Big Data Infrastructure

CS 489/698 Big Data Infrastructure (Winter 2016)

Week 4: Analyzing Text (2/2)

January 28, 2016

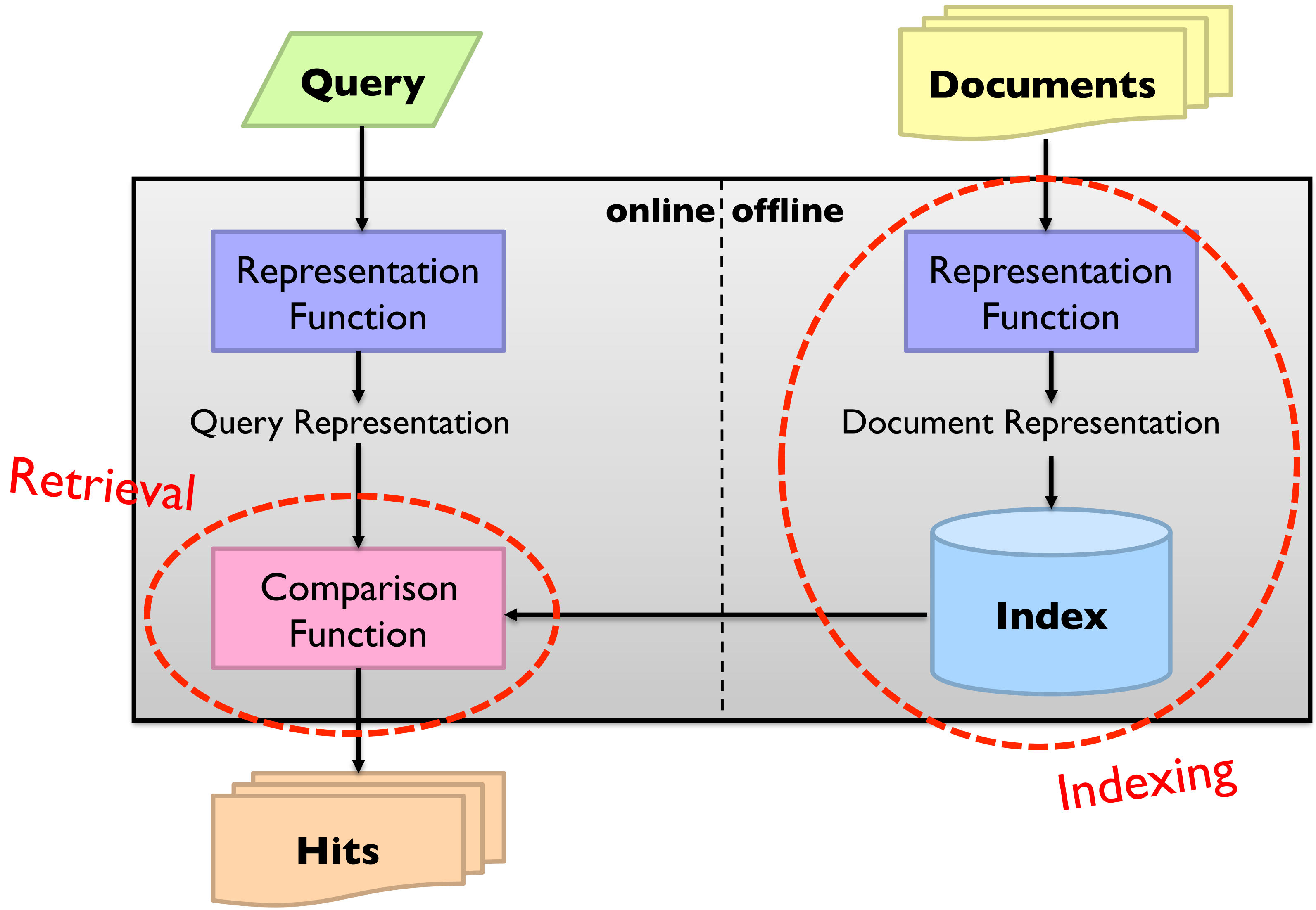
Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at http://lintool.github.io/bigdata-2016w/

Count. Search!

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?

