UNIVERSITY OF WATERLOO

Data-Intensive Distributed Computing

CS 451/651 (Fall 2018)

Part 3: Analyzing Text (2/2)

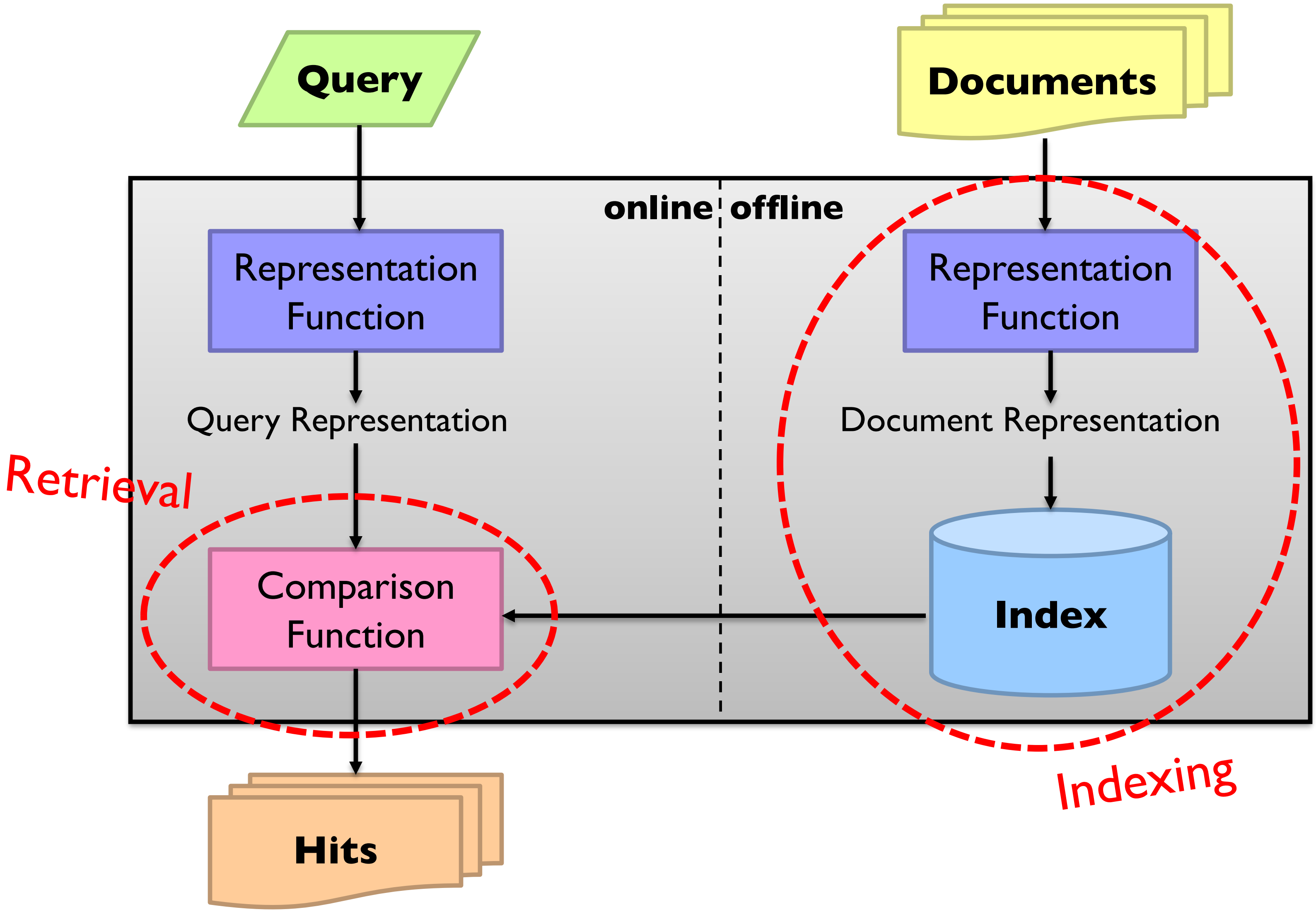
October 2, 2018

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

These slides are available at http://lintool.github.io/bigdata-2018f/

Source: http://www.flickr.com/photos/guvnah/7861418602/

Abstract IR Architecture



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: manipulating this structure

