*, 18(11):613-620.

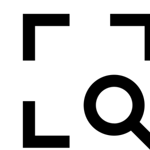




Term Weighting







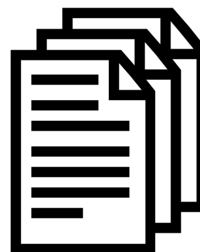
The Manhattan Project and its atomic bomb helped bring an end to World War II. Its legacy of peaceful uses of atomic energy continues to have an impact on history and science.

```
{'atom': 4.0140, 'bomb': 4.0704, 'bring': 2.7239, 'continu':
2.4331, 'end': 2.1559, 'energi': 2.5045, 'have': 1.0742, 'help':
1.8157, 'histori': 2.4213, 'ii': 3.0998, 'impact': 3.0304, 'it':
2.0473, 'legaci': 4.1335, 'manhattan': 4.1345, 'peac': 3.5205,
'project': 2.6442, 'scienc': 2.8700, 'us': 0.9967, 'war':
2.6454, 'world': 1.9974}
```

"bag of words"

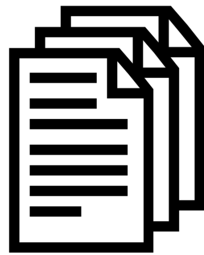
"lexical"

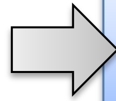
sparse vector



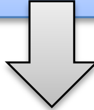
“Documents”

$$\text{BM25}(q,d) = \sum_{t \in q \cap d} \log \frac{N - \text{df}(t) + 0.5}{\text{df}(t) + 0.5} \cdot \frac{\text{tf}(t,d) \cdot (k_1 + 1)}{\text{tf}(t,d) + k_1 \cdot (1 - b + b \cdot \frac{l_d}{L})}$$

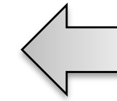




Term Weighting



Query



[. . .]

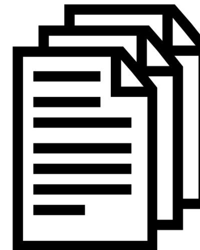
The Manhattan Project and its atomic bomb helped bring an end to World War II. Its legacy of peaceful uses of atomic energy continues to have an impact on history and science.

{'atom': 4.0140, 'bomb': 4.0704, 'bring': 2.7239, 'continu': 2.4331, 'end': 2.1559, 'energi': 2.5045, 'have': 1.0742, 'help': 1.8157, 'histori': 2.4213, 'ii': 3.0998, 'impact': 3.0304, 'it': 2.0473, 'legaci': 4.1335, 'manhattan': 4.1345, 'peac': 3.5205, 'project': 2.6442, 'scienc': 2.8700, 'us': 0.9967, 'war': 2.6454, 'world': 1.9974}

“bag of words”

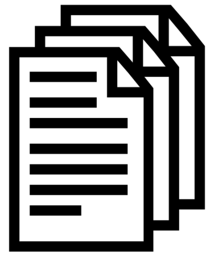
“lexical”

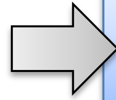
sparse vector



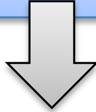
Results

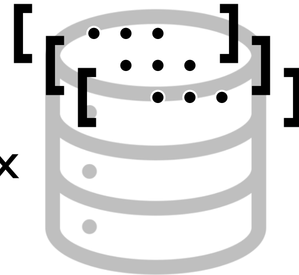
"Documents"





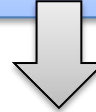
Term Weighting



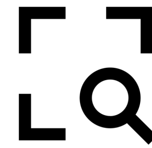


Inverted Index

Multi-hot

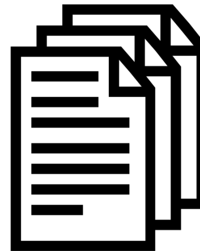


Query



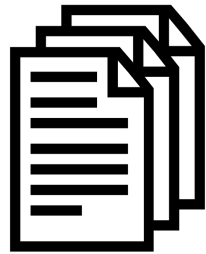
atomic bomb

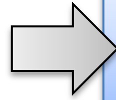
{'atom': 1, 'bomb': 1}



Results

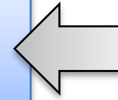
“Documents”



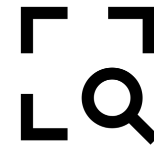


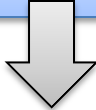
Term Weighting

Multi-hot



Query

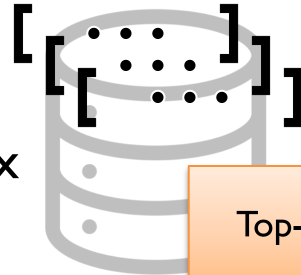




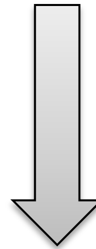
atomic bomb

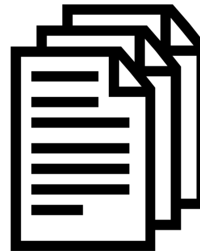
{'atom': 1, 'bomb': 1}

Inverted Index

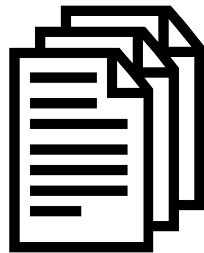


Top-*k* Retrieval

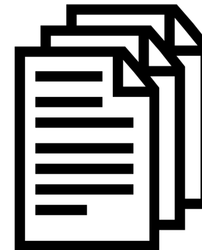




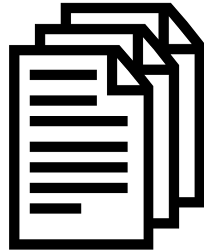
Results



{'atom': 4.0140, 'bomb': 4.0704, 'bring': 2.7239, 'continu': 2.4331, 'end': 2.1559, 'energi': 2.5045, 'have': 1.0742, 'help': 1.8157, 'histori': 2.4213, 'ii': 3.0998, 'impact': 3.0304, 'it': 2.0473, 'legaci': 4.1335, 'manhattan': 4.1345... }



“Documents”



Term Weighting

Multi-hot

Query

atomic bomb

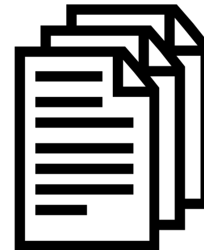
{'atom': 1, 'bomb': 1}

The Manhattan Project and its atomic bomb helped bring an end to World War II…

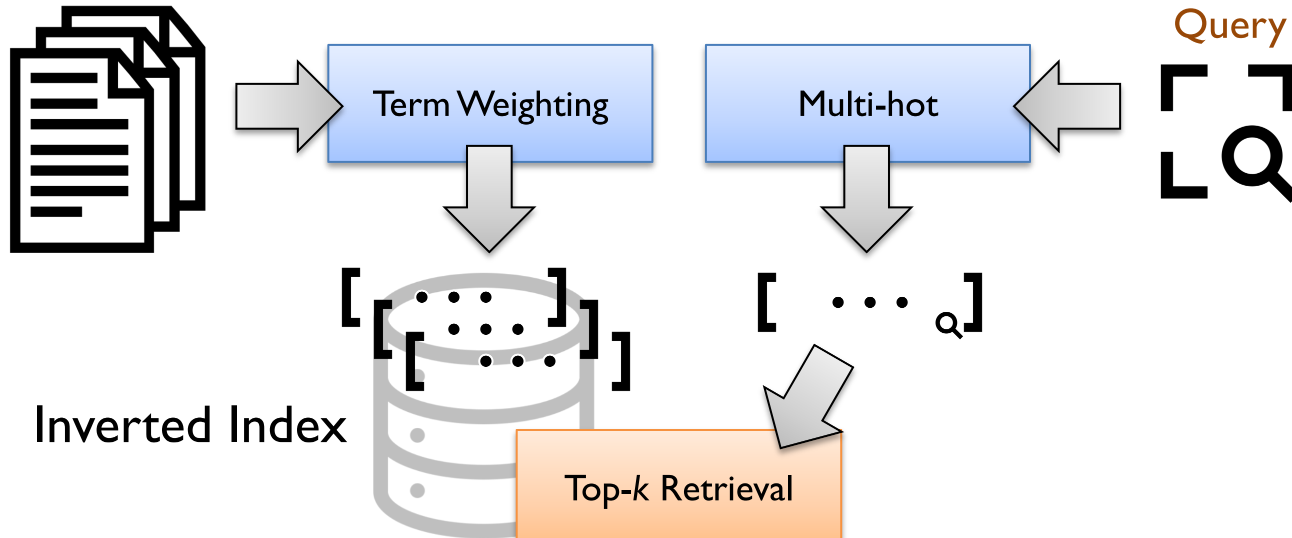
Top-*k* Retrieval

{'atom': 4.0140, 'bomb': 4.0704, 'bring': 2.7239, 'continu': 2.4331, 'end': 2.1559, 'energi': 2.5045, 'have': 1.0742, 'help': 1.8157, 'histori': 2.4213, 'ii': 3.0998, 'impact': 3.0304, 'it': 2.0473, 'legaci': 4.1335, 'manhattan': 4.1345... }