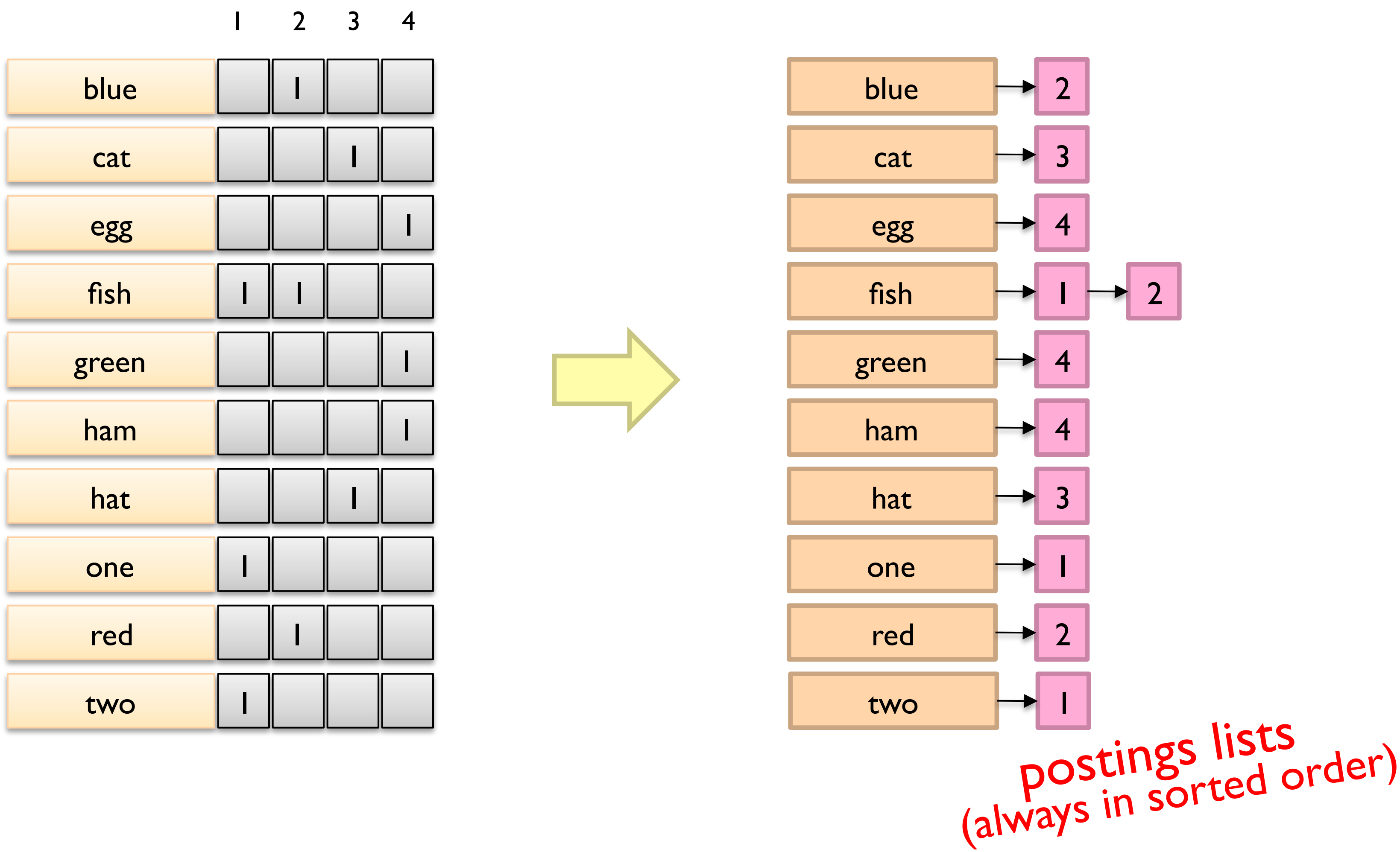
- boolean
- count
- positions

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

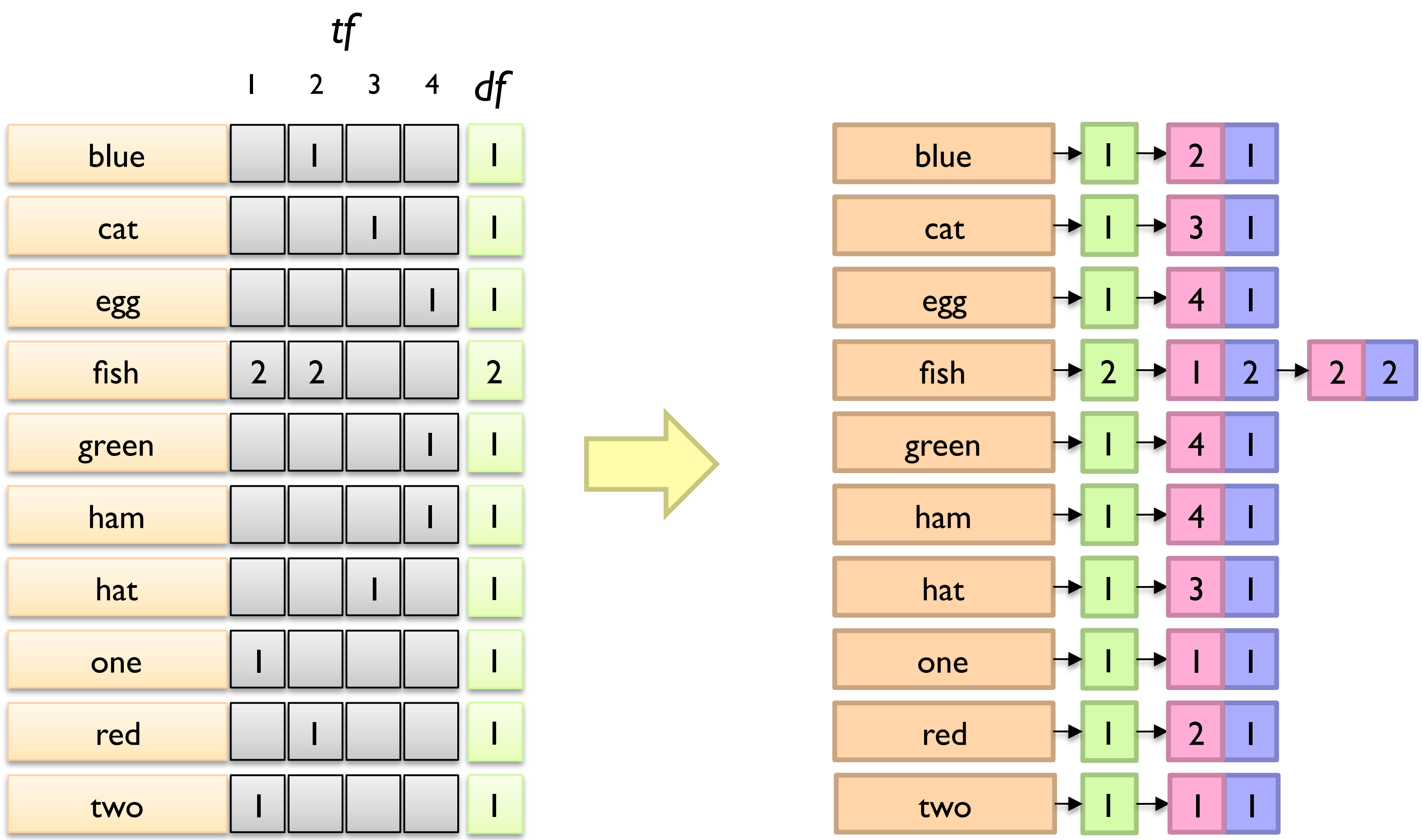


Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham



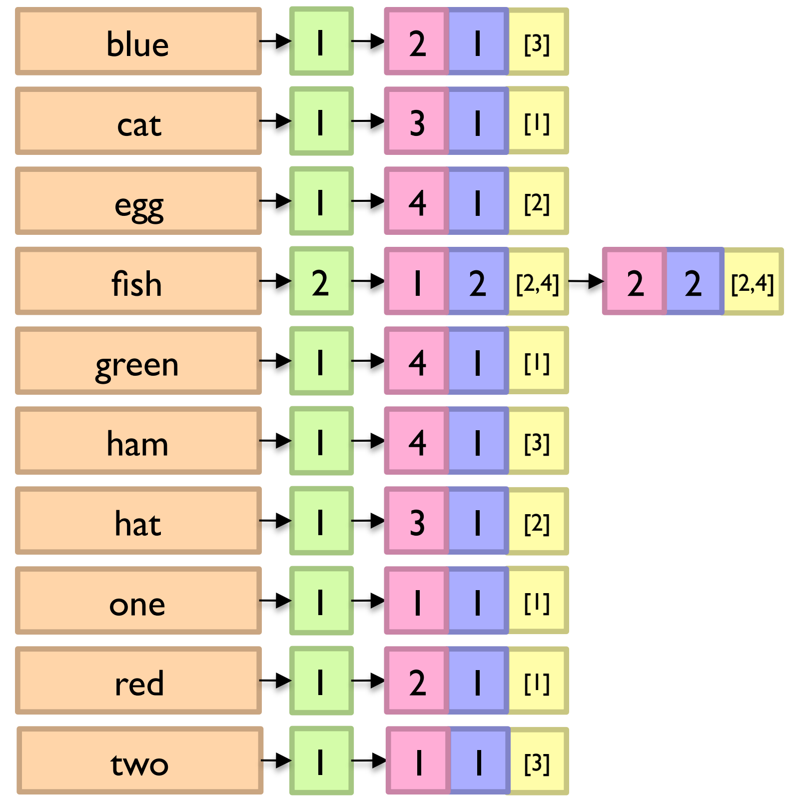
Doc 1
one fish, two fish

Doc 2
red fish, blue fish

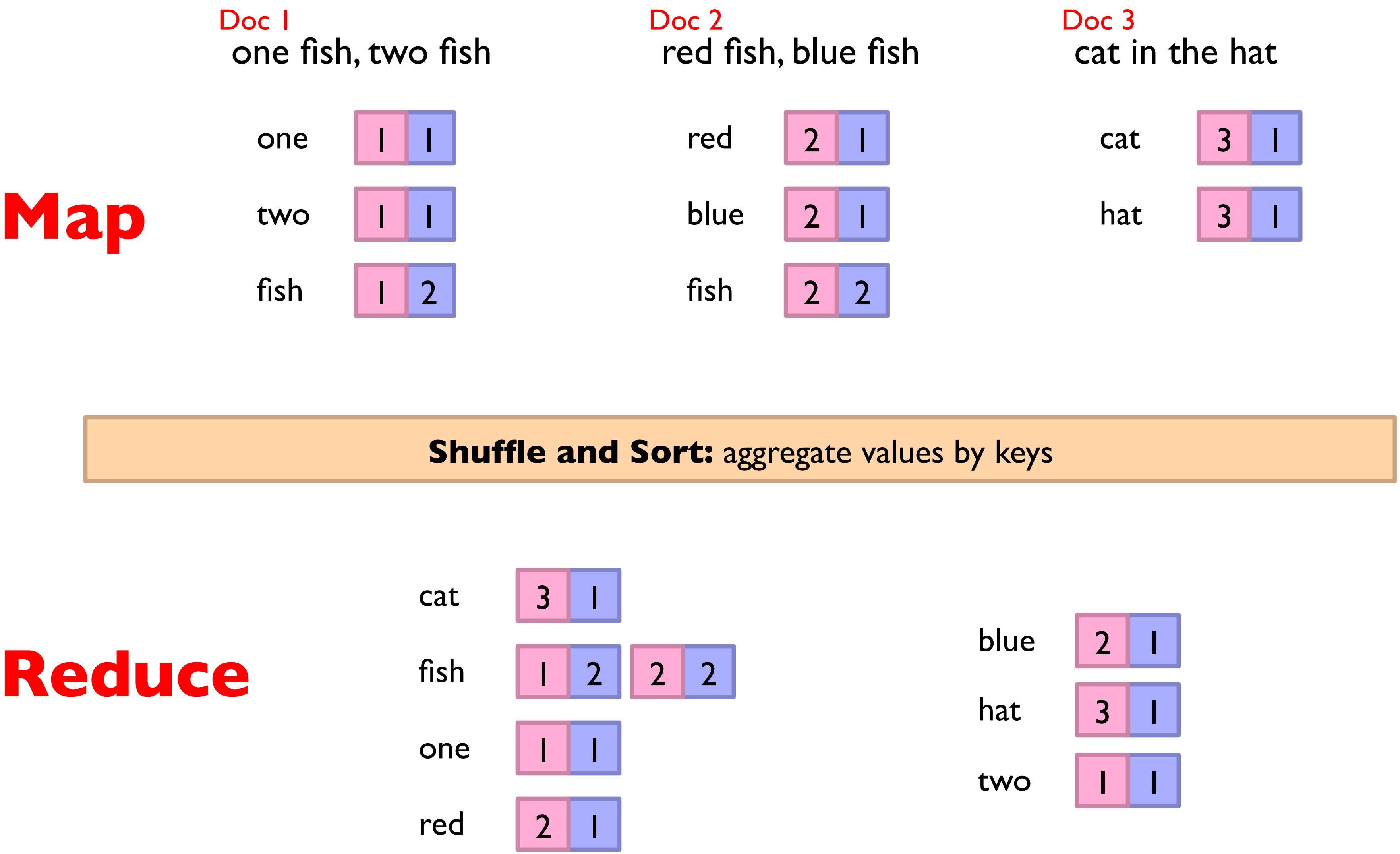
Doc 3
cat in the hat

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



Inverted Indexing with MapReduce

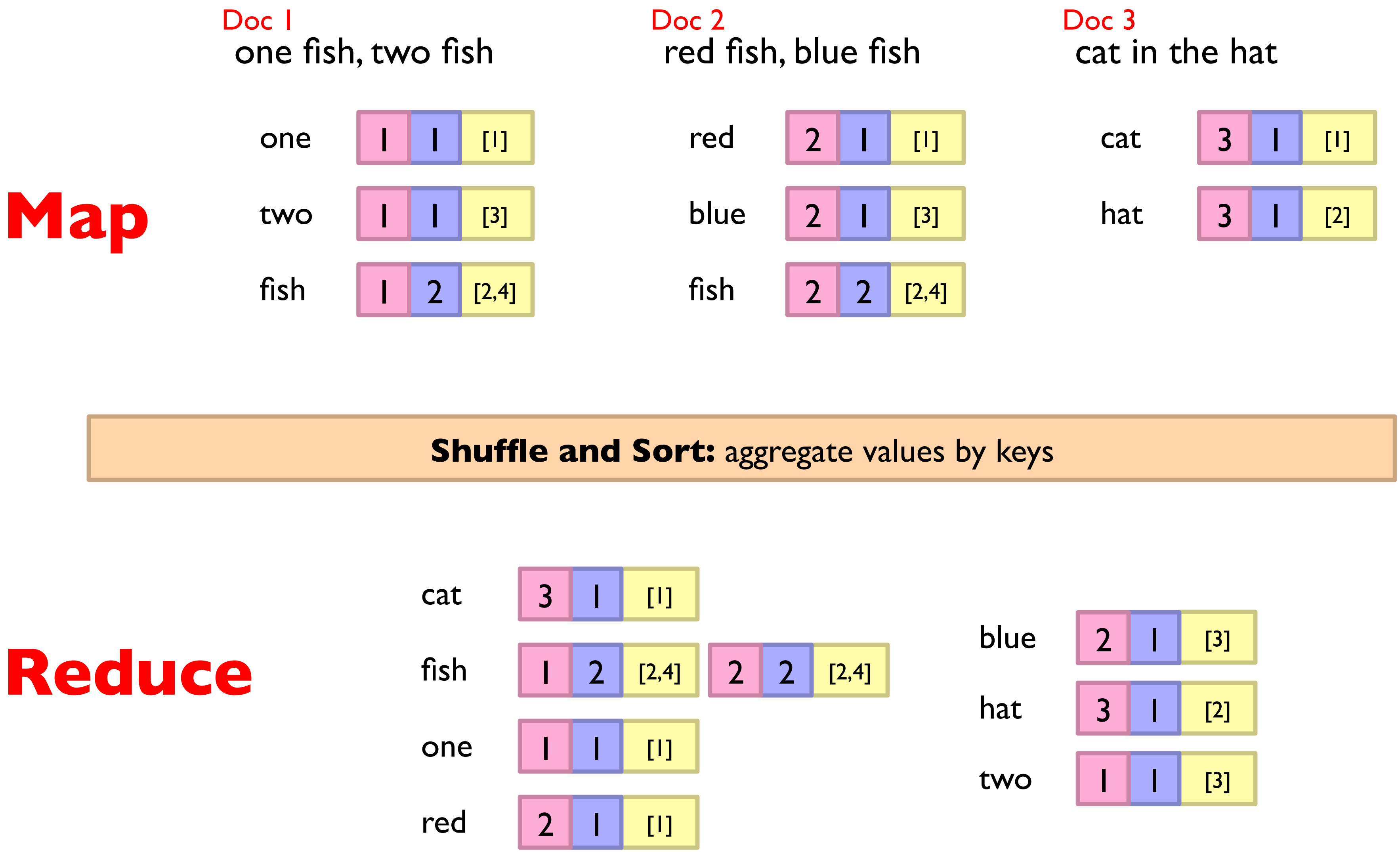


Inverted Indexing: Pseudo-Code

```
class Mapper
    method Map(docid n, doc d)
        H ← new AssociativeArray                    ▷ histogram to hold term frequencies
        for all term t ∈ doc d do                    ▷ processes the doc, e.g., tokenization and stopword removal
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(term t, posting ⟨n, H{t}⟩)          ▷ emits individual postings

class Reducer
    method Reduce(term t, postings [⟨n1, f1⟩ ...])
        P ← new List
        for all ⟨n, f⟩ ∈ postings [⟨n1, f1⟩ ...] do