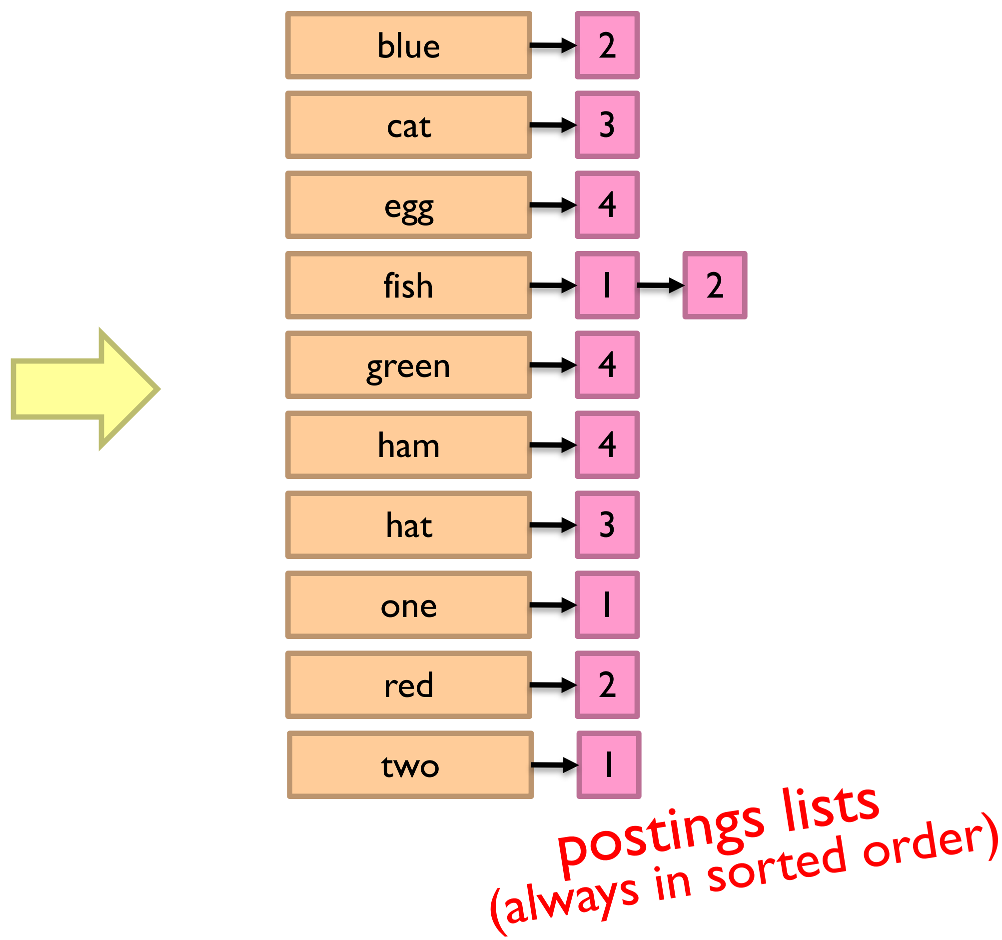
Doc 1
one fish, two fish

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			



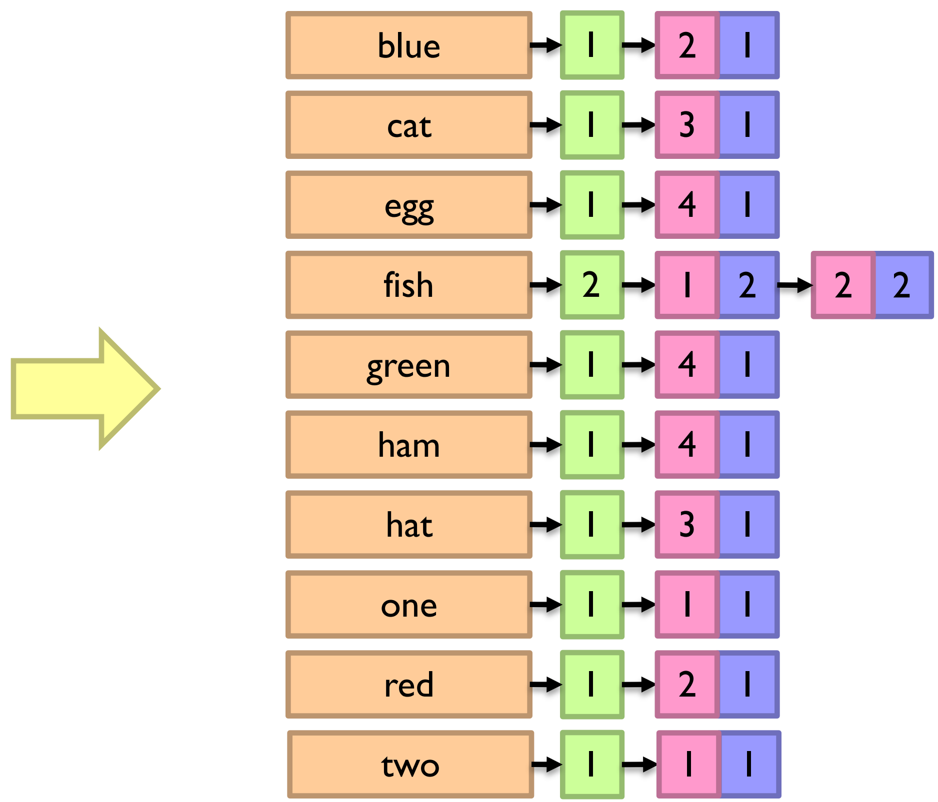
Doc 1 one fish, two fish

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

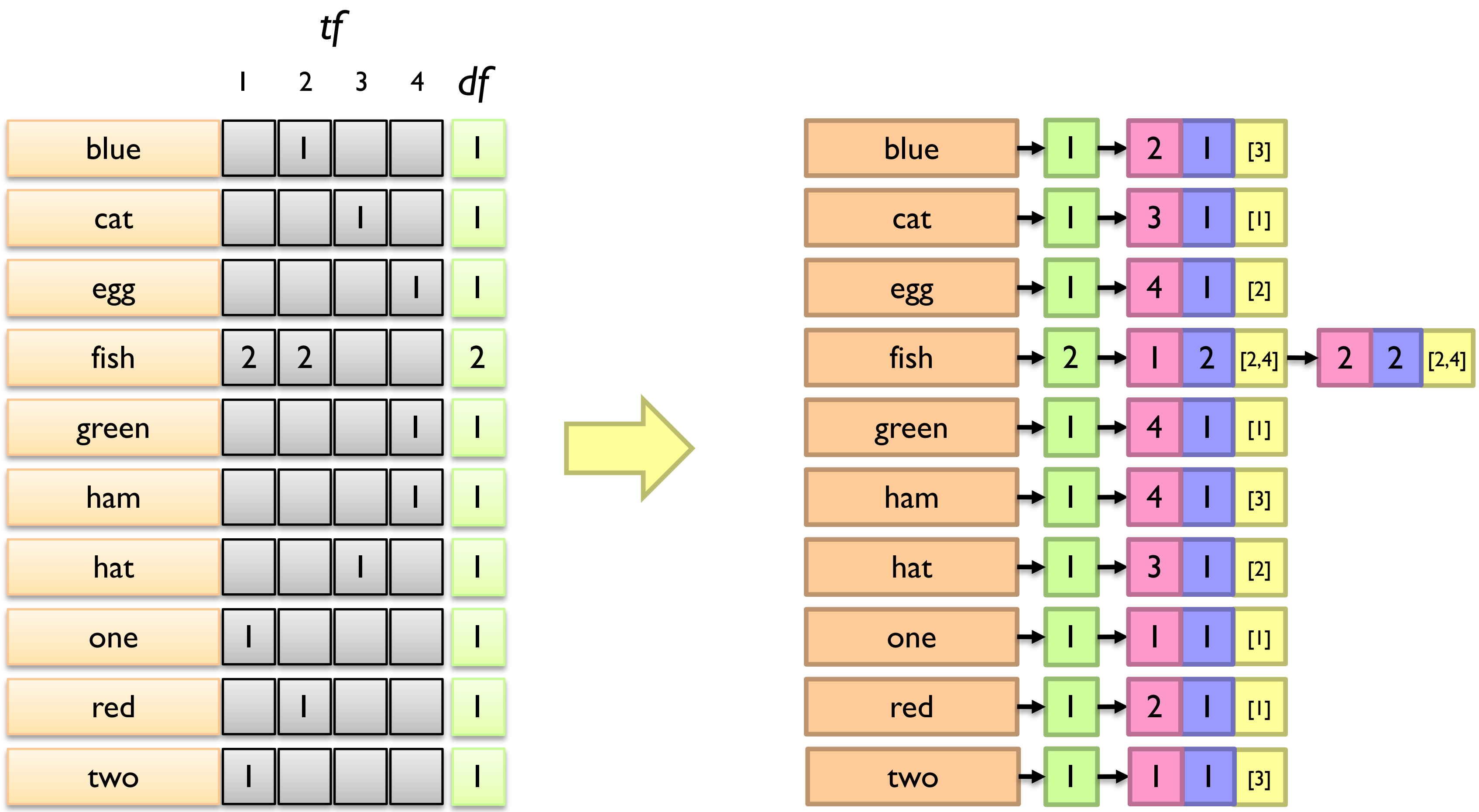


Doc 1
one fish, two fish

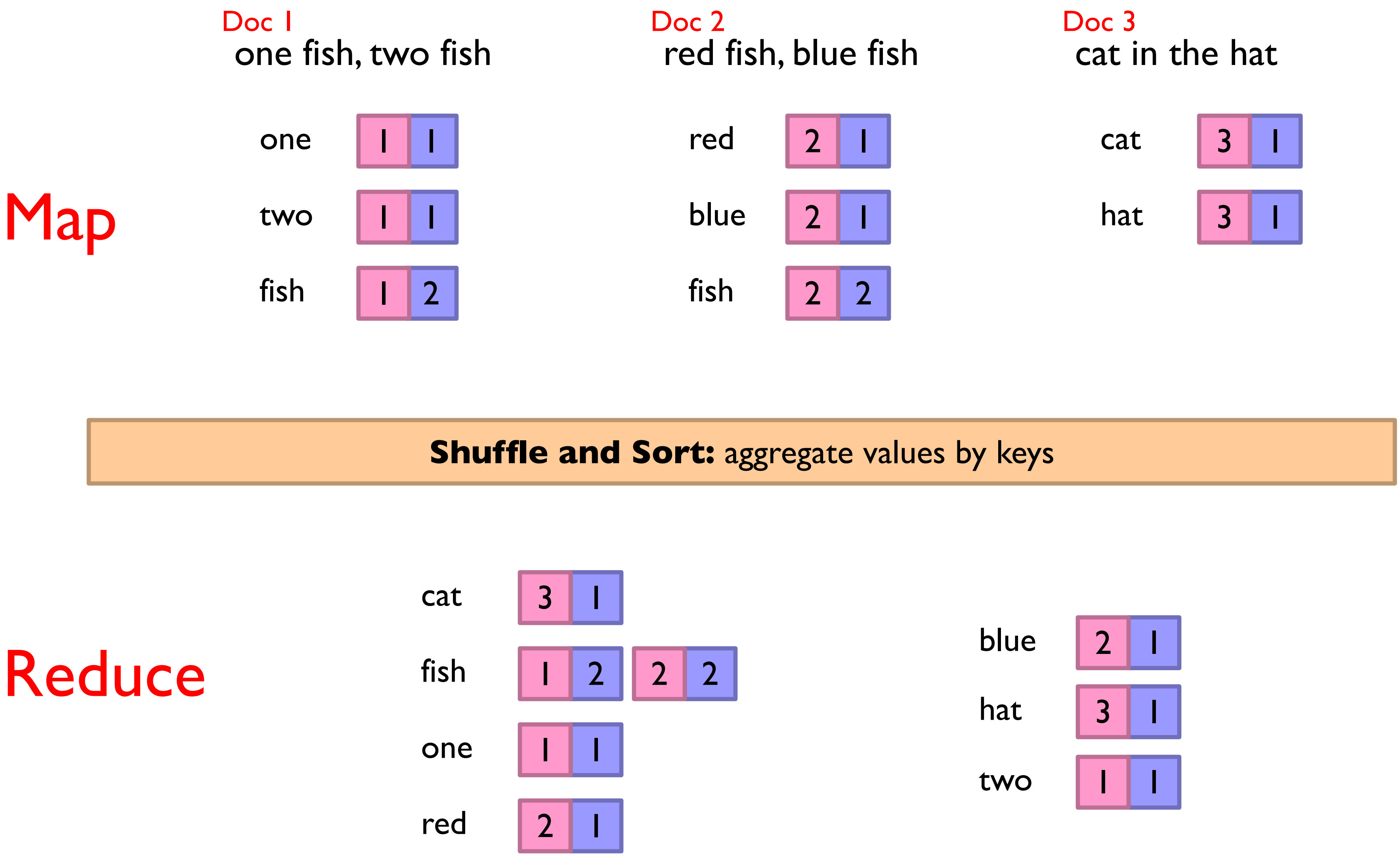
Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham