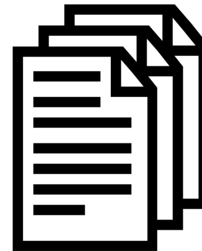
inner (dot) product



Results

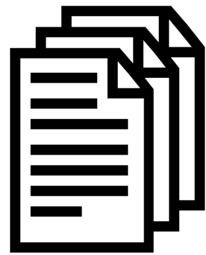


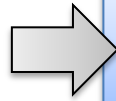
tl;dr – retrieval ~ computing dot products on vector representations!



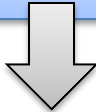
Results

“Documents”

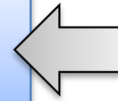




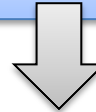
Term Weighting

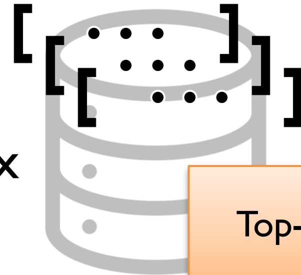


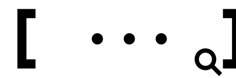
Multi-hot



Query

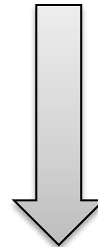






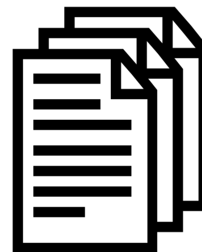
Inverted Index

Top-*k* Retrieval



Challenge #1: What should the term weights be?

Challenge #2: How do we perform top-*k* retrieval efficiently?



Results

Lexical Retrieval: Challenges

The Manhattan Project and its atomic bomb helped bring an end to World War II. Its legacy of peaceful uses of atomic energy continues to have an impact on history and science.

```
{'atom': 4.0140, 'bomb': 4.0704, 'bring': 2.7239, 'continu':
2.4331, 'end': 2.1559, 'energi': 2.5045, 'have': 1.0742, 'help':
1.8157, 'histori': 2.4213, 'ii': 3.0998, 'impact': 3.0304, 'it':
2.0473, 'legaci': 4.1335, 'manhattan': 4.1345, 'peac': 3.5205,
'project': 2.6442, 'scienc': 2.8700, 'us': 0.9967, 'war':
2.6454, 'world': 1.9974}
```

Challenge #1: What should the term weights be?

Challenge #2: How do we perform top-*k* retrieval efficiently?

Let's start with boolean retrieval….

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			

What goes in each cell?

- boolean
- count
- positions

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			

Indexing: building this structure

Retrieval: using it to perform top-*k* retrieval

How?

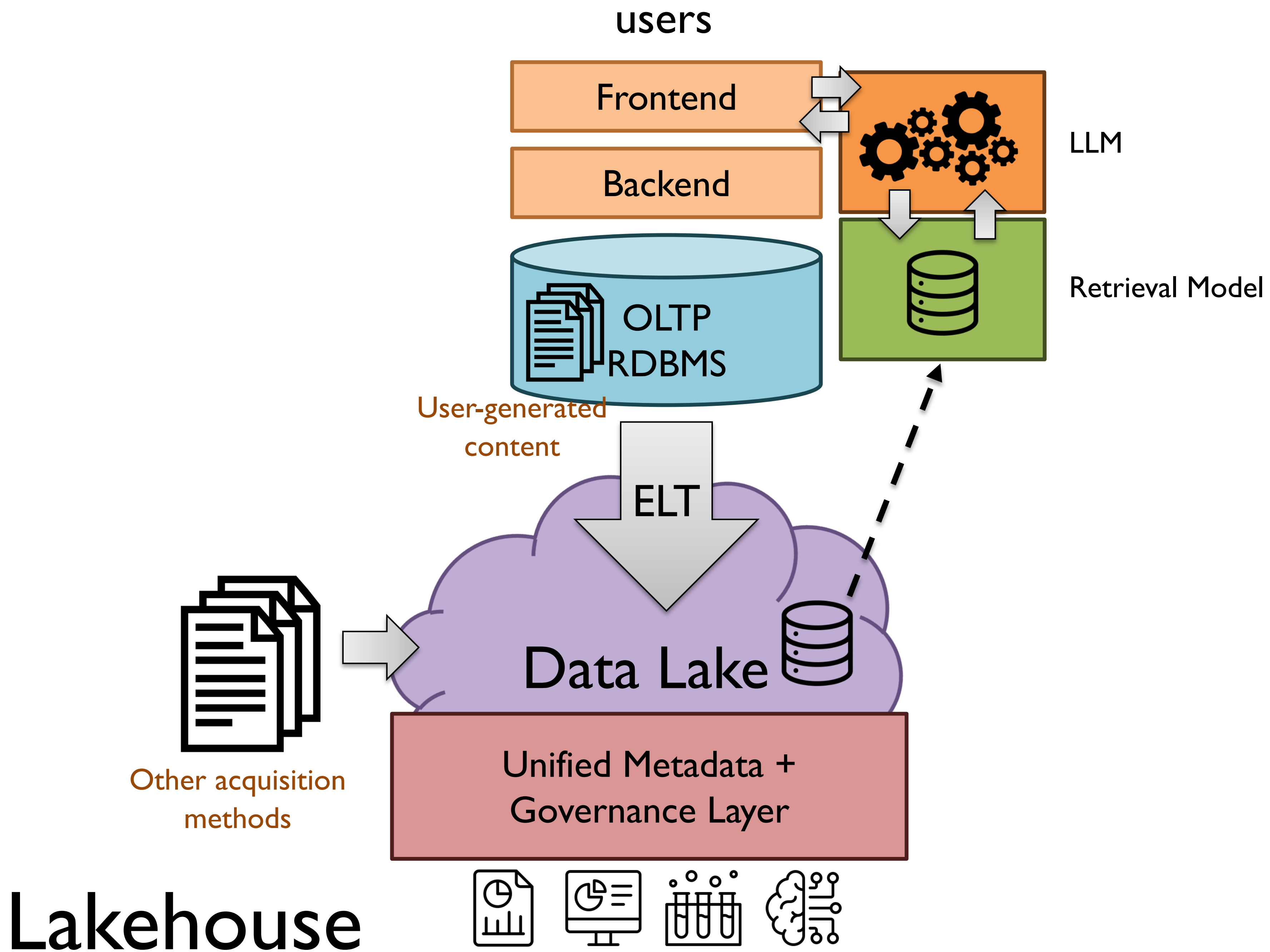
(1) Let's learn how to actually do it…

~~(2) Call an external package~~

Inverted Indexing in MapReduce

One of our most significant uses of MapReduce to date has been a complete rewrite of the production indexing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations.