            P.Append(⟨n, f⟩)                         ▷ appends postings unsorted
        P.Sort()                                     ▷ sorts for compression
        Emit(term t, postingsList P)
```

Positional Indexes



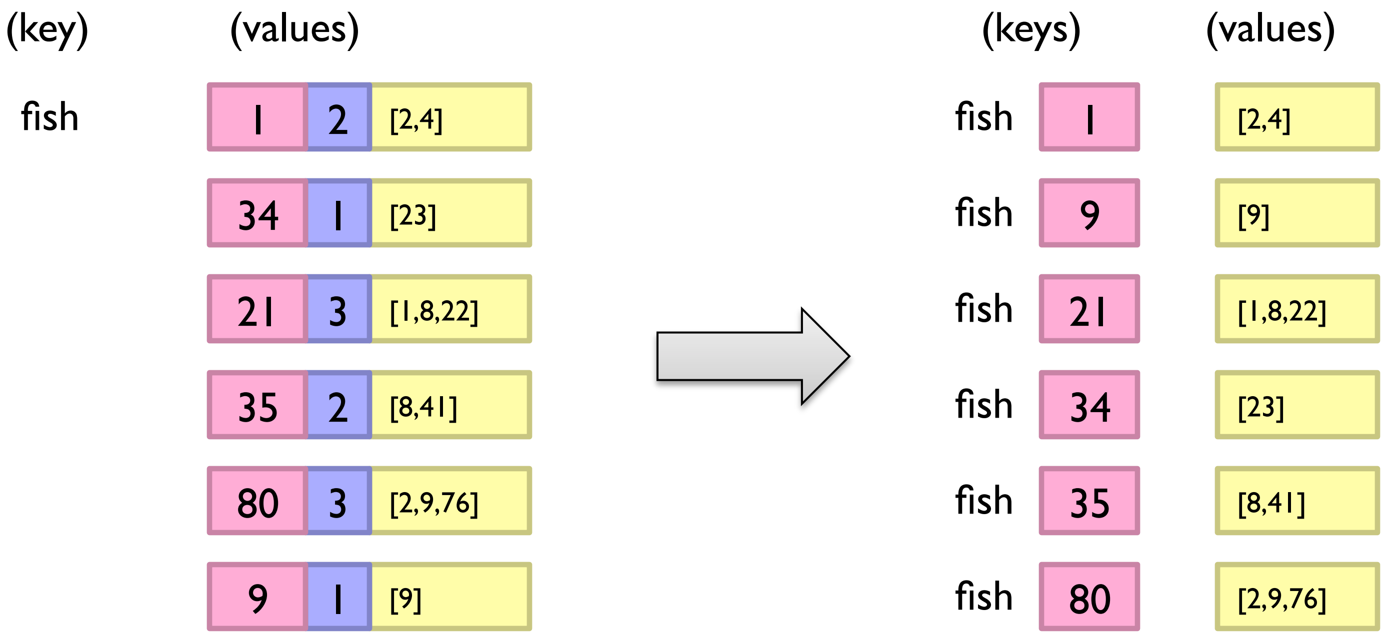
Inverted Indexing: Pseudo-Code

class MAPPER
  method MAP(docid n, doc d)
    $H \leftarrow$ new ASSOCIATIVEARRAY ▷ histogram to hold term frequencies
    for all term $t \in$ doc d **do** ▷ processes the doc, e.g., tokenization and stopword removal
      $H\{t\} \leftarrow H\{t\} + 1$
    for all term $t \in H$ **do**
      EMIT(term t, posting $\langle n, H\{t\}\rangle$) ▷ emits individual postings

class REDUCER
  method REDUCE(term t, postings $[\langle n_1, f_1\rangle \ldots]$)
    $P \leftarrow$ new LIST
    for all $\langle n, f\rangle \in$ postings $[\langle n_1, f_1\rangle \ldots]$ **do**
      P.APPEND($\langle n, f\rangle$) ▷ appends postings unsorted
    P.SORT() ▷ sorts for compression
    EMIT(term t, postingsList P)

What's the problem?