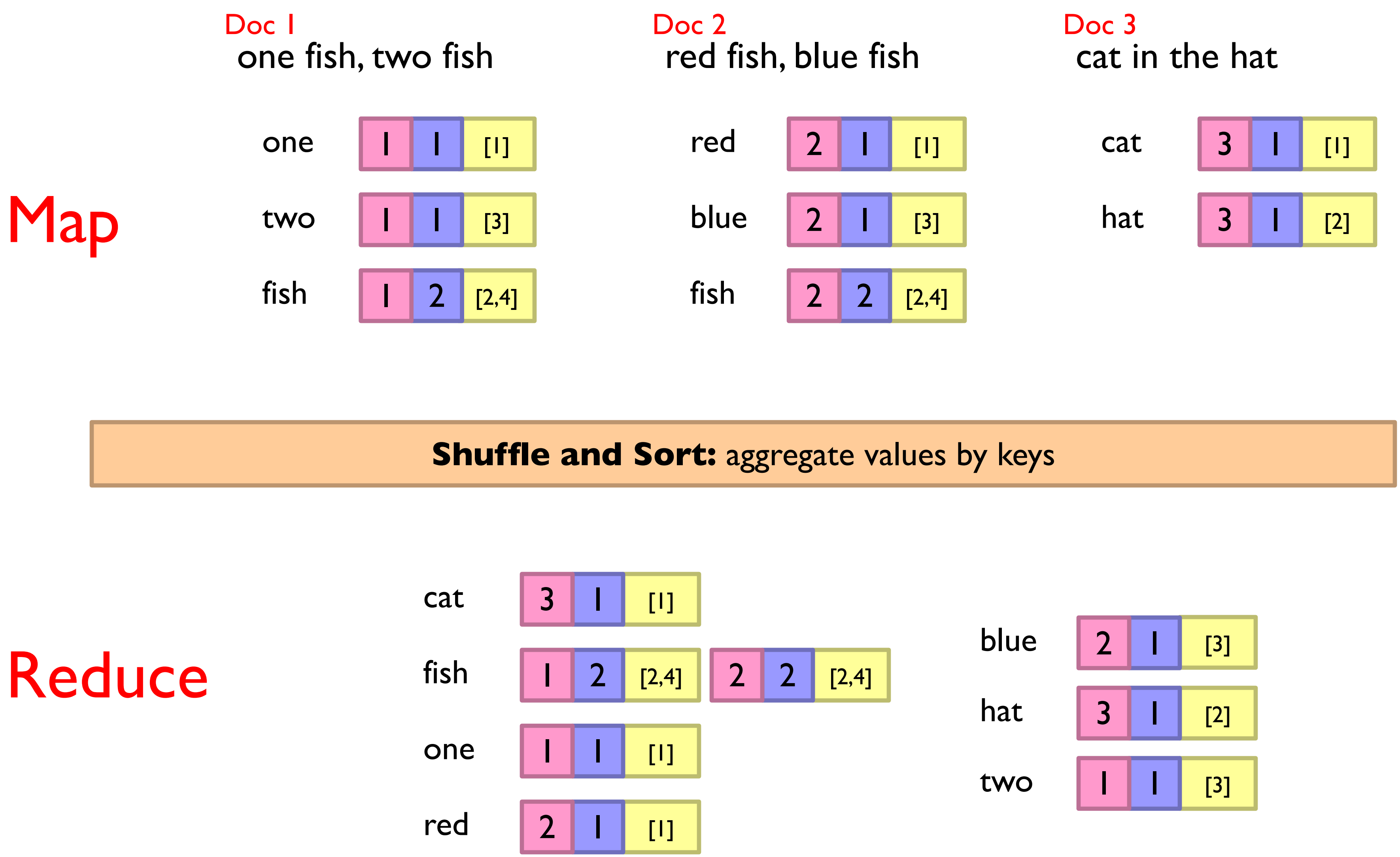
Inverted Indexing with MapReduce



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)
  }
}
```

Positional Indexes

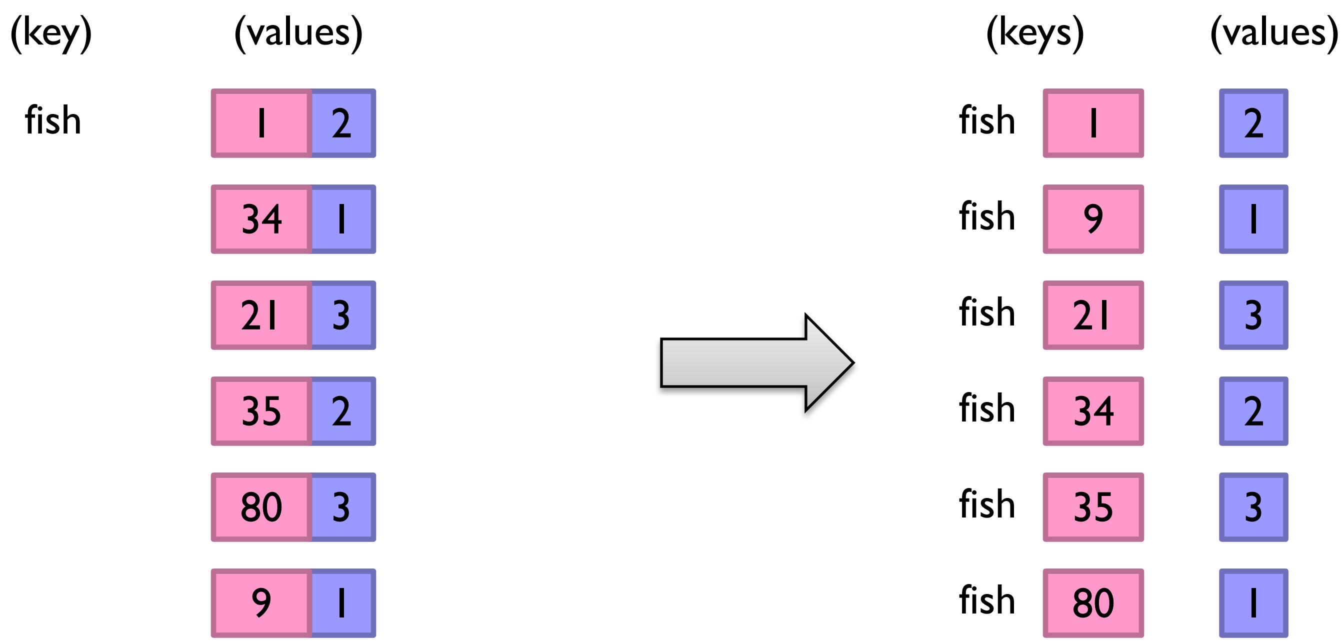


Inverted Indexing: Pseudo-Code

```
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 problem?

Another Try…



How is this different?

Let the framework do the sorting!

Where have we seen this before?

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:

fish	1	2	9	1	21	3	34	1	35	2	80	3	…

In Practice:

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

fish	1	2	8	1	12	3	13	1	1	2	45	3	…

= delta encoding, delta compression, gap compression

Overview of Integer Compression

Byte-aligned technique

VByte

Bit-aligned

Unary codes

γ/δ codes

Golomb codes (local Bernoulli model)

Word-aligned

Simple family

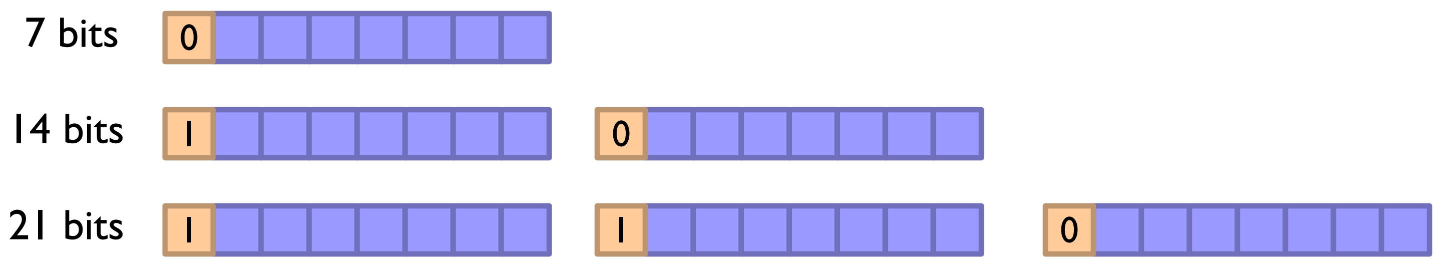
Bit packing family (PForDelta, etc.)

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

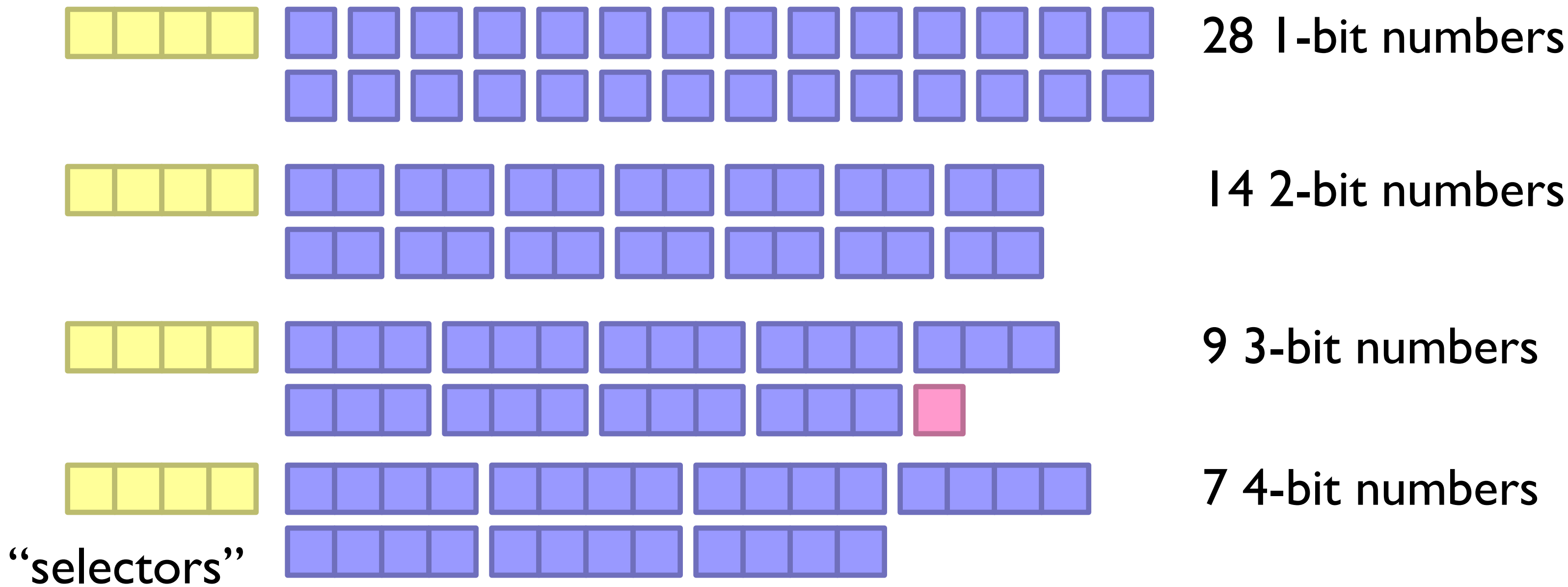


Works okay, easy to implement…

Beware of branch mispredicts!

Simple-9

How many different ways can we divide up 28 bits?



(9 total ways)

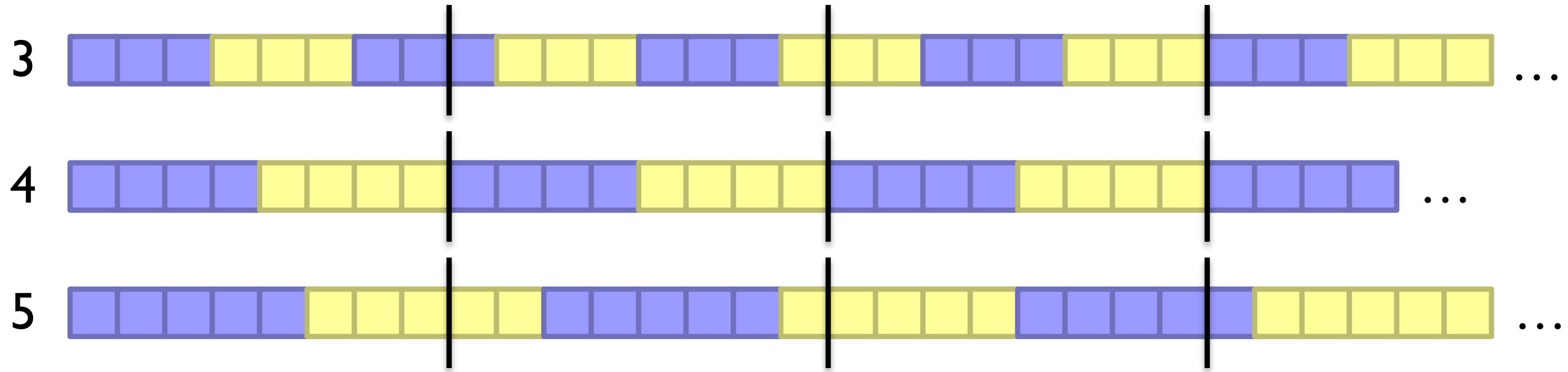
Efficient decompression with hard-coded decoders

Simple Family – general idea applies to 64-bit words, etc.

Beware of branch mispredicts?

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

Beware of branch mispredicts?