Dean and Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI 2004*.


users
Frontend
Backend
LLM
Retrieval Model
OLTP
RDBMS
User-generated
content
ELT
Data Lake
Other acquisition
methods
Unified Metadata +
Governance Layer
Lakehouse

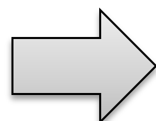
Doc 1
one fish, two fish

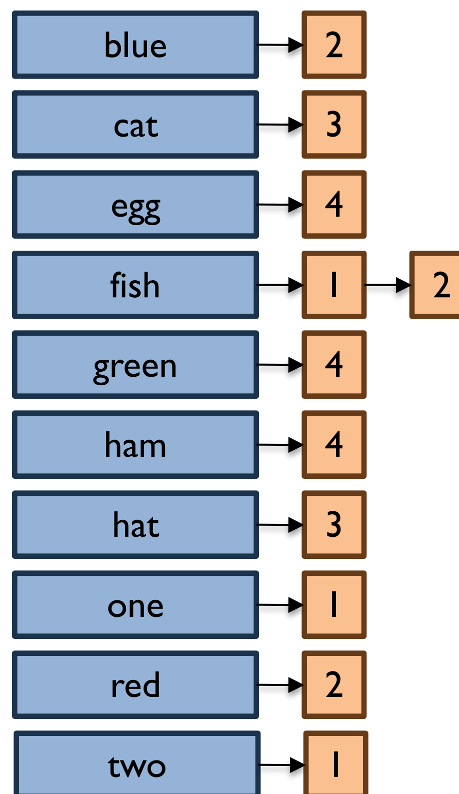
Doc 2
red fish, blue fish

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			





postings lists

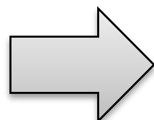
Doc 1
one fish, two fish

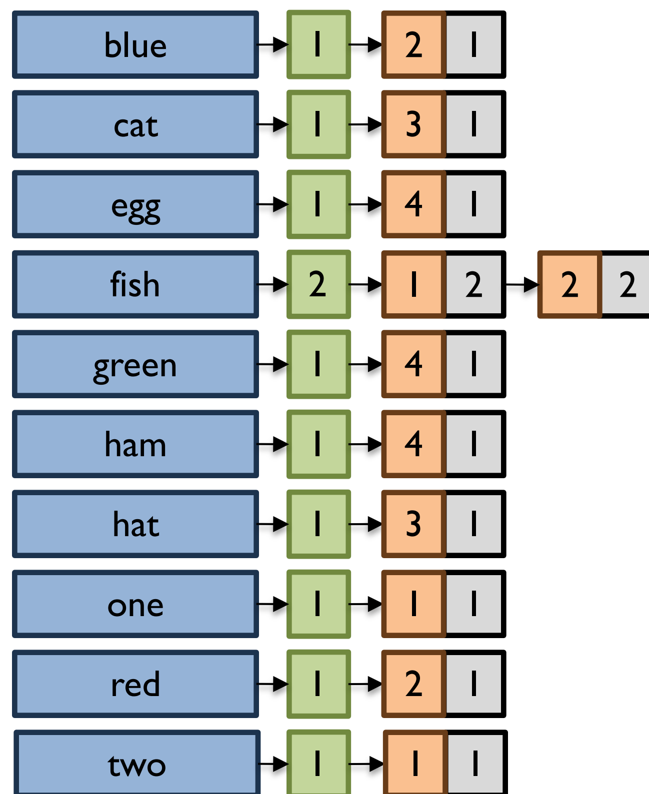
Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

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





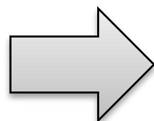
Doc 1
one fish, two fish

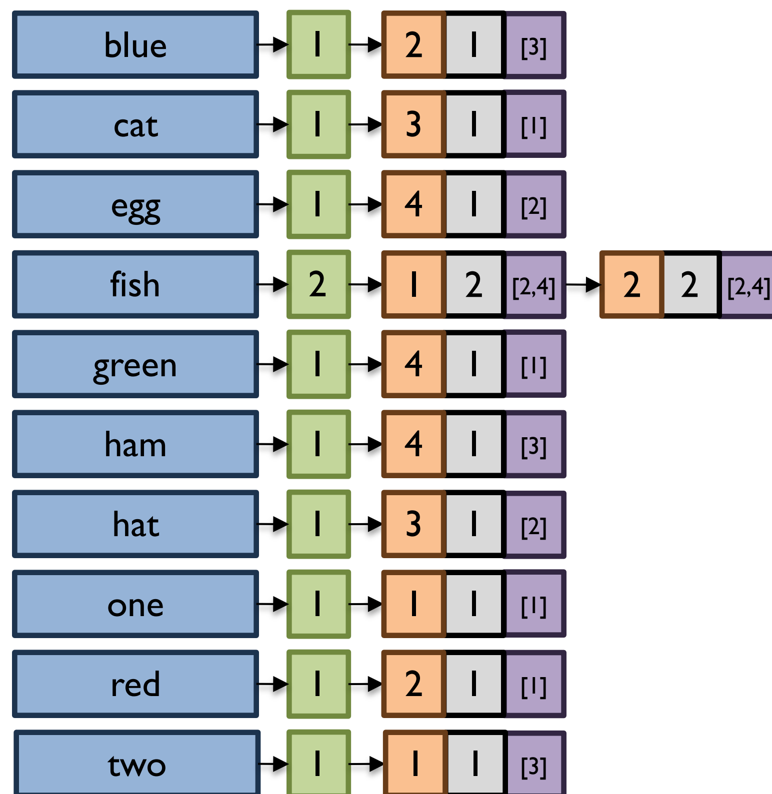
Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

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





MapReduce: Index Construction

Map over all documents

Emit *term* as key, (*docid*, *tf*) as value

Emit other information as necessary (e.g., term position)

Sort/shuffle: group postings by term

Reduce

Gather and sort the postings (typically by *docid*)

Write postings to disk

MapReduce does all the heavy lifting!

Inverted Indexing with MapReduce

Map

	Doc 1: one fish, two fish	
one	1	1
two	1	1
fish	1	2

	Doc 2: red fish, blue fish	
red	2	1
blue	2	1
fish	2	2

	Doc 3: cat in the hat	
cat	3	1
hat	3	1

Shuffle and Sort: aggregate values by keys

Reduce

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

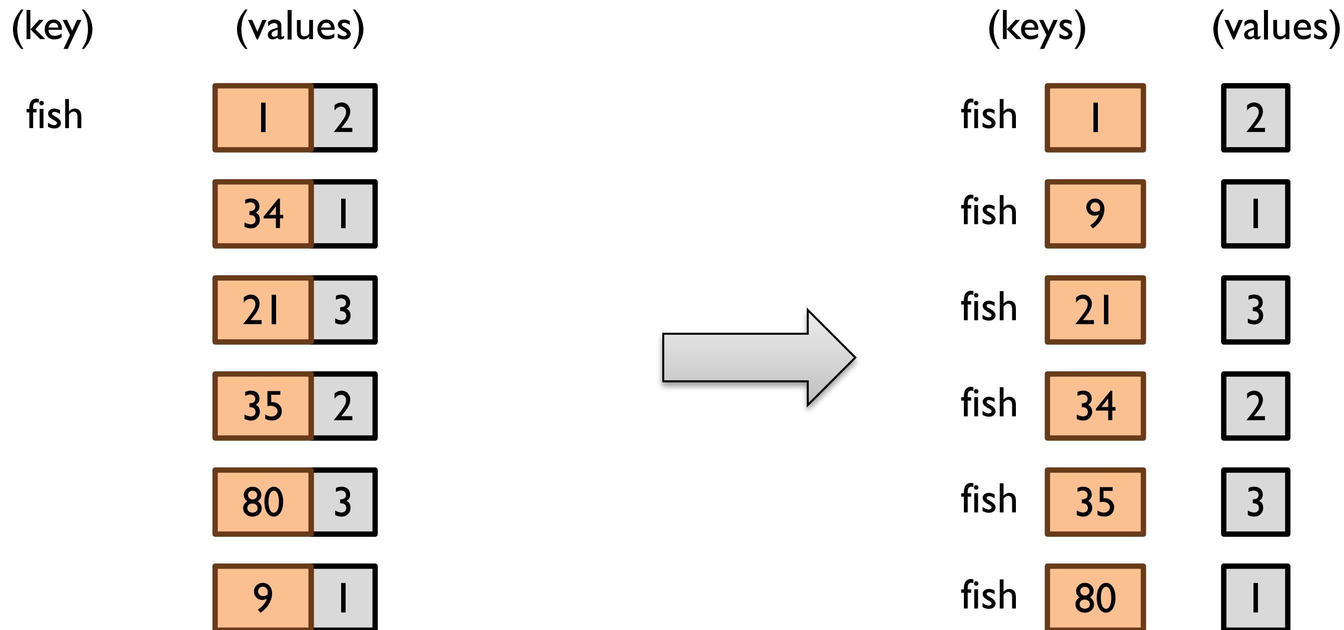
blue	2	1
hat	3	1
two	1	1

Inverted Indexing: Pseudo-Code

```
class Mapper {
  def map(docid: Long, doc: String) = {
    val counts = new Map()
    for (term <- tokenize(doc)) {
      counts(term) += 1
    }
    for ((term, tf) <- counts) {
      emit(term, (docid, tf))
    }
  }
}

class Reducer {
  def reduce(term: String, postings: Iterable[(docid, tf)]) = {
    val p = new List()
    for ((docid, tf) <- postings) {
      p.append((docid, tf))
    }
    p.sort()
    emit(term, p)
  }
}
```

What's the inefficiency?