Another Try…



How is this different?

- Let the framework do the sorting
- Term frequency implicitly stored

Where have we seen this before?

Inverted Indexing: Pseudo-Code

```
class Mapper
    method Map(docid n, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do                        ▷ builds a histogram of term frequencies
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(tuple ⟨t, n⟩, tf H{t})                  ▷ emits individual postings, with a tuple as the key

class Partitioner
    method Partition(tuple ⟨t, n⟩, tf f)
        return Hash(t) mod NumOfReducers                 ▷ keys of same term are sent to same reducer

class Reducer
    method Initialize
        t_prev ← ∅
        P ← new PostingsList
    method Reduce(tuple ⟨t, n⟩, tf [f])
        if t ≠ t_prev ∧ t_prev ≠ ∅ then
            Emit(term t, postings P)                     ▷ emits postings list of term t_prev
            P.Reset()
        P.Append(⟨n, f⟩)                                 ▷ appends postings in sorted order
        t_prev ← t
    method Close
        Emit(term t, postings P)                         ▷ emits last postings list from this reducer
```

Postings Encoding

Conceptually:

fish	1	2	9	1	21	3	34	1	35	2	80	3	…

In Practice:

- Don't encode docnos, 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	…

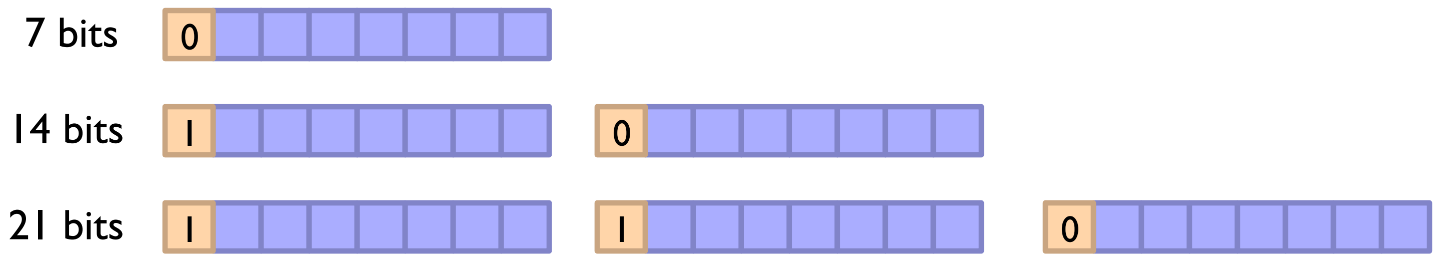
= delta encoding, delta compression, gap compression

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

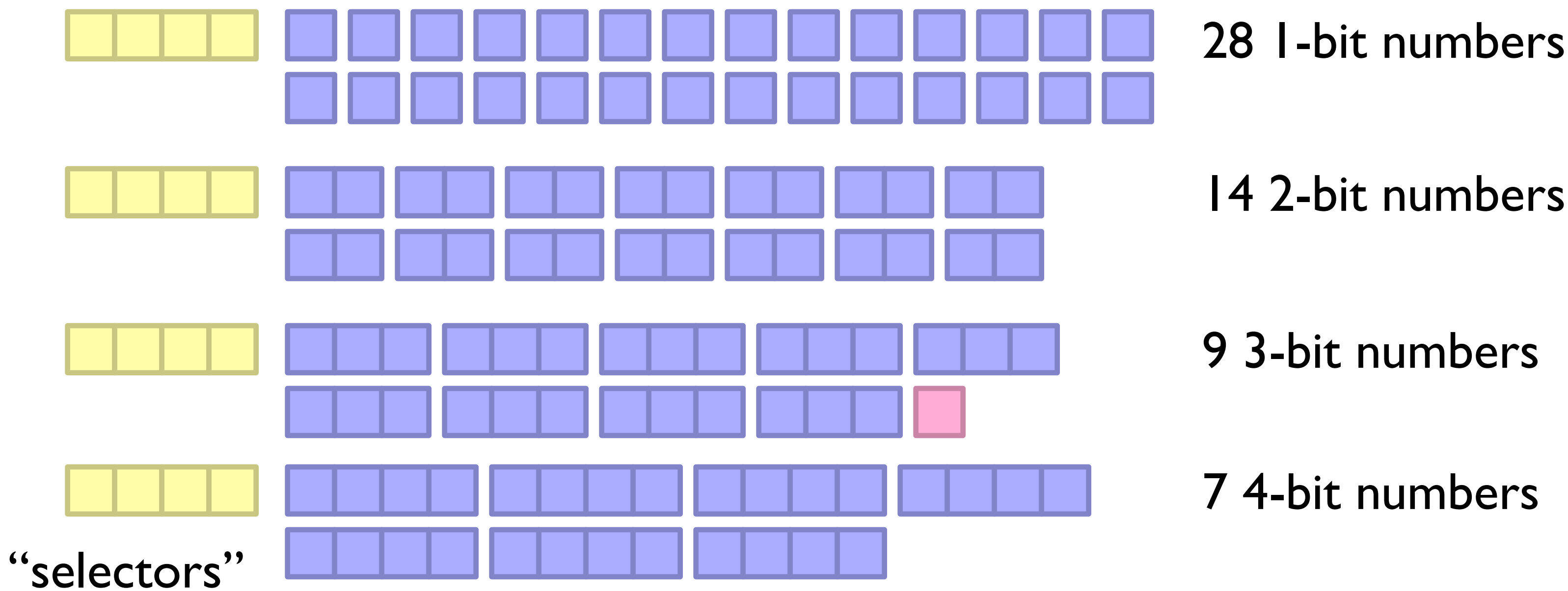


- Works okay, easy to implement…

Beware of branch mispredicts!

Simple-9

- How many different ways can we divide up 28 bits?



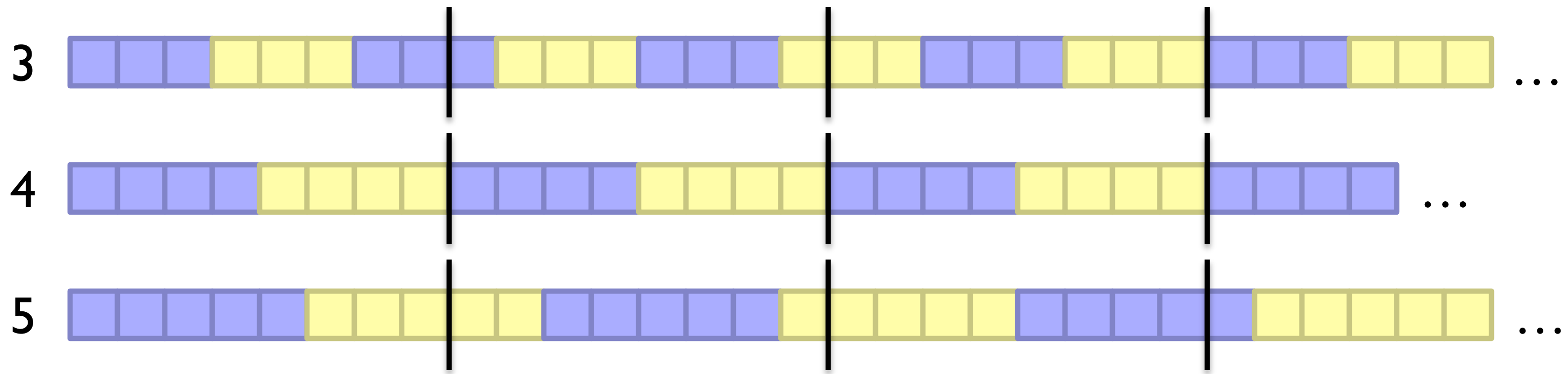
(9 total ways)

  - 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?