Golomb Codes

$x \geq 1$, parameter b:

- $q + 1$ in unary, where $q = \lfloor (x - 1) / b \rfloor$
- r in binary, where $r = x - qb - 1$, in $\lfloor \log b \rfloor$ or $\lceil \log b \rceil$ bits

Example:

- $b = 3$, $r = 0, 1, 2$ (0, 10, 11)
- $b = 6$, $r = 0, 1, 2, 3, 4, 5$ (00, 01, 100, 101, 110, 111)
- $x = 9$, $b = 3$: $q = 2$, $r = 2$, code = 110:11
- $x = 9$, $b = 6$: $q = 1$, $r = 2$, code = 10:100

Punch line: optimal $b \sim 0.69 (N/df)$

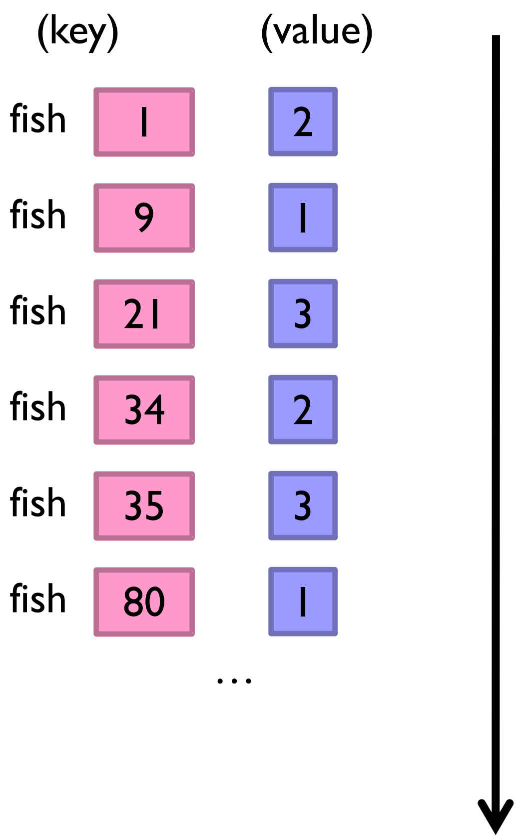
Different b for every term!

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)
 }
}
```

Ah, now we know why this is different!

Chicken and Egg?



Write postings *compressed*

But wait! How do we set the Golomb parameter *b*?

Recall: optimal $b \sim 0.69\ (N/df)$

We need the *df* to set *b*…

But we don't know the *df* until we've seen all postings!

Sound familiar?

Getting the *df*

In the mapper:

Emit “special” key-value pairs to keep track of *df*

In the reducer:

Make sure “special” key-value pairs come first: process them to determine *df*

Remember: proper partitioning!

Getting the *df*: Modified Mapper

Doc 1

one fish, two fish

Input document…

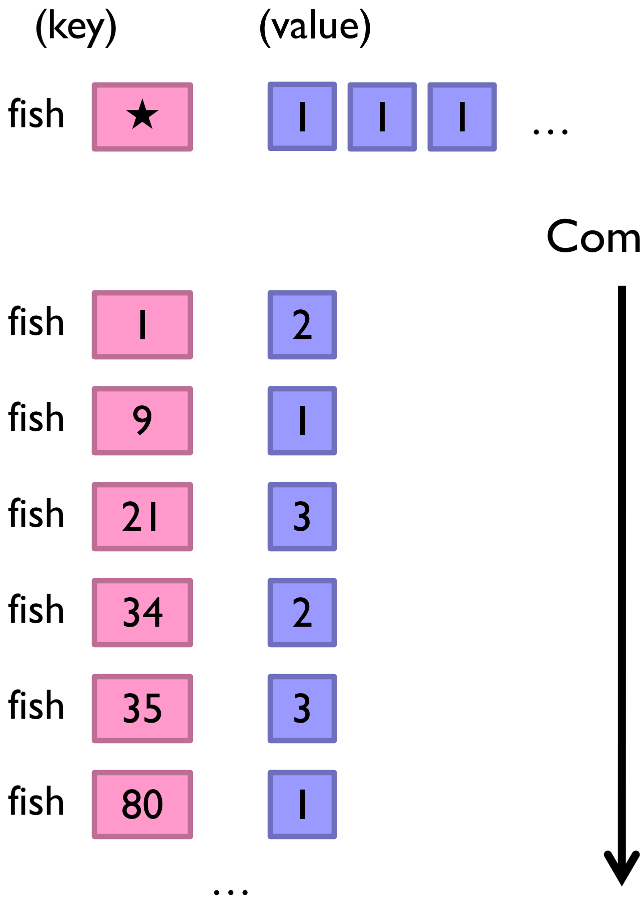
	(key)	(value)
fish	1	2
one	1	1
two	1	1

Emit normal key-value pairs…

	(key)	(value)
fish	★	1
one	★	1
two	★	1

Emit “special” key-value pairs to keep track of *df*…

Getting the *df*: Modified Reducer



First, compute the *df* by summing contributions from all “special” key-value pair…

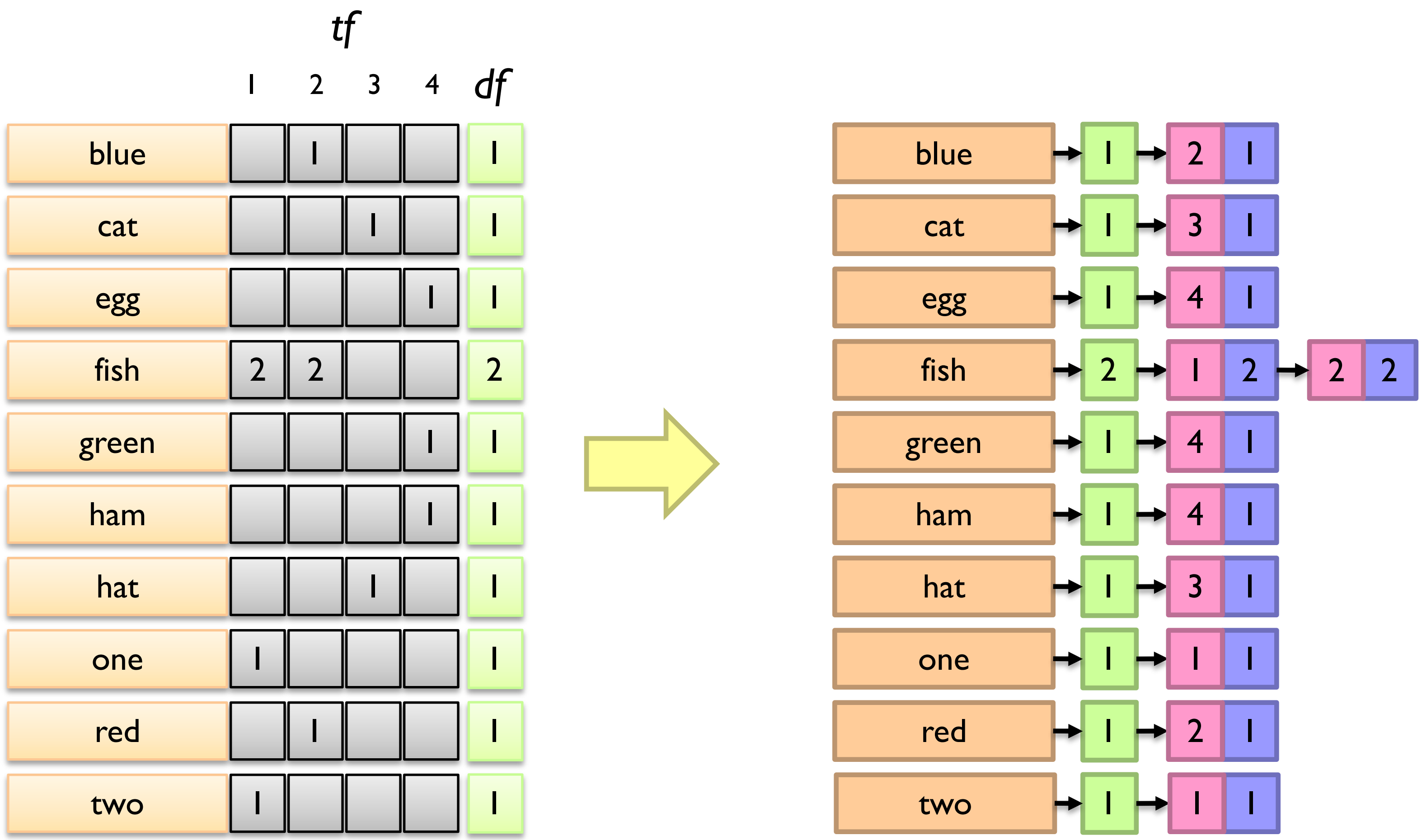
Compute *b* from *df*

Important: properly define sort order to make sure “special” key-value pairs come first!

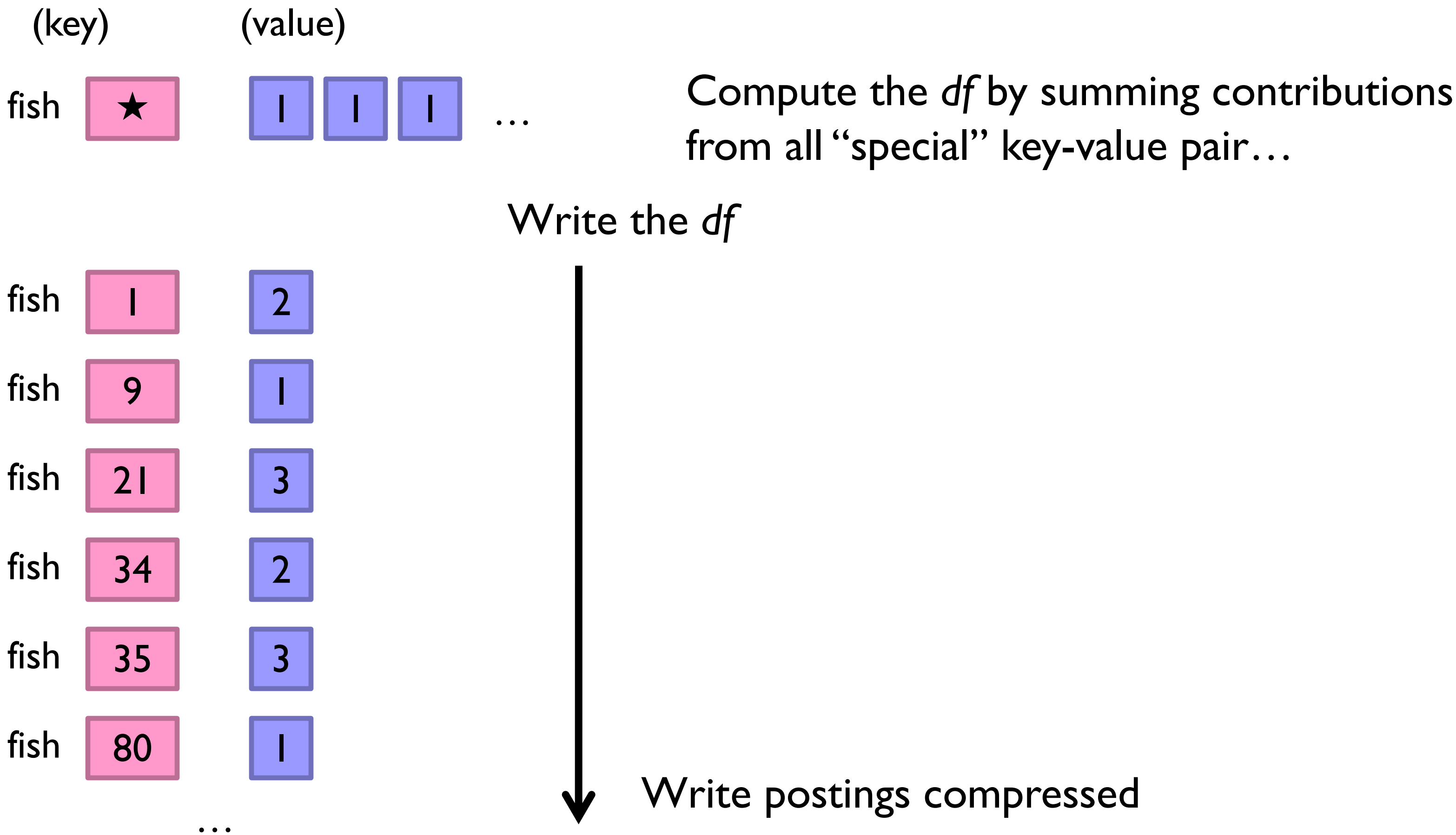
Write postings compressed

Where have we seen this before?

But I don't care about Golomb Codes!



Basic Inverted Indexer: Reducer



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)
 }
}
```