Another Try…



How is this different?

Let the framework do the sorting!

Inverted Indexing: Pseudo-Code

```
class Mapper {
  def map(docid: Long, doc: String) = {
    val counts = new Map()
    for (term <- tokenize(doc)) {
      counts(term) += 1
    }
    for ((term, tf) <- counts) {
      emit((term, docid), tf)
    }
  }
}

class Reducer {
  var prev = null
  val postings = new PostingsList()

  def reduce(key: Pair, tf: Iterable[Int]) = {
    if key.term != prev and prev != null {
      emit(prev, postings)
      postings.reset()
    }
    postings.append(key.docid, tf.first)
    prev = key.term
  }

 def cleanup() = {
   emit(prev, postings)
 }
}
```

Wait, how's this any better?

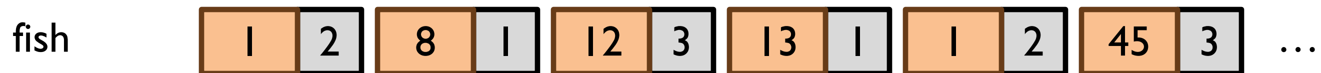
What else do we need to do?

Postings Encoding

Conceptually:

In Practice:

Don't encode docids, encode gaps (or *d*-gaps)
But it's not obvious that this save space…



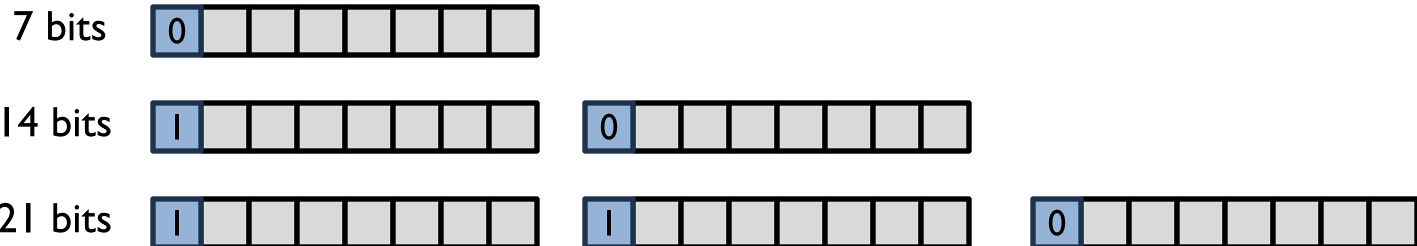
= delta encoding, delta compression, gap compression

VByte

Simple idea: use only as many bytes as needed

Need to reserve one bit per byte as the "continuation bit"

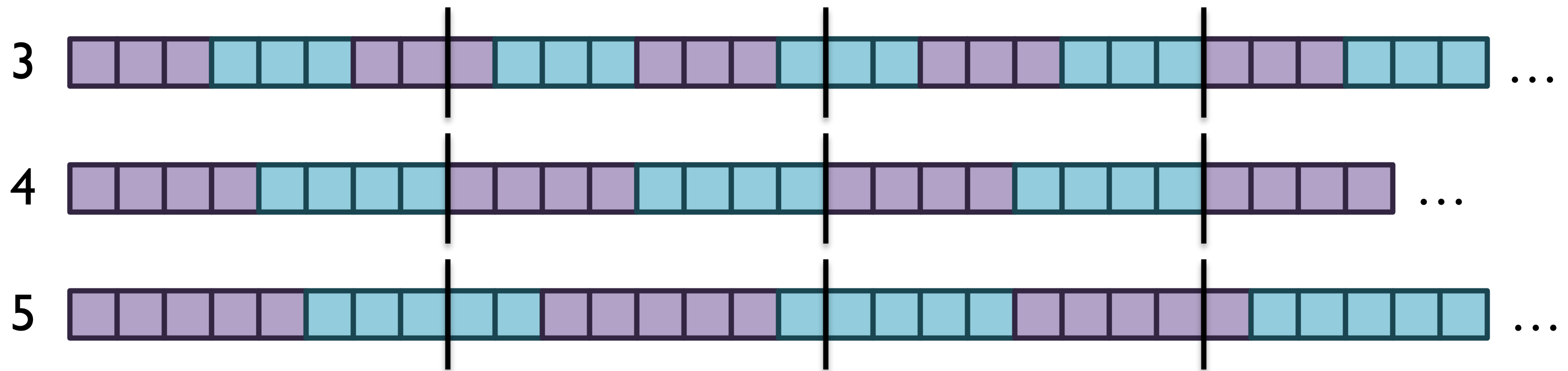
Use remaining bits for encoding value



Works okay, easy to implement…

Bit Packing

What's the smallest number of bits we need to code a block (=128) of integers?



Efficient decompression with hard-coded decoders

PForDelta – bit packing + separate storage of "overflow" bits

Inverted Indexing: IP (~Pairs)

```
class Mapper {
  def map(docid: Long, doc: String) = {
    val counts = new Map()
    for (term <- tokenize(doc)) {
      counts(term) += 1
    }
    for ((term, tf) <- counts) {
      emit((term, docid), tf)
    }
  }
}

class Reducer {
  var prev = null
  val postings = new PostingsList()

  def reduce(key: Pair, tf: Iterable[Int]) = {
    if key.term != prev and prev != null {
      emit(key.term, postings)
      postings.reset()
    }
    postings.append(key.docid, tf.first)
    prev = key.term
  }

 def cleanup() = {
   emit(prev, postings)
 }
}
```

Apply compression here…

Merging Postings

Let's define an operation $\oplus$ on postings lists *P*:

Postings(1, 15, 22, 39, 54) $\oplus$ Postings(2, 46)
= Postings(1, 2, 15, 22, 39, 46, 54)

Apply compression as needed!

What have we done?

Super powers:
Associativity and Commutativity!

The Power of Associativity

You can put parentheses wherever you want!

$$(v_1 \oplus v_2 \oplus v_3) \oplus (v_4 \oplus v_5 \oplus v_6 \oplus v_7) \oplus (v_8 \oplus v_9)$$

$$(v_1 \oplus v_2) \oplus (v_3 \oplus v_4 \oplus v_5) \oplus$$

$$(v_1 \oplus v_2 \oplus (v_3 \oplus v_4 \oplus v_5)) \oplus$$

Credit to Oscar Boykin for the idea behind these slides

The Power of Commutativity

You can swap order of operands however you want!

$$(v_1 \oplus v_2 \oplus v_3) \oplus (v_4 \oplus v_5 \oplus v_6 \oplus v_7) \oplus (v_8 \oplus v_9)$$

$$(v_4 \oplus v_5 \oplus v_6 \oplus v_7) \oplus (v_1 \oplus v_2 \oplus v_3) \oplus (v_8 \oplus v_9)$$

$$(v_8 \oplus v_9) \oplus (v_4 \oplus v_5 \oplus v_6 \oplus v_7) \oplus (v_1 \oplus v_2 \oplus v_3)$$

Credit to Oscar Boykin for the idea behind these slides

Merging Postings

Let's define an operation $\oplus$ on postings lists *P*:

Postings(1, 15, 22, 39, 54) $\oplus$ Postings(2, 46)
= Postings(1, 2, 15, 22, 39, 46, 54)

Apply compression as needed!