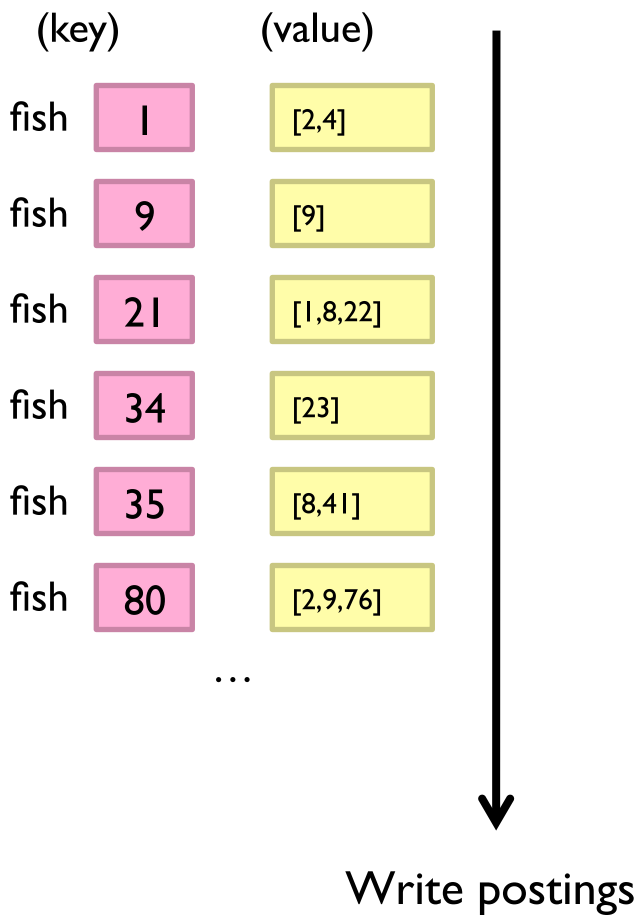
  - Efficient decompression with hard-coded decoders
- PForDelta – bit packing + separate storage of "overflow" bits

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
- Optimal $b \approx 0.69$ (N/df)
  - Different b for every term!

Chicken and Egg?



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

- Recall: optimal $b \approx 0.69\ (N/df)$
- 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	[2,4]
one	1	[1]
two	1	[3]

Emit normal key-value pairs…

fish	★	[1]
one	★	[1]
two	★	[1]

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

Getting the *df*: Modified Reducer

(key)		(value)
fish	★	[63] [82] [27] …

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

Compute *b* from *df*

fish	1	[2,4]
fish	9	[9]
fish	21	[1,8,22]
fish	34	[23]
fish	35	[8,41]
fish	80	[2,9,76]
	…	

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

Write postings

Where have we seen this before?

Inverted Indexing: IP

```
class Mapper
    method Map(docid n, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do                      ▷ builds a histogram of term frequencies
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(tuple ⟨t, n⟩, tf H{t})                ▷ emits individual postings, with a tuple as the key

class Partitioner
    method Partition(tuple ⟨t, n⟩, tf f)
        return Hash(t) mod NumOfReducers               ▷ keys of same term are sent to same reducer

class Reducer
    method Initialize
        t_prev ← ∅
        P ← new PostingsList
    method Reduce(tuple ⟨t, n⟩, tf [f])
        if t ≠ t_prev ∧ t_prev ≠ ∅ then
            Emit(term t, postings P)                   ▷ emits postings list of term t_prev
            P.Reset()
        P.Append(⟨n, f⟩)                               ▷ appends postings in sorted order
        t_prev ← t
    method Close
        Emit(term t, postings P)                       ▷ emits last postings list from this reducer
```

What's the assumption?
Is it okay?

Merging Postings

- Let's define an operation ⊕ on postings lists *P*:

  Postings(1, 15, 22, 39, 54) ⊕ 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: ⊕

What's the issue?

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

Solution: apply compression as needed!

Inverted Indexing: LP

Slightly less elegant implementation… but uses same idea

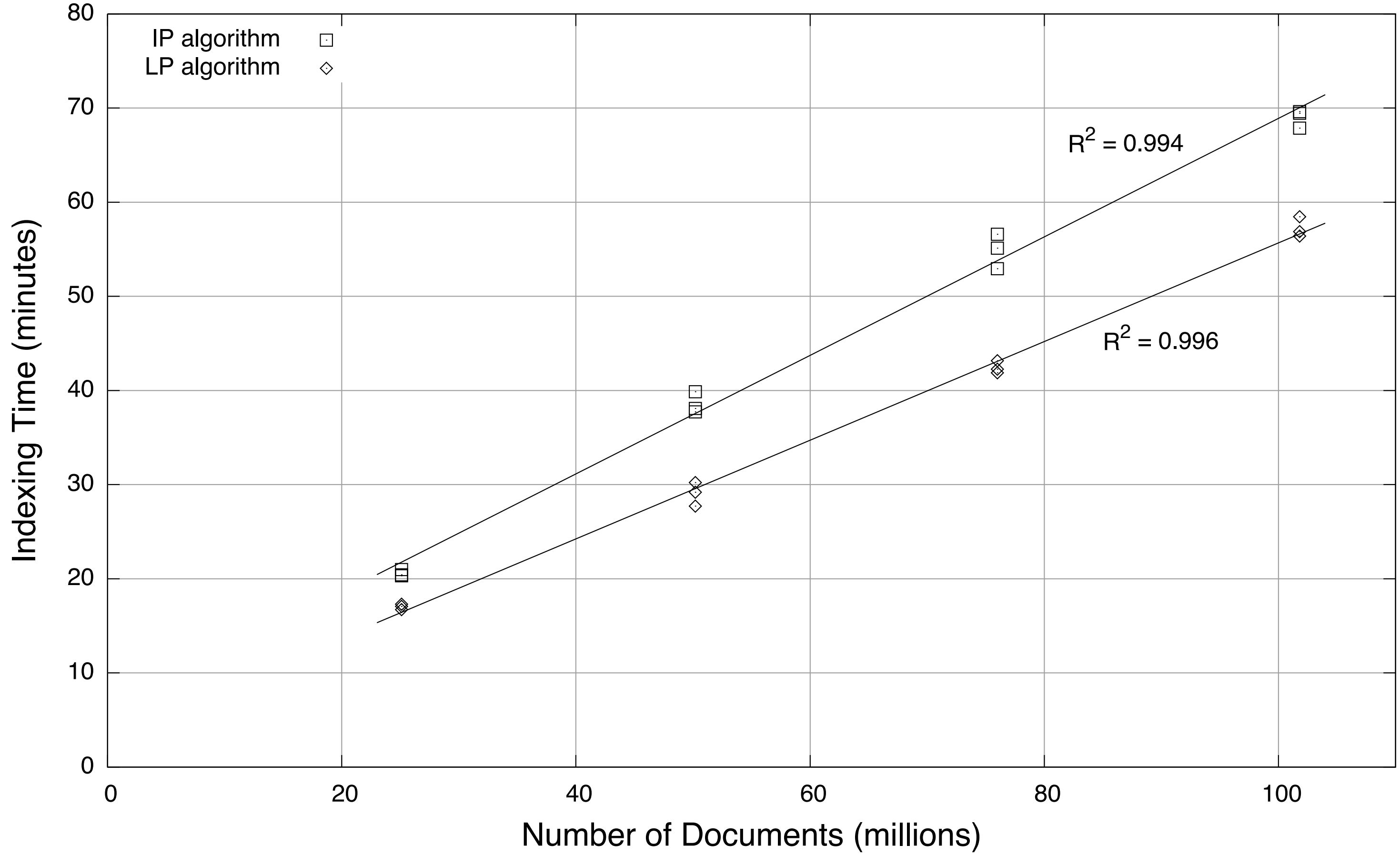
```
class Mapper
    method Initialize
        M ← new AssociativeArray                          ▷ holds partial lists of postings
    method Map(docid n, doc d)
        H ← new AssociativeArray                          ▷ builds a histogram of term frequencies
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            M{t}.Add(posting ⟨n, H{t}⟩)                   ▷ adds a posting to partial postings lists
        if MemoryFull() then
            Flush()
    method Flush                                          ▷ flushes partial lists of postings as intermediate output
        for all term t ∈ M do
            P ← SortAndEncodePostings(M{t})
            Emit(term t, postingsList P)
        M.Clear()
    method Close
        Flush()
```

Inverted Indexing: LP

```
class Reducer
    method Reduce(term t, postingsLists [P1, P2, ...])
        Pf ← new List                          ▷ temporarily stores partial lists of postings
        R ← new List                           ▷ stores merged partial lists of postings
        for all P ∈ postingsLists [P1, P2, ...] do
            Pf.Add(P)
            if MemoryNearlyFull() then
                R.Add(MergeLists(Pf))
                Pf.Clear()
        R.Add(MergeLists(Pf))
        Emit(term t, postingsList MergeLists(R))    ▷ emits fully merged postings list of term t
```

IP vs. LP?