What's the assumption?
Is it okay?

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)

What exactly is this operation?
What have we created?

Then we can rewrite our indexing algorithm!

flatMap: emit singleton postings
reduceByKey: $\oplus$

What's the issue?

$$\text{Postings}_1 \oplus \text{Postings}_2 = \text{Postings}_M$$

Solution: apply compression as needed!

Inverted Indexing: LP (~Stripes)

Slightly less elegant implementation… but uses same idea

```
class Mapper {
  val m = new Map()

  def map(docid: Long, doc: String) = {
    val counts = new Map()
    for (term <- tokenize(doc)) {
      counts(term) += 1
    }
    for ((term, tf) <- counts) {
      m(term).append((docid, tf))
    }
    if memoryFull()
      flush()
  }

  def cleanup() = {
    flush()
  }

  def flush() = {
    for (term <- m.keys) {
      emit(term, new PostingsList(m(term)))
    }
    m.clear()
  }
}
```

What's happening here?

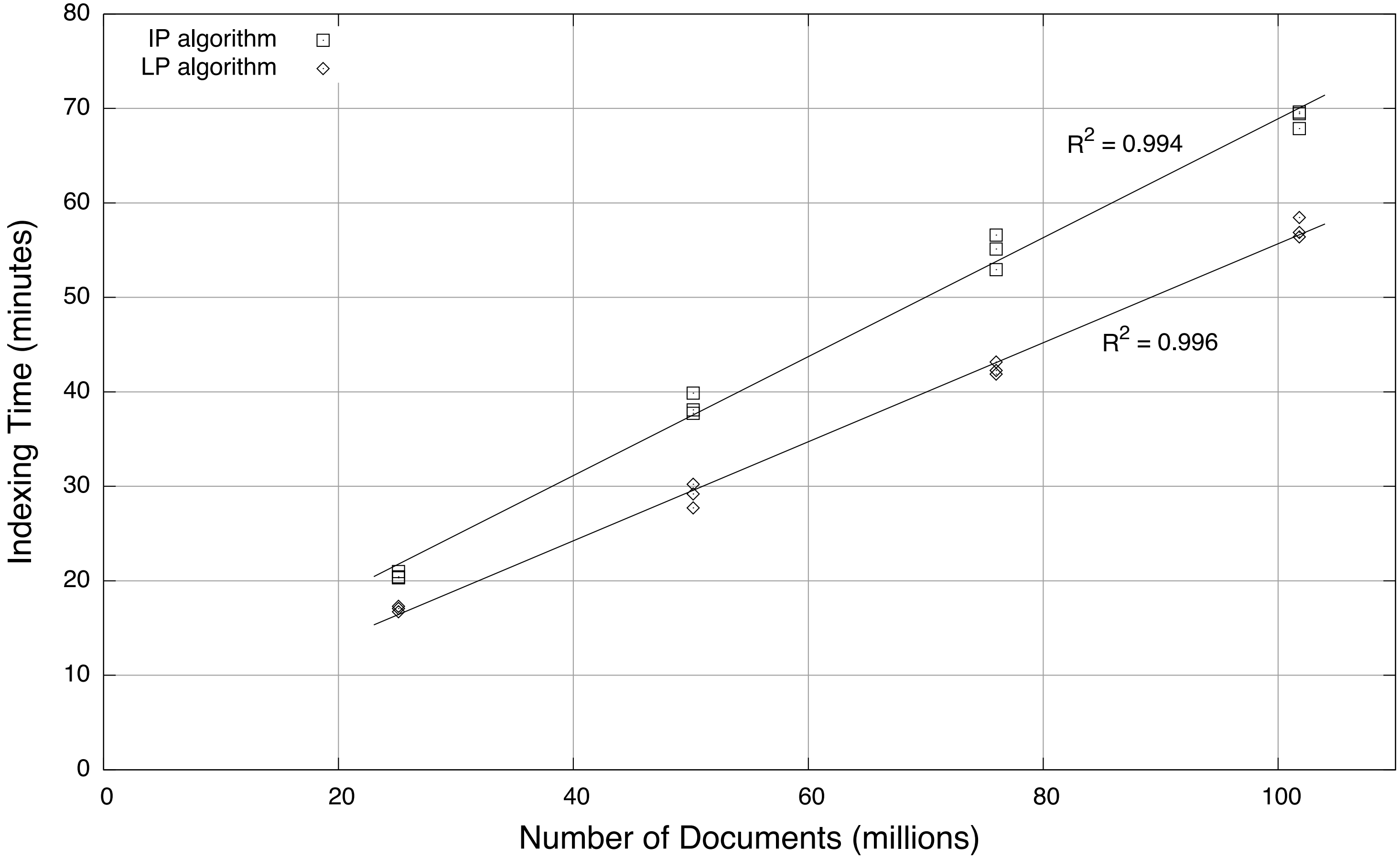
Inverted Indexing: LP (~Stripes)

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

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

    emit(term, f)
  }
}
```

What's happening here?

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

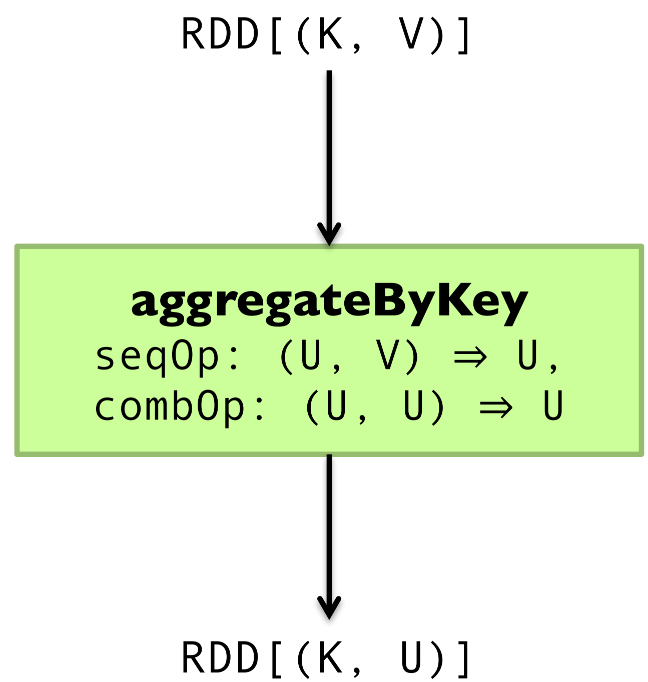
Another Look at LP

```
class Mapper {
  val m = new Map()

  def map(docid: Long, doc: String) = {
    val counts = new Map()
    for (term <- tokenize(doc)) {
      counts(term) += 1
    }
    for ((term, tf) <- counts) {
      m(term).append((docid, tf))
    }
    if memoryFull()
      flush()
  }

  def cleanup() = {
    flush()
  }

  def flush() = {
    for (term <- m.keys) {
      emit(term, new PostingsList(m(term)))
    }
    m.clear()
  }
}