What have we done?

Then we can rewrite our indexing algorithm!

flatMap: emit singleton postings

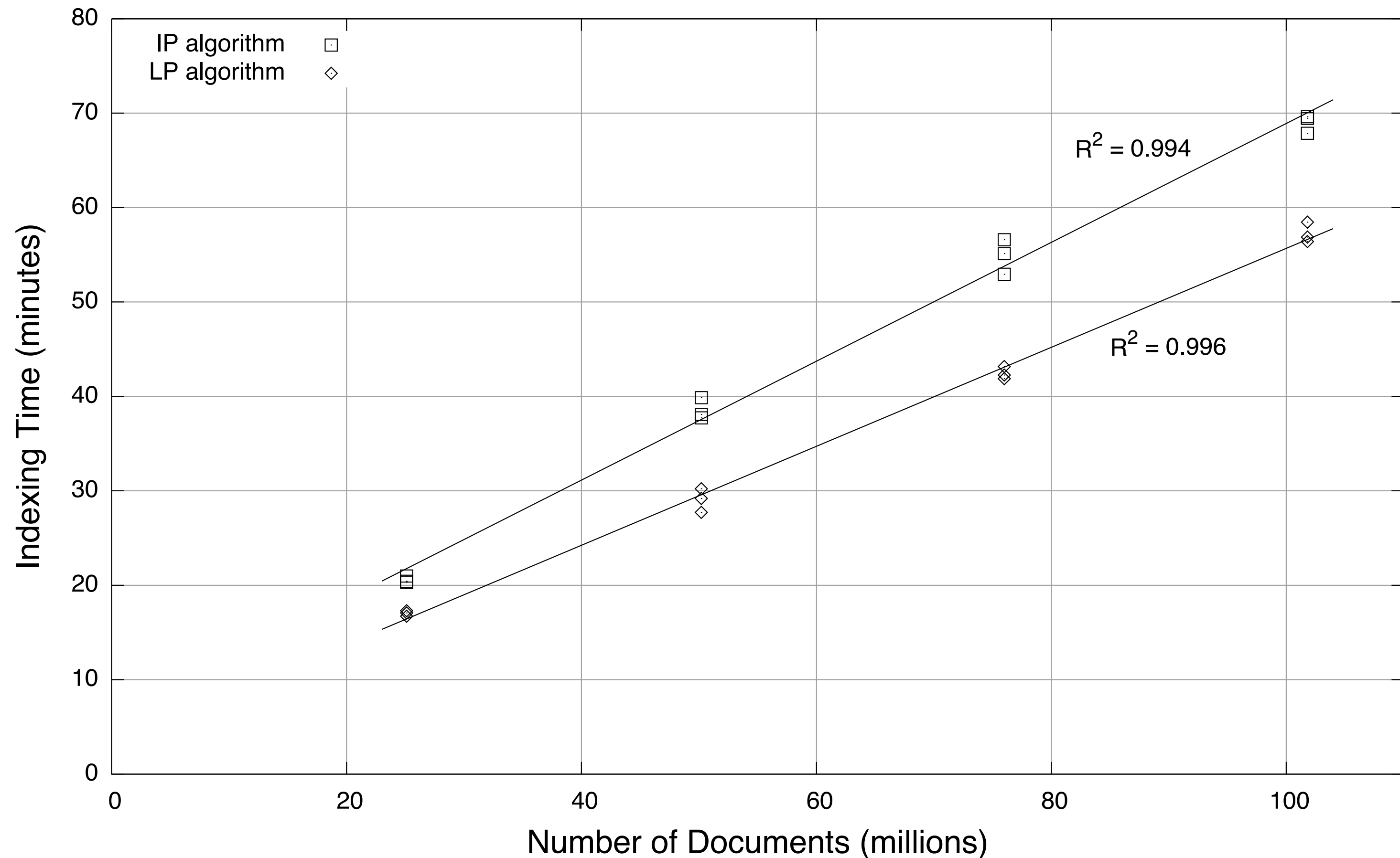
reduceByKey: $\oplus$

Inverted Indexing: LP (~Stripes)

```
class Mapper {
  def map(docid: Long, doc: String) = {
    val counts = new Map()
    for (term <- tokenize(doc)) {
      counts(term) += 1
    }
    for ((term, tf) <- counts) {
      emit(term, new PostingsList((docid, tf)))
    }
  }
}

class Reducer {
  def reduce(term: String, lists: Iterable[PostingsList]) = {
    var f = new PostingsList()

    for (list <- lists) {
      f = f + list
    }

    emit(term, f)
  }
}
```

LP vs. IP?

Experiments on ClueWeb09 collection: segments 1 + 2
101.8m documents (472 GB compressed, 2.97 TB uncompressed)



Alg.	Time	Intermediate Pairs	Intermediate Size
IP	38.5 min	13×10^9	306×10^9 bytes
LP	29.6 min	614×10^6	85×10^9 bytes

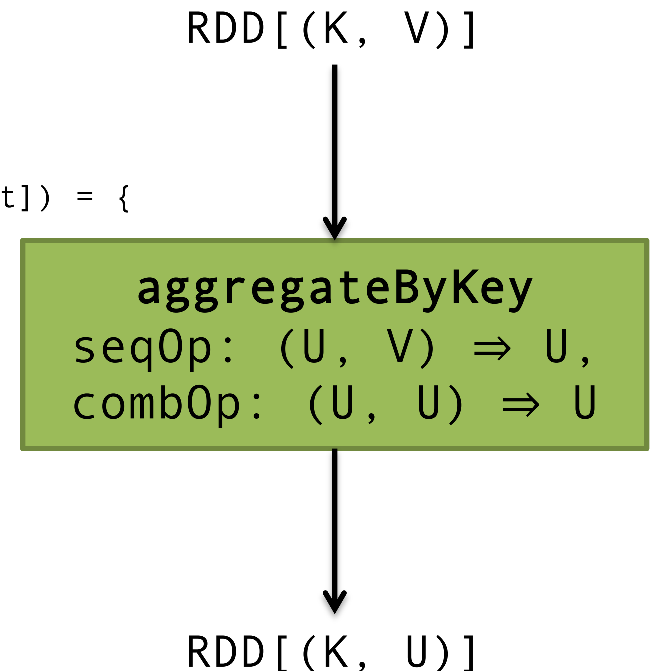
From: Elsayed et al., Brute-Force Approaches to Batch Retrieval: Scalable Indexing with MapReduce, or Why Bother? 2010

Another Look at LP

flatMap: emit singleton postings
reduceByKey: ⊕

```
class Mapper {
  def map(docid: Long, doc: String) = {
    val counts = new Map()
    for (term <- tokenize(doc)) {
      counts(term) += 1
    }
    for ((term, tf) <- counts) {
      emit(term, new PostingsList((docid, tf)))
    }
  }
}

class Reducer {
  def reduce(term: String, lists: Iterable[PostingsList]) = {
    var f = new PostingsList()

    for (list <- lists) {
      f = f + list
    }

    emit(term, f)
  }
}
```

Remind you of anything in Spark?



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			

Indexing: building this structure

Retrieval: using it top perform top-*k* retrieval

How?

(1) Let's learn how to actually do it…

~~(2) Call an external package~~

users

Frontend

Backend

LLM

Retrieval Model

OLTP
RDBMS

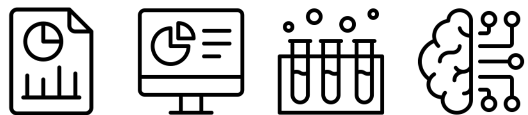
User-generated
content

ELT

Data Lake

Other acquisition
methods

Unified Metadata +
Governance Layer



Lakehouse

MapReduce it?

Perfect for MapReduce!