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

Remind you of anything in Spark?

```
class Mapper
    method Initialize
        M ← new AssociativeArray                          ▷ holds partial lists of postings
    method Map(docid n, doc d)
        H ← new AssociativeArray                          ▷ builds a histogram of term frequencies
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            M{t}.Add(posting ⟨n, H{t}⟩)                   ▷ adds a posting to partial postings lists
        if MemoryFull() then
            Flush()
    method Flush                  ▷ flushes partial lists of postings as intermediate output
        for all term t ∈ M do
            P ← SortAndEncodePostings(M{t})
            Emit(term t, postingsList P)
        M.Clear()
    method Close
        Flush()
```

```
class Reducer
    method Reduce(term t, postingsLists [P1, P2, ...])
        Pf ← new List                                     ▷ temporarily stores partial lists
        R ← new List                                      ▷ stores merged partial lists
        for all P ∈ postingsLists [P1, P2, ...] do
            Pf.Add(P)
            if MemoryNearlyFull() then
                R.Add(MergeLists(Pf))
                Pf.Clear()
        R.Add(MergeLists(Pf))
        Emit(term t, postingsList MergeLists(R))          ▷ emits fully merged postings list of term t
```

RDD[(K, V)]

↓

aggregateByKey
seqOp: (U, V) ⇒ U,
combOp: (U, U) ⇒ U

↓

RDD[(K, U)]

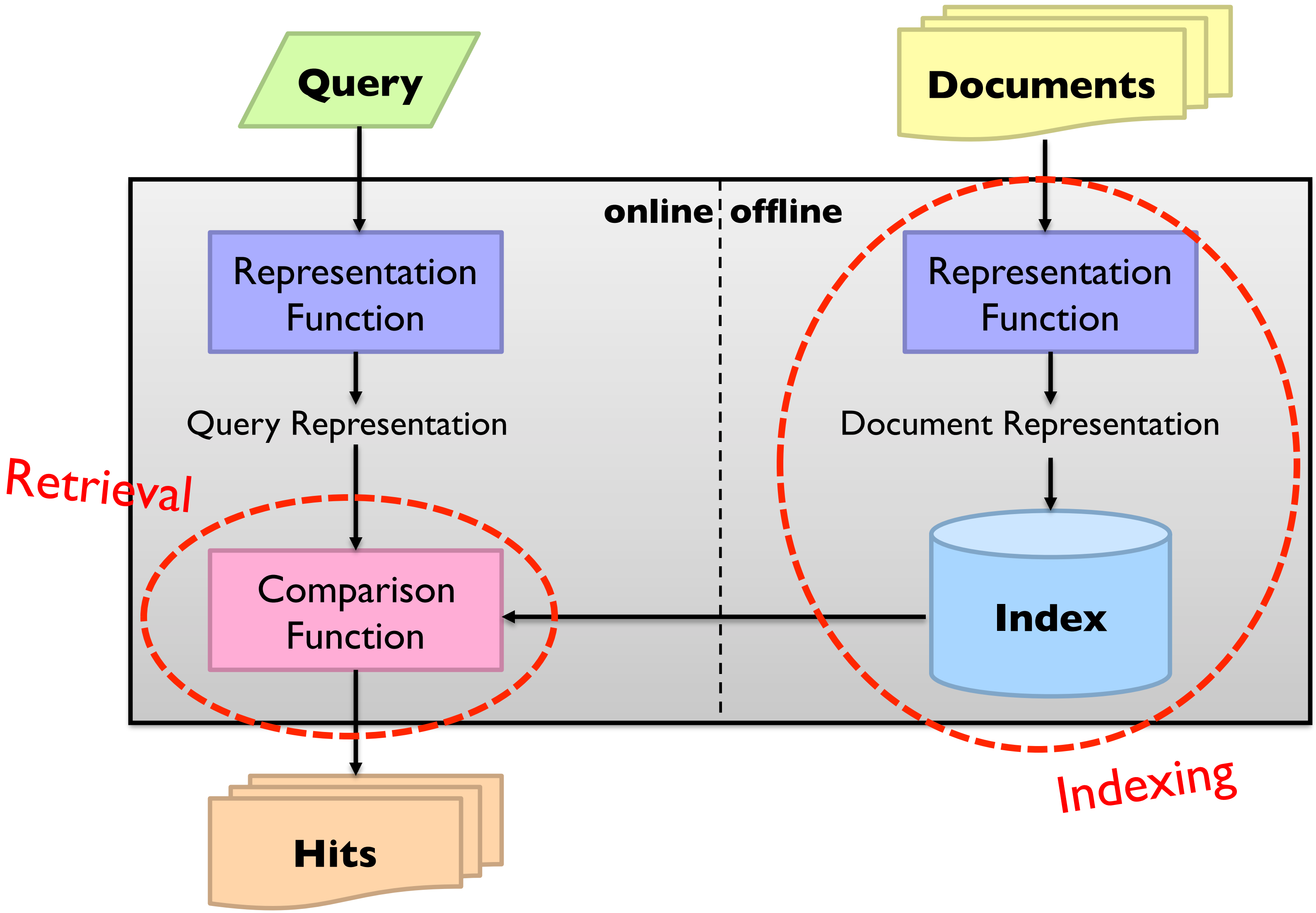
Algorithm design in a nutshell…

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



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

MapReduce it?

- 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

Perfect for MapReduce!

- 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…

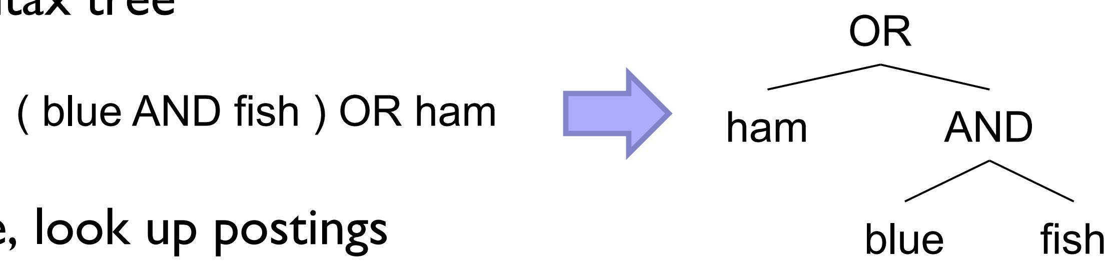
(For now)

Boolean Retrieval

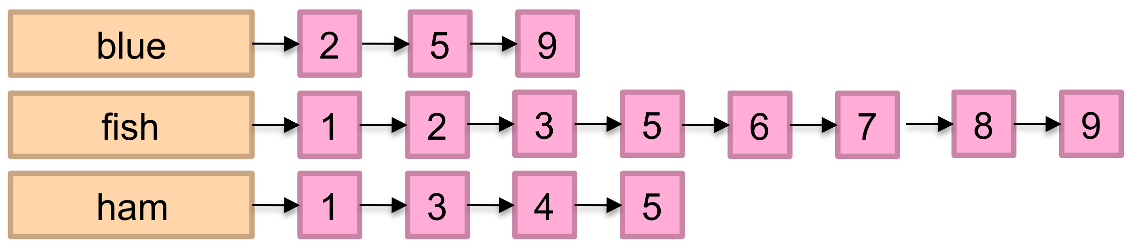
- Users express queries as a Boolean expression
  - AND, OR, NOT
  - Can be arbitrarily nested
- Retrieval is based on the notion of sets
  - Any given query divides the collection into two sets: retrieved, not-retrieved
  - Pure Boolean systems do not define an ordering of the results