class Reducer {
  def reduce(term: String, lists: Iterable[PostingsList]) = {
    val f = new PostingsList()
    for (list <- lists) {
      f = f + list
    }
    emit(term, f)
  }
}
```

flatMap: emit singleton postings

reduceByKey: $\oplus$

Remind you of anything in Spark?



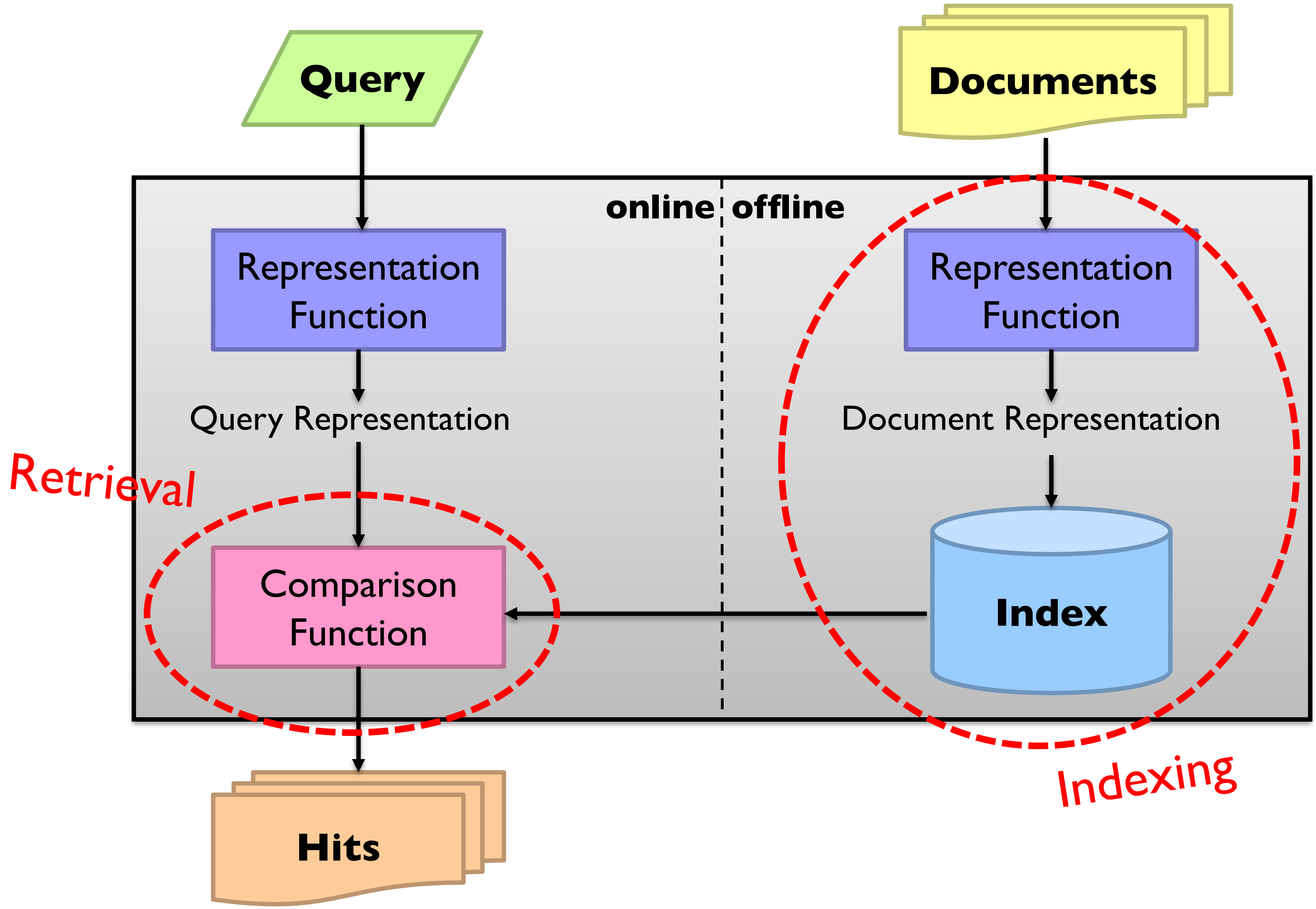
Algorithm design in a nutshell…

Exploit associativity and commutativity via commutative monoids (if you can)

Exploit framework-based sorting to sequence computations (if you can't)

Source: Wikipedia (Walnut)

Abstract IR Architecture



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

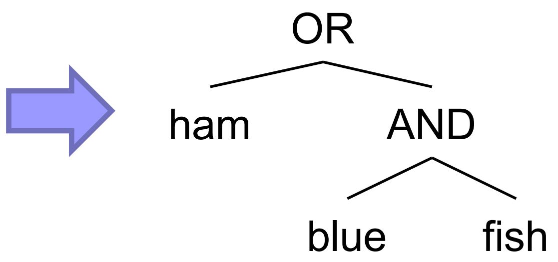
Pure Boolean systems do not define an ordering of the results

Boolean Retrieval

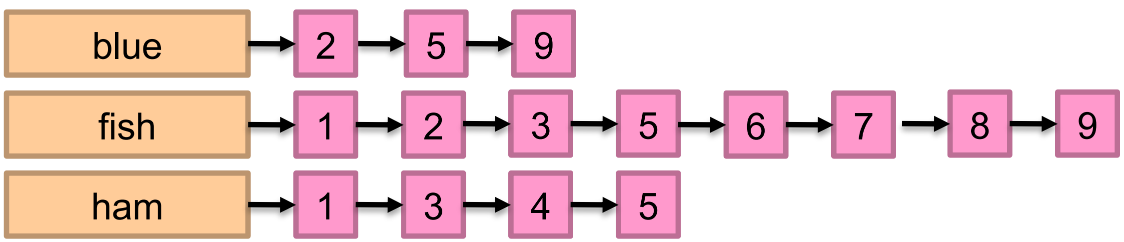
To execute a Boolean query:

Build query syntax tree

(blue AND fish) OR ham

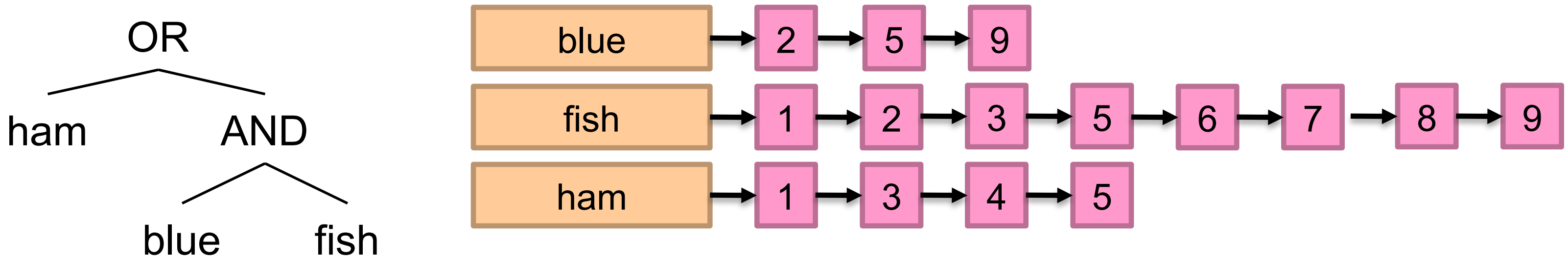


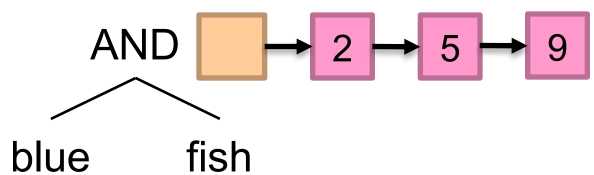
For each clause, look up postings



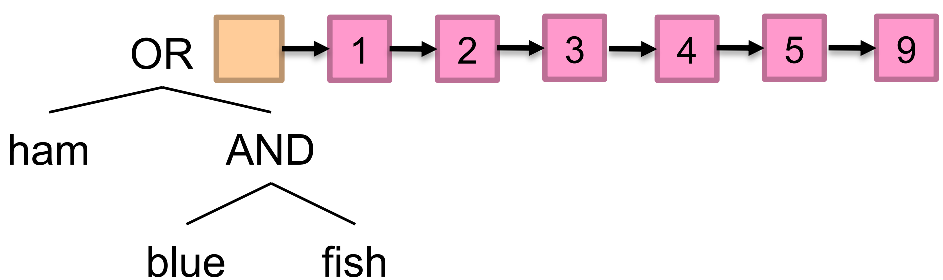
Traverse postings and apply Boolean operator

Term-at-a-Time



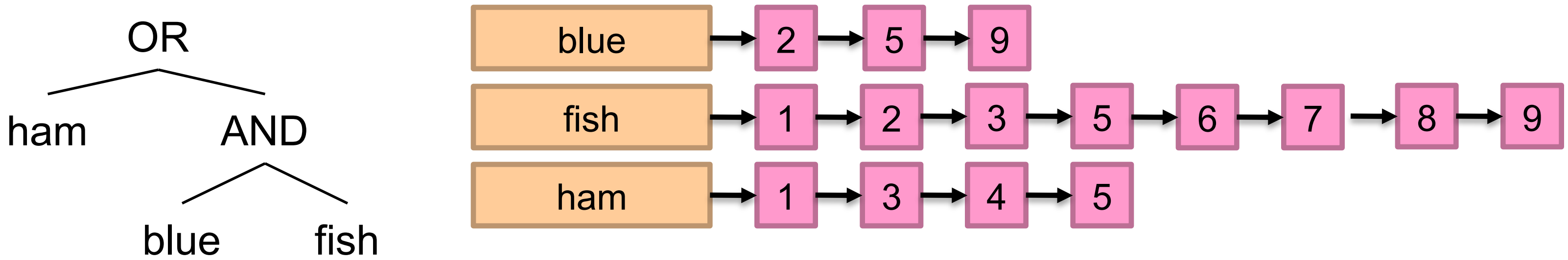


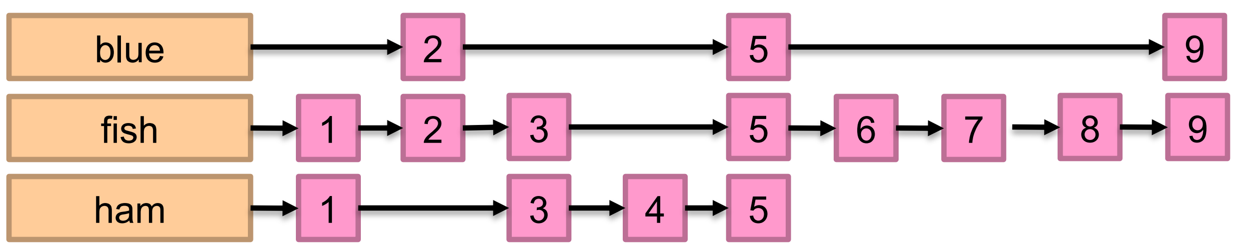
Efficiency analysis?



What's RPN?

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?

Ranked Retrieval