The indexing problem

Scalability is critical

Must be relatively fast, but need not be real time

Fundamentally a batch operation

Incremental updates may or may not be important

For the web, crawling is a challenge in itself

The retrieval problem

Must have sub-second response time

For the web, only need relatively few results

Uh… not so good…

Assume everything fits in memory on a single machine…

(For now)

Boolean Retrieval

Users express queries as a boolean expression

AND, OR, NOT

Can be arbitrarily nested

Retrieval is based on the notion of sets

Any query divides the collection into two sets: retrieved, not-retrieved

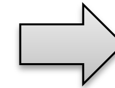
Pure boolean systems do not define an ordering of the results

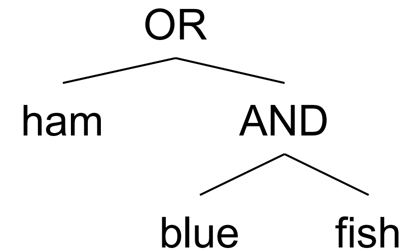
Boolean Retrieval

To execute a boolean query:

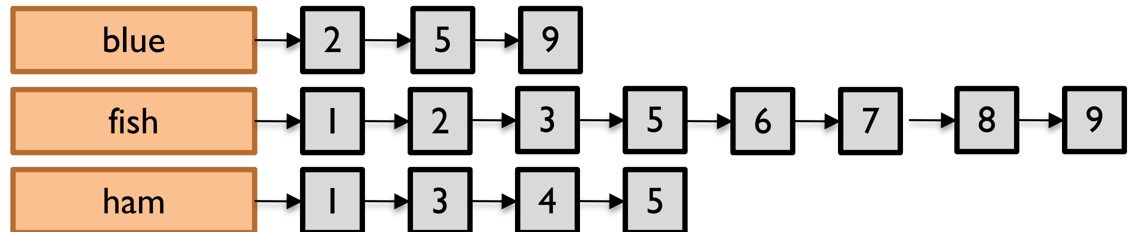
Build query syntax tree

(blue AND fish) OR ham





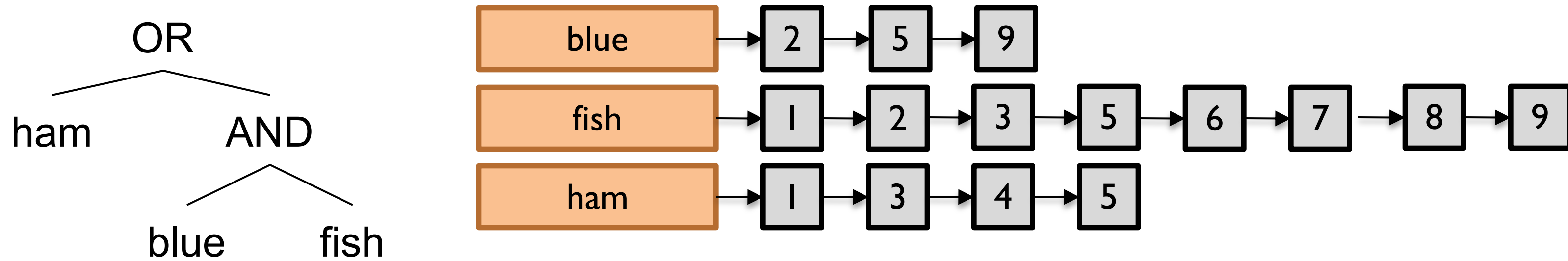
For each clause, look up postings

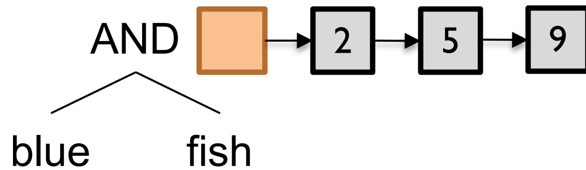


Traverse postings and apply boolean operator

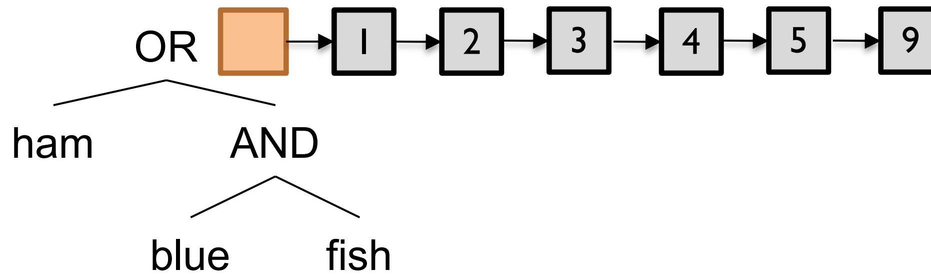
What's RPN?

Term-at-a-Time

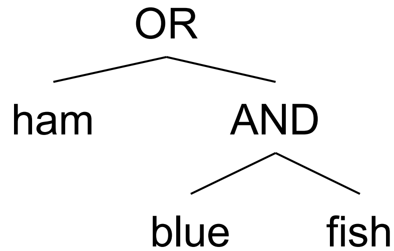


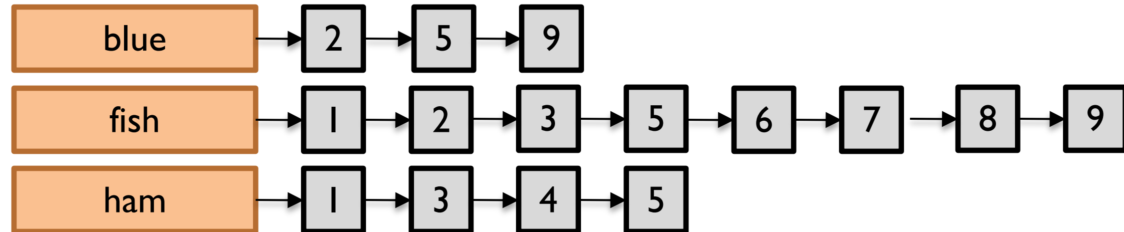


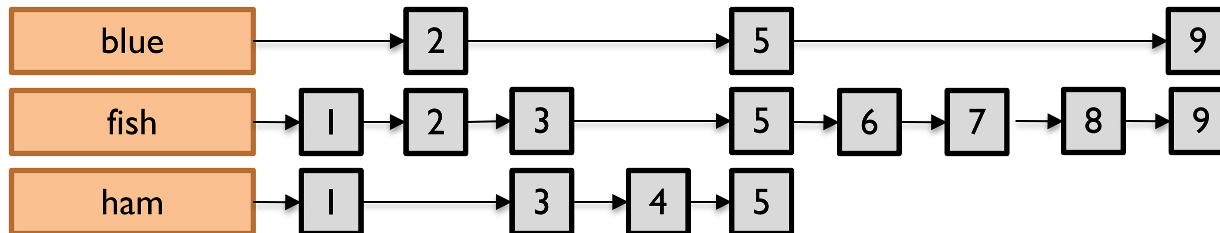
Efficiency analysis?



Document-at-a-Time







Tradeoffs?

Efficiency analysis?

Boolean Retrieval

Users express queries as a boolean expression

AND, OR, NOT

Can be arbitrarily nested

Retrieval is based on the notion of sets

Any query divides the collection into two sets: retrieved, not-retrieved

Pure boolean systems do not define an ordering of the results

What's the issue?

Lexical Retrieval: Challenges

Challenge #1: What should the term weights be?

Challenge #2: How do we perform top-*k* retrieval efficiently?

~~Let's start with boolean retrieval….~~

Ranked Retrieval

Order documents by how likely they are to be relevant

Estimate relevance(q, d_i)

Sort documents by relevance

How do we estimate relevance?

A Vector Space Model for Automatic Indexing

1. Document Space Configurations

Consider a document space consisting of documents D_i, each identified by one or more index terms T_j;

index terms are present. In that case, each document D_i is represented by a t-dimensional vector

$$D_i = (d_{i1}, d_{i2}, \ldots, d_{it}),$$

d_{ij} representing the weight of the jth term.