Boolean Retrieval

- To execute a Boolean query:
  - Build query syntax tree

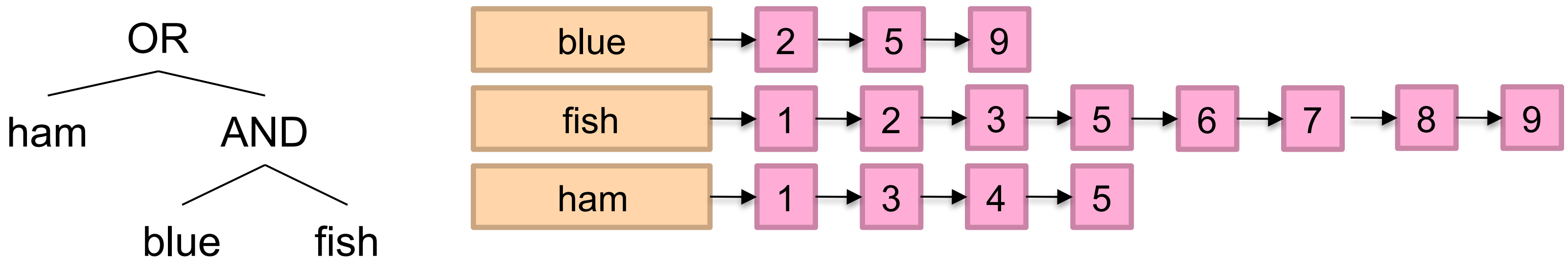


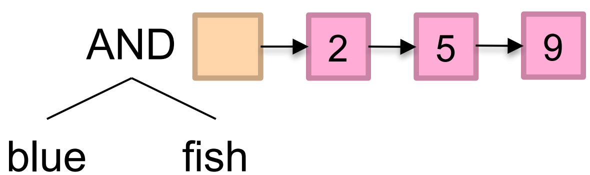
  - For each clause, look up postings



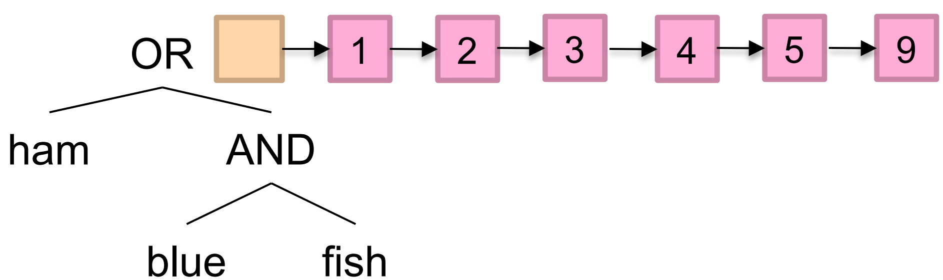
  - Traverse postings and apply Boolean operator

Term-at-a-Time



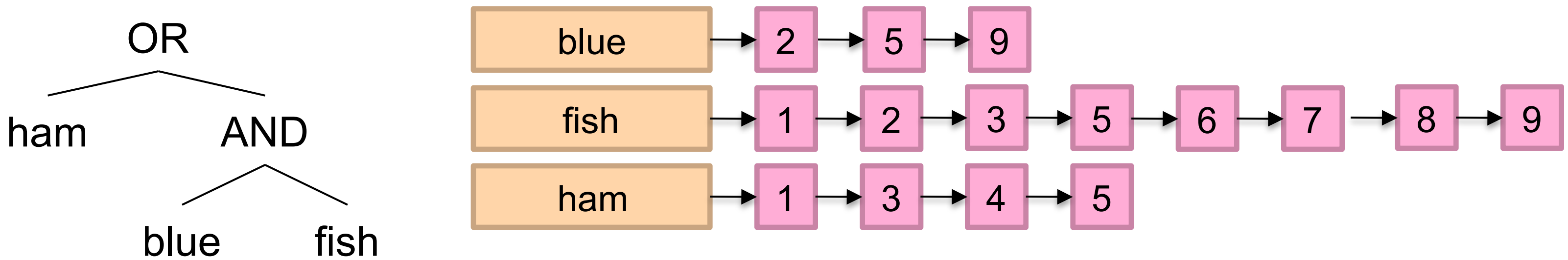


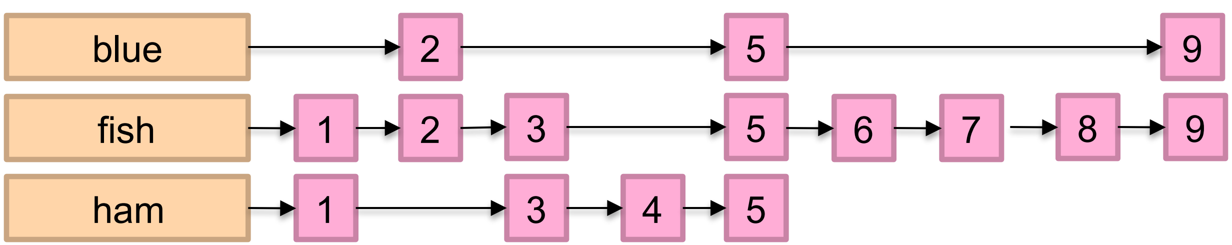
Efficiency analysis?



What's RPN?

Document-at-a-Time





Tradeoffs?

Efficiency analysis?

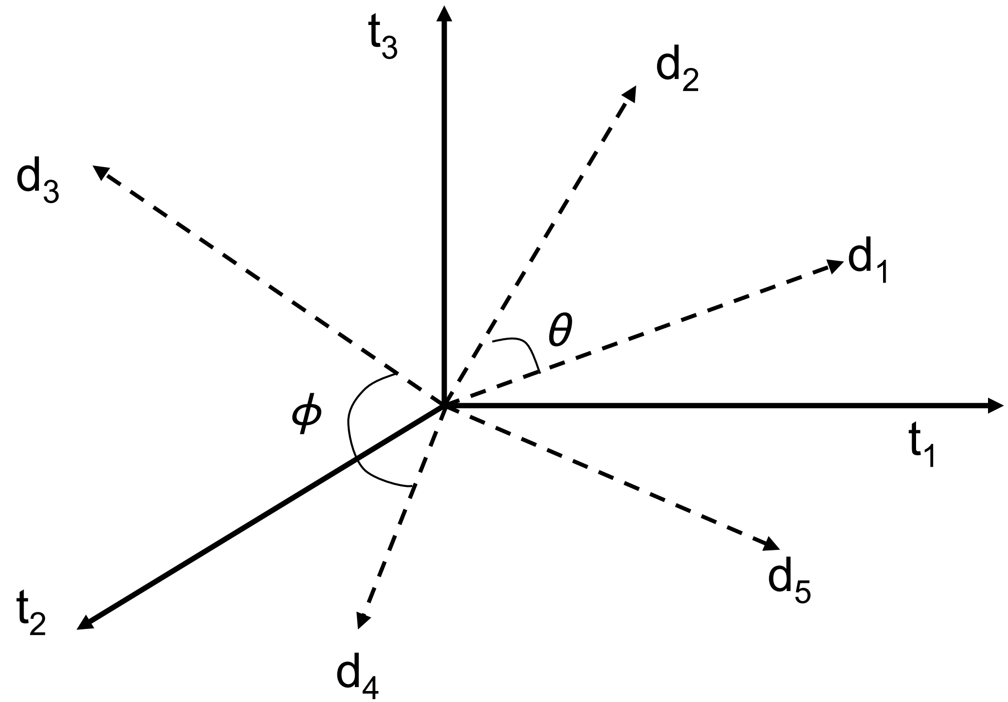
Strengths and Weaknesses

- Strengths
  - Precise, if you know the right strategies
  - Precise, if you have an idea of what you're looking for
  - Implementations are fast and efficient
- Weaknesses
  - Users must learn Boolean logic
  - Boolean logic insufficient to capture the richness of language
  - No control over size of result set: either too many hits or none
  - When do you stop reading? All documents in the result set are considered "equally good"
  - What about partial matches? Documents that "don't quite match" the query may be useful also