Order documents by how likely they are to be relevant

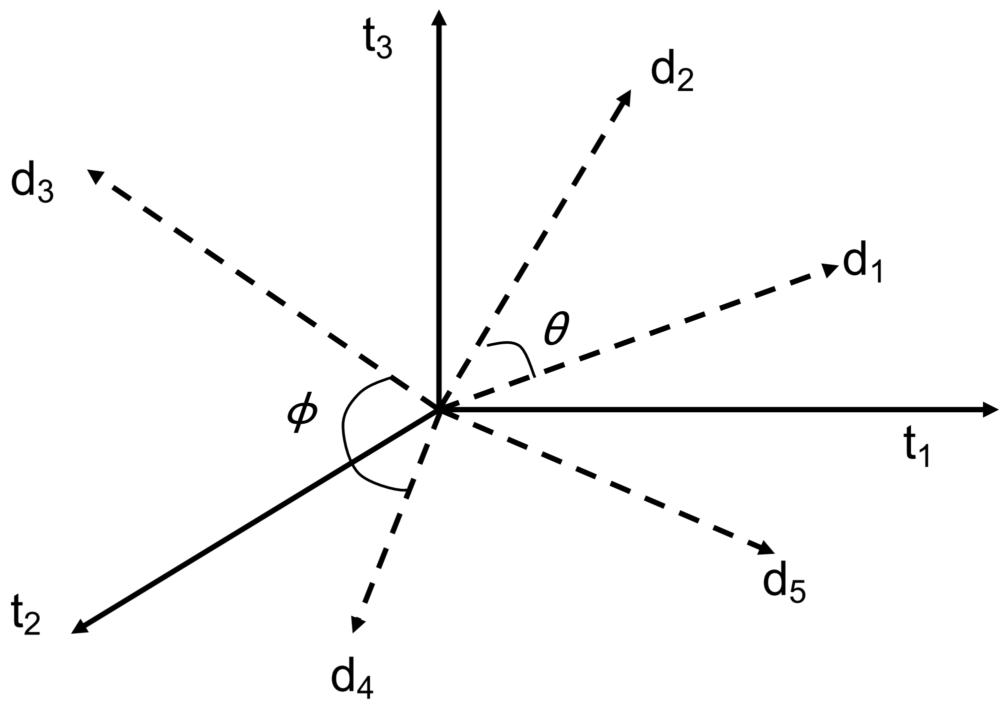
Estimate relevance(q, d_i)

Sort documents by relevance

How do we estimate relevance?

Take “similarity” as a proxy for relevance

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:

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

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

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

Or, more generally, inner products:

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

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

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

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

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

N number of documents in entire collection

n_i number of documents with term *i*

Retrieval in a Nutshell

Look up postings lists corresponding to query terms

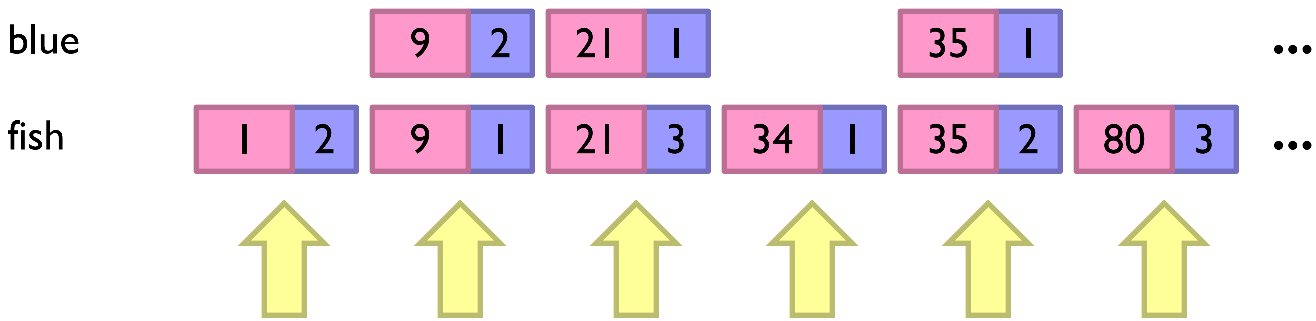
Traverse postings for each query term

Store partial query-document scores in accumulators

Select top *k* results to return

Retrieval: 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

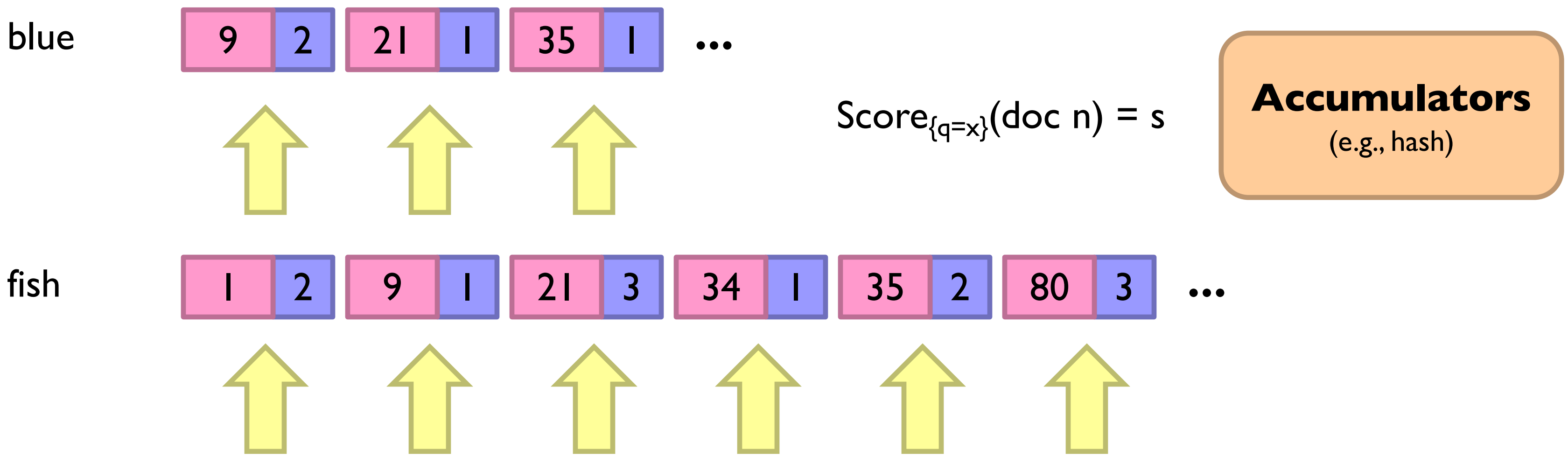
Tradeoffs:

Small memory footprint (good)
Skipping possible to avoid reading all postings (good)