Given the index vectors for two documents, it is possible to compute a similarity coefficient between them, $s(D_i, D_j)$, which reflects the degree of similarity in the corresponding terms and term weights. Such a similarity measure might be the inner product of the two vectors, or alternatively an inverse function of the angle between the corresponding vector pairs; when the

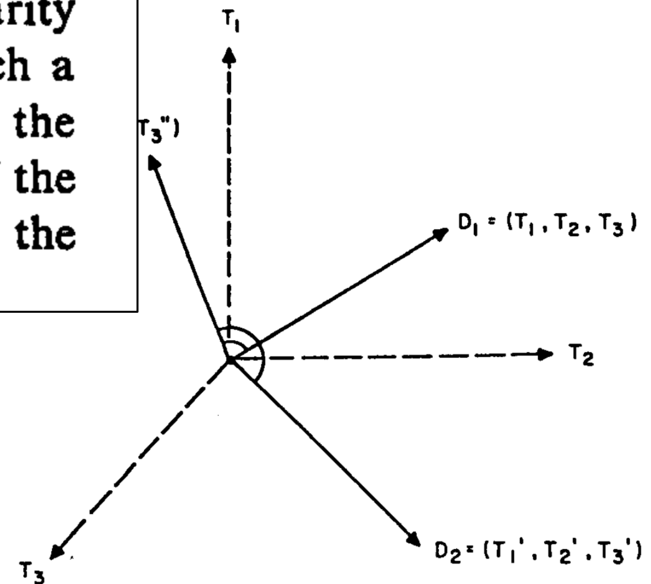
of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This study was supported in part by the National Science Foundation under grant GN 43505.

Authors' addresses: G. Salton and A. Wong, Department of Computer Science, Cornell University, Ithaca, NY 14850; C. S. Yang, Department of Computer Science, The University of Iowa, Iowa City, IA, 52240.

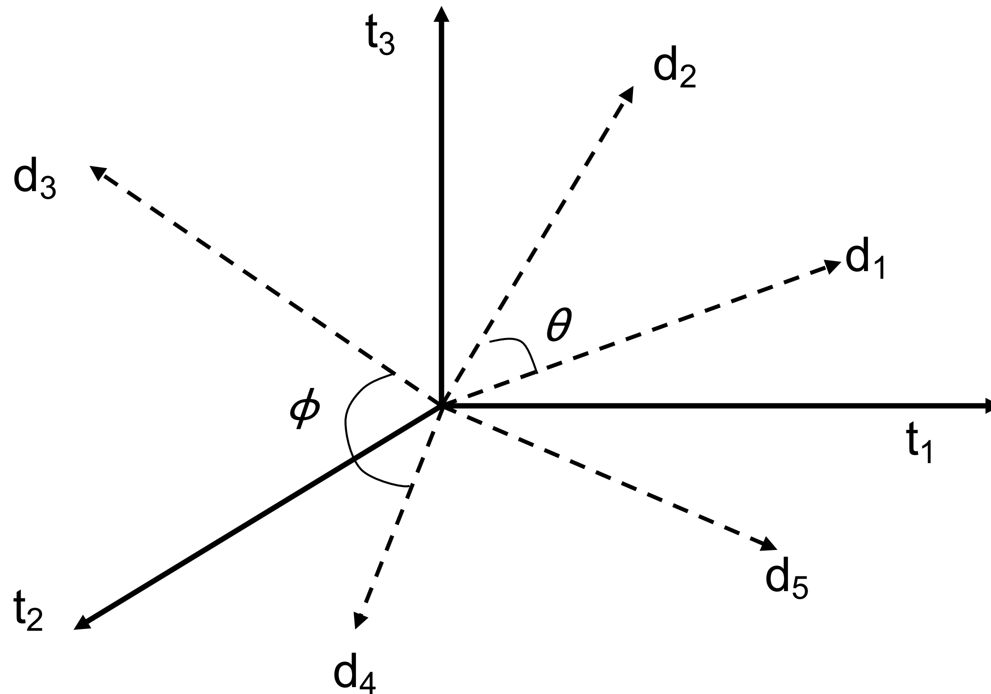
[1] Although we speak of documents and index terms, the present development applies to any set of entities identified by weighted property vectors.

[2] Retrieval performance is often measured by parameters such as *recall* and *precision*, reflecting the ratio of relevant items actually retrieved and of retrieved items actually relevant. The question concerning optimum space configurations may then be more conventionally expressed in terms of the relationship between document indexing, on the one hand, and retrieval performance, on the other.



Salton et al. (1975) A Vector Space Model for Automatic Indexing. *Communications of the ACM*, 18(11):613-620.

Vector Space Model



Assumption: Documents that are "close together" in vector space "talk about" the same things

Therefore, retrieve documents based on how close the document is to the query (i.e., similarity ~ "closeness")

Similarity Metric

Use "angle" between the vectors:

$$d_j = [w_{j,1}, w_{j,2}, w_{j,3}, \ldots w_{j,n}]$$
$$d_k = [w_{k,1}, w_{k,2}, w_{k,3}, \ldots w_{k,n}]$$

$$\cos\theta = \frac{d_j \cdot d_k}{|d_j||d_k|}$$

$$\text{sim}(d_j, d_k) = \frac{d_j \cdot d_k}{|d_j||d_k|} = \frac{\sum_{i=0}^{n} w_{j,i} w_{k,i}}{\sqrt{\sum_{i=0}^{n} w_{j,i}^2}\sqrt{\sum_{i=0}^{n} w_{k,i}^2}}$$

Or, more generally, inner products:

$$\text{sim}(d_j, d_k) = d_j \cdot d_k = \sum_{i=0}^{n} w_{j,i} w_{k,i}$$

Term Weighting

Term weights consist of two components

Local: how important is the term in this document?

Global: how important is the term in the collection?

Here's the intuition:

Terms that appear often in a document should get high weights

Terms that appear in many documents should get low weights

How do we capture this mathematically?

Term frequency (local)

Inverse document frequency (global)

TF.IDF Term Weighting

$$w_{i,j} = \text{tf}_{i,j} \cdot \log \frac{N}{n_i}$$

$w_{i,j}$ weight assigned to term *i* in document *j*

$\text{tf}_{i,j}$ number of occurrence of term *i* in document *j*

N number of documents in entire collection

n_i number of documents with term *i*

BM25

Similar underlying intuitions as TF.IDF

$$\text{BM25}(q,d) = \sum_{t \in q \cap d} \log \frac{N - \text{df}(t) + 0.5}{\text{df}(t) + 0.5} \cdot \frac{\text{tf}(t,d) \cdot (k_1 + 1)}{\text{tf}(t,d) + k_1 \cdot \left(1 - b + b \cdot \frac{l_d}{L}\right)}$$

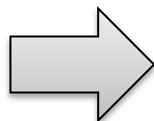
Doc 1
one fish, two fish

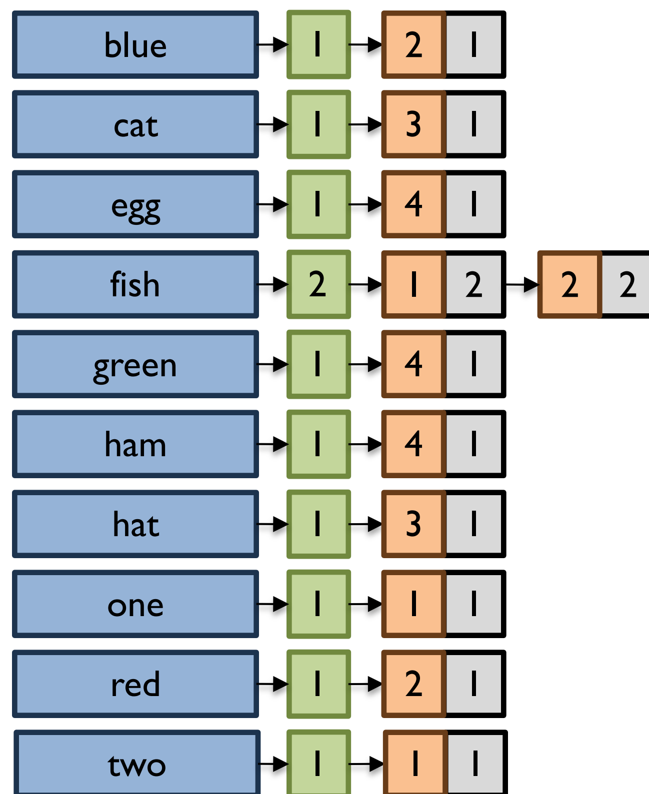
Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

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





What's the final piece of information you need?