More seeks and irregular data accesses (bad)

Retrieval: Term-At-A-Time

Evaluate documents one query term at a time

Usually, starting from most rare term (often with *tf*-sorted postings)



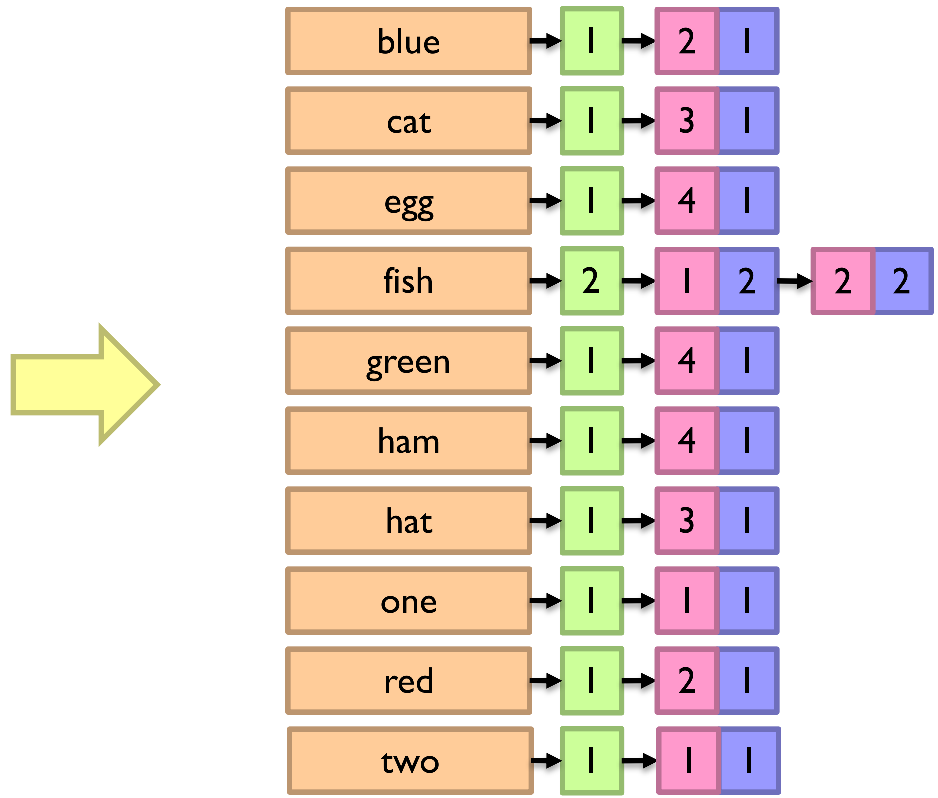
Tradeoffs:

Early termination heuristics (good)

Large memory footprint (bad), but filtering heuristics possible

Why store *df* as part of postings?

	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



Assume everything fits in memory on a single machine…

Okay, let's relax this assumption now

Important Ideas

Partitioning (for scalability)

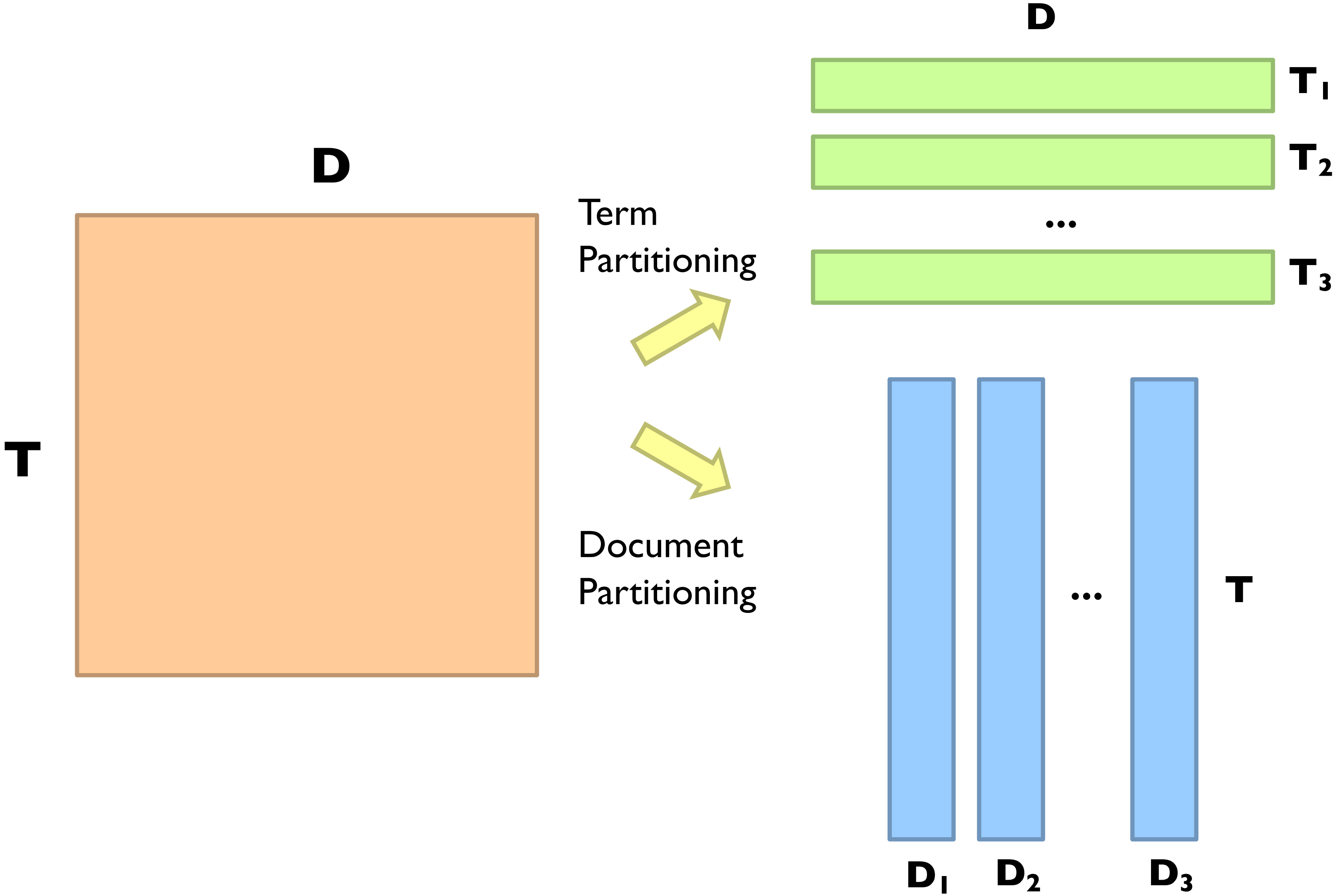
Replication (for redundancy)

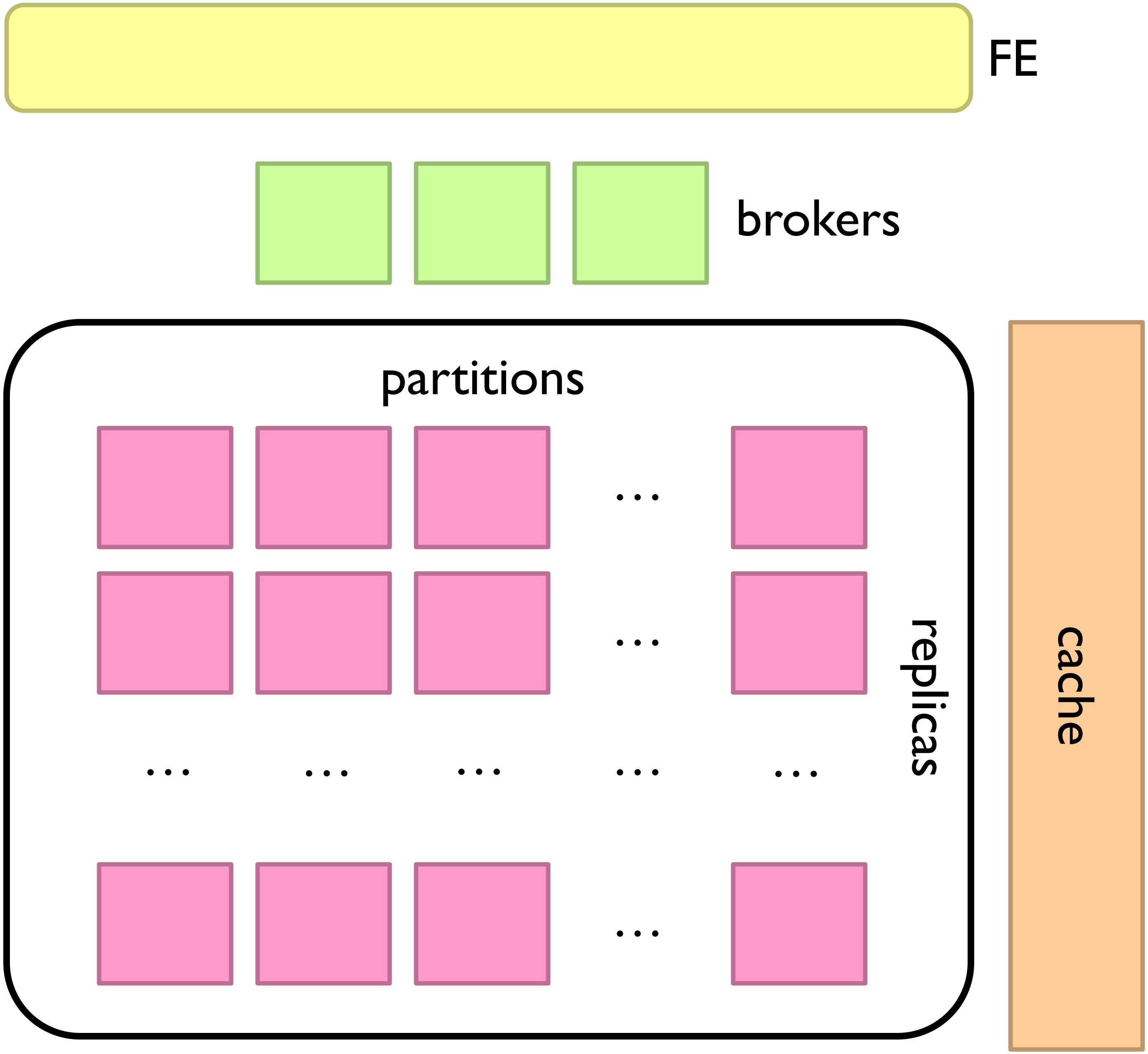
Caching (for speed)

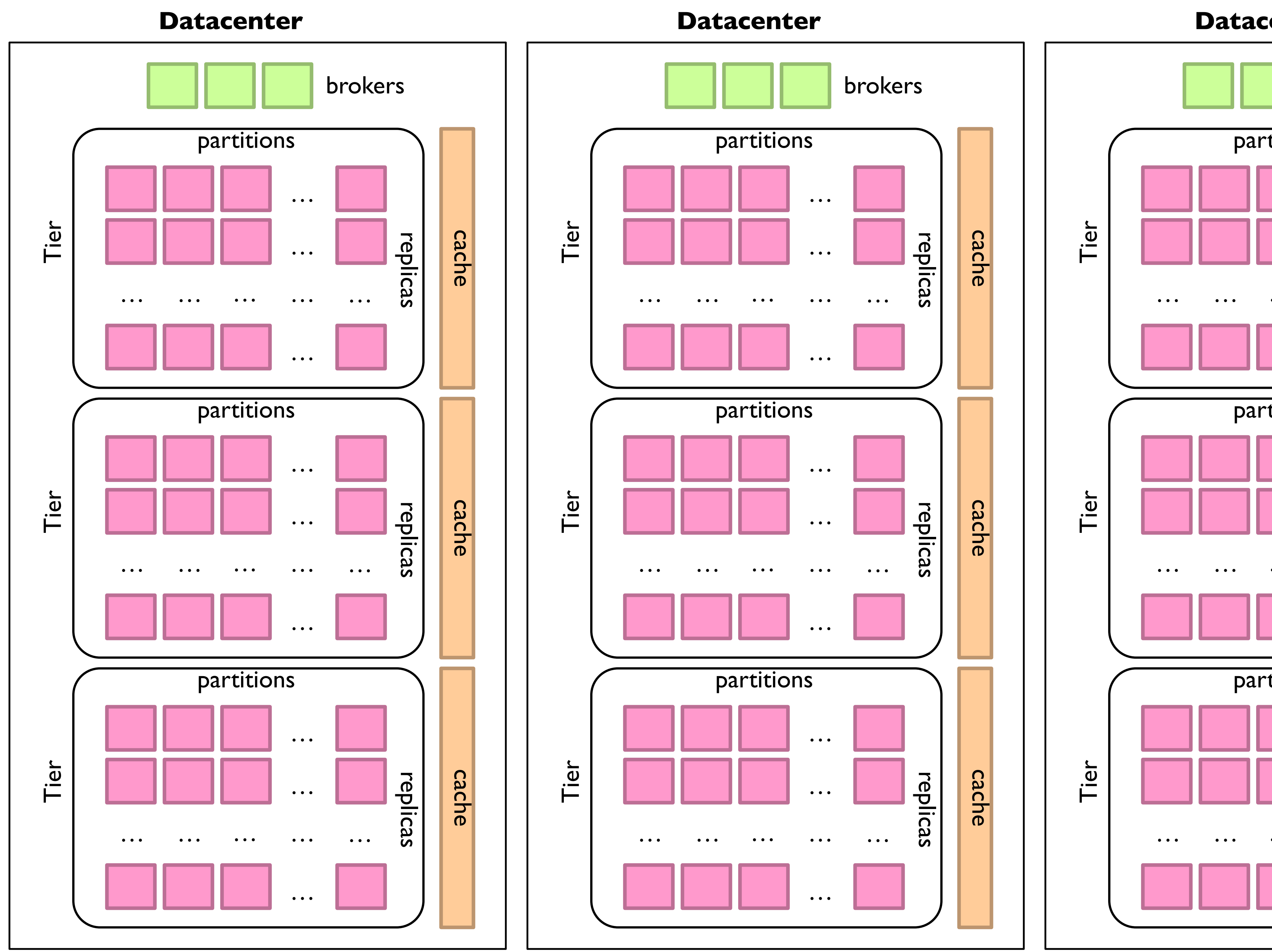
Routing (for load balancing)

The rest is just details!

Term vs. Document Partitioning




FE
brokers
partitions
replicas
cache


Datacenter
brokers
partitions
Tier
replicas
cache
partitions
Tier
replicas
cache
partitions
Tier
replicas
cache
Datacenter
brokers
partitions
Tier
replicas
cache
partitions
Tier
replicas
cache
partitions
Tier
replicas
cache
Tier
Tier
Tier

Important Ideas

Partitioning (for scalability)

Replication (for redundancy)

Caching (for speed)

Routing (for load balancing)


Source: Wikipedia (Japanese rock garden)