Retrieval in a Nutshell

Look up postings lists corresponding to query terms

Traverse postings for each query term

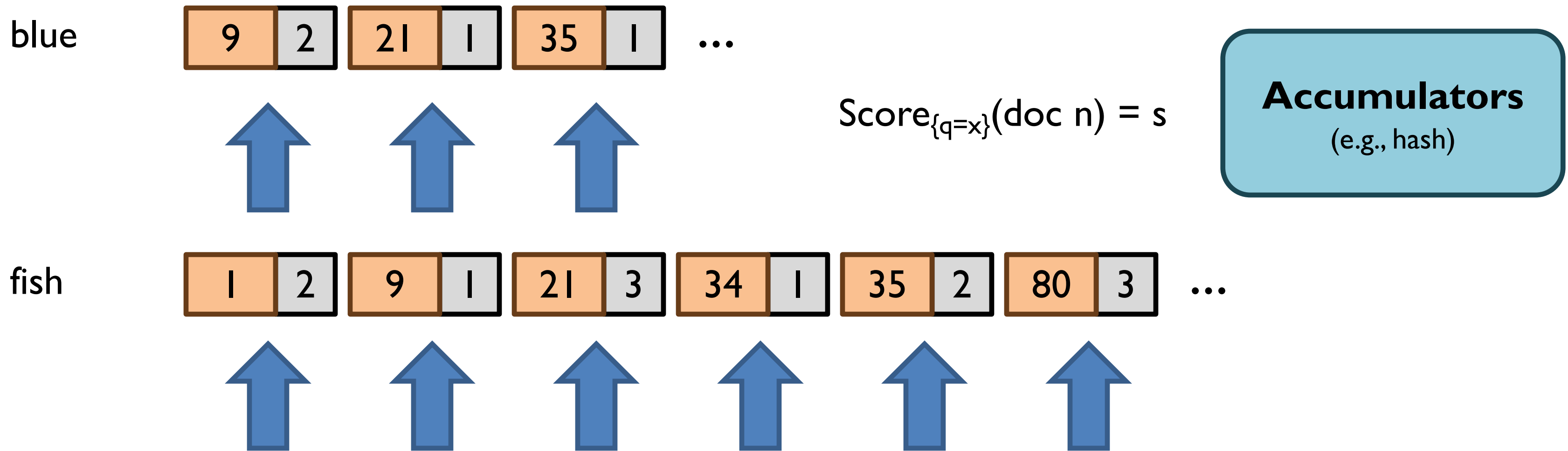
Store partial query-document scores in accumulators

Select top *k* results to return

Retrieval: Term-At-A-Time

Evaluate documents one query term at a time

Usually, starting from most rare term



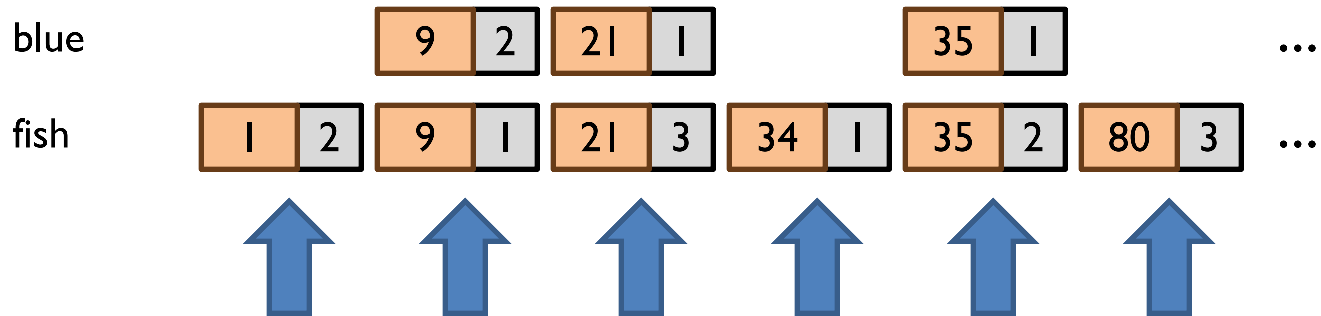
Tradeoffs:

Early termination heuristics (good)

Large memory footprint (bad), but filtering heuristics possible

Retrieval: Document-at-a-Time

Evaluate documents one at a time (score all query terms)



Accumulators
(e.g. min heap)

Document score in top *k*?

- **Yes**: Insert document score, extract-min if heap too large
- **No**: Do nothing

Tradeoffs:

Small memory footprint (good)

Skipping possible to avoid reading all postings (good)

More seeks and irregular data accesses (bad)

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			

Indexing: building this structure

Retrieval: using it to perform top-*k* retrieval

How?

~~(1) Let's learn how to actually do it…~~

(2) Call an external package

users

Frontend

Backend

OLTP
RDBMS

LLM

Retrieval Model

elastic

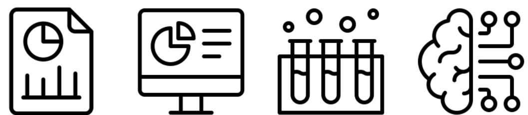
User-generated
content

ELT

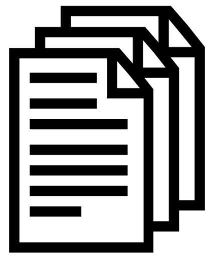
Data Lake

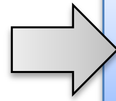
Other acquisition
methods

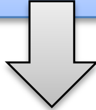
Unified Metadata +
Governance Layer

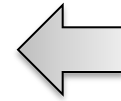


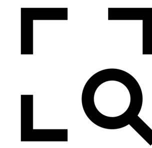
Lakehouse







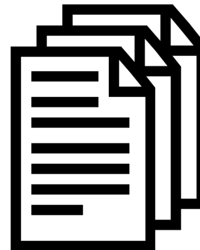




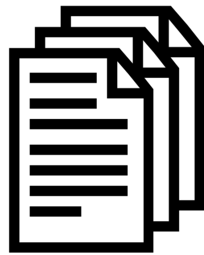
The Manhattan Project and its atomic bomb helped bring an end to World War II. Its legacy of peaceful uses of atomic energy continues to have an impact on history and science.

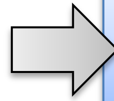
```
{'atom': 4.0140, 'bomb': 4.0704, 'bring': 2.7239, 'continu':
2.4331, 'end': 2.1559, 'energi': 2.5045, 'have': 1.0742, 'help':
1.8157, 'histori': 2.4213, 'ii': 3.0998, 'impact': 3.0304, 'it':
2.0473, 'legaci': 4.1335, 'manhattan': 4.1345, 'peac': 3.5205,
'project': 2.6442, 'scienc': 2.8700, 'us': 0.9967, 'war':
2.6454, 'world': 1.9974}
```

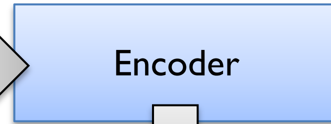
sparse vector

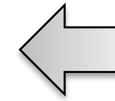


“Documents”







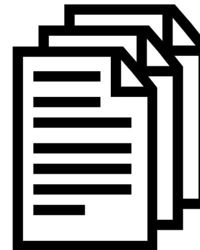


Query

The Manhattan Project and its atomic bomb helped bring an end to World War II. Its legacy of peaceful uses of atomic energy continues to have an impact on history and science.

```
[0.099843978881836, 0.8700575828552246, 0.520509719848633,
0.030491352081299, 0.7239298820495605, 0.134523391723633,
0.4331274032592773, 0.644286632537842, 0.645430564880371,
0.0473427772521973, 0.070496082305908, 0.504533529281616,
0.8157329559326172, 0.133575916290283, 0.9974448680877686,
0.0742542743682861, 0.1559412479400635, 0.421395778656006,
0.014032363891602, 0.996794581413269...]
```

dense vector



Results

冨嶽三十六景
神奈川沖浪裏