Ranked Retrieval

- Order documents by how likely they are to be relevant
  - Estimate relevance(q, d_i)
  - Sort documents by relevance
  - Display sorted results
- User model
  - Present hits one screen at a time, best results first
  - At any point, users can decide to stop looking
- How do we estimate relevance?
  - Assume document is relevant if it has a lot of query terms
  - Replace relevance(q, d_i) with sim(q, d_i)
  - Compute similarity of vector representations

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:

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

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

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

- 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

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

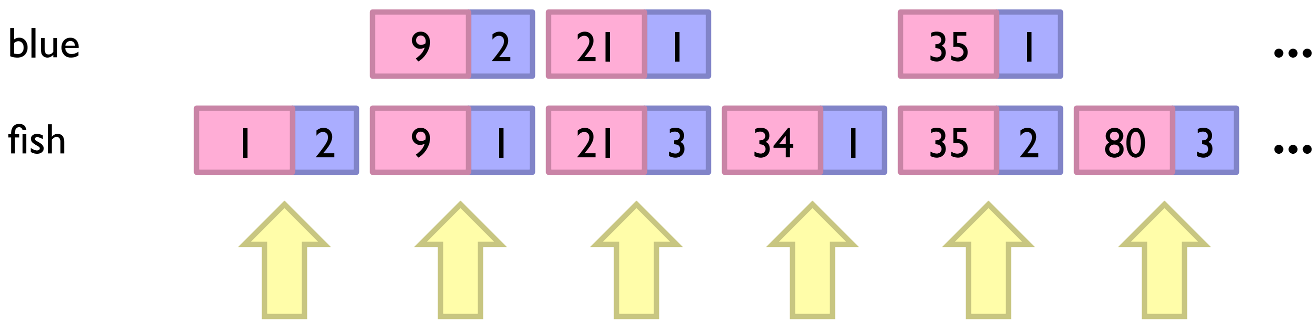
$w_{i,j}$	weight assigned to term *i* in document *j*
$\mathrm{tf}_{i,j}$	number of occurrence of term *i* in document *j*
N	number of documents in entire collection
n_i	number of documents with term *i*

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)

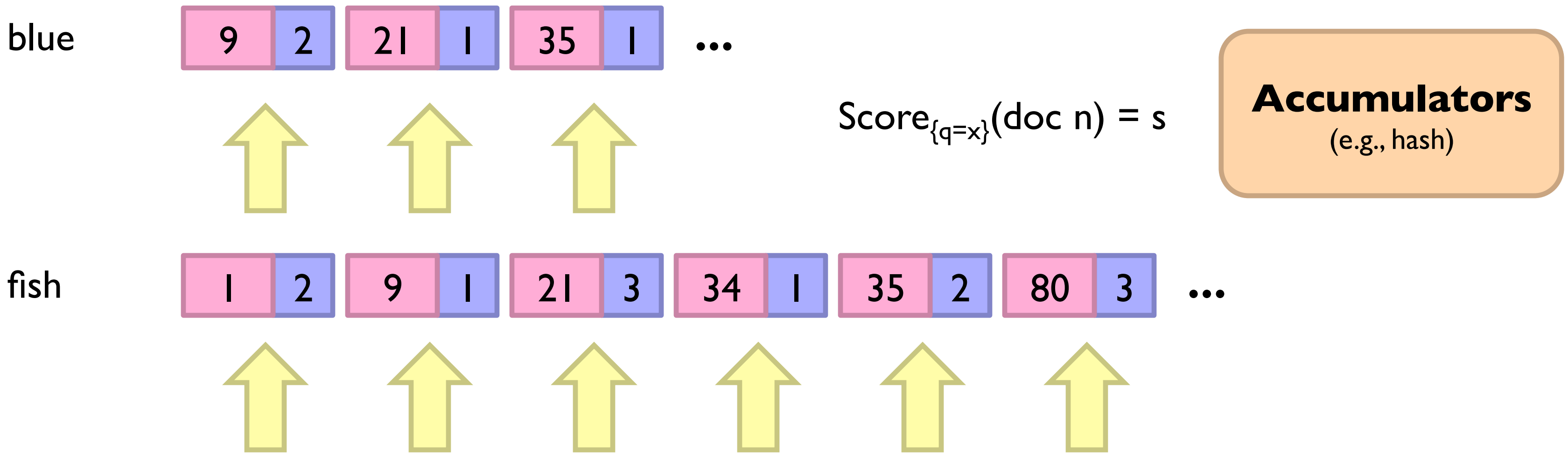
Document score in top k?

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

Assume everything fits in memory on a single machine…

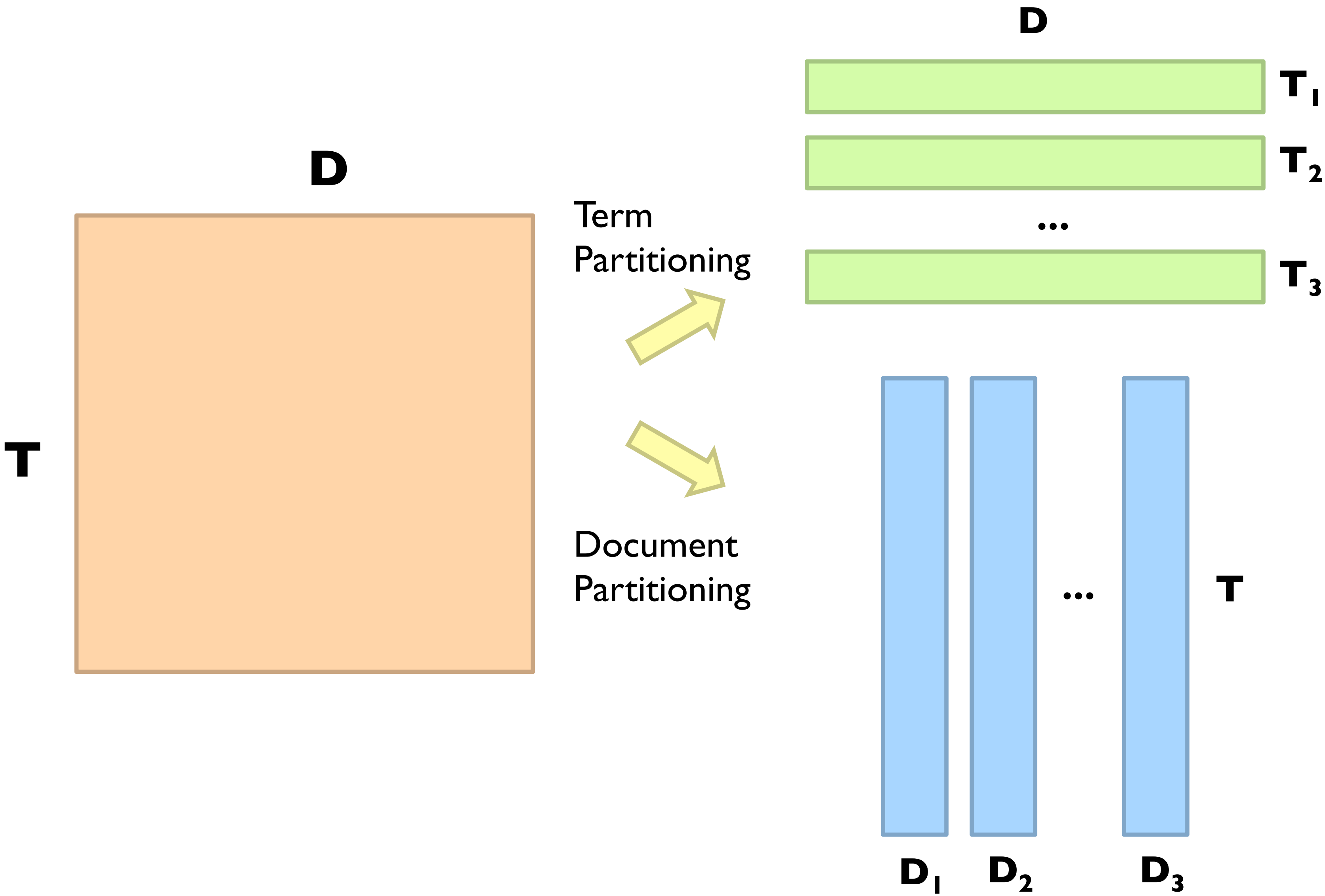
Okay, let's relax this assumption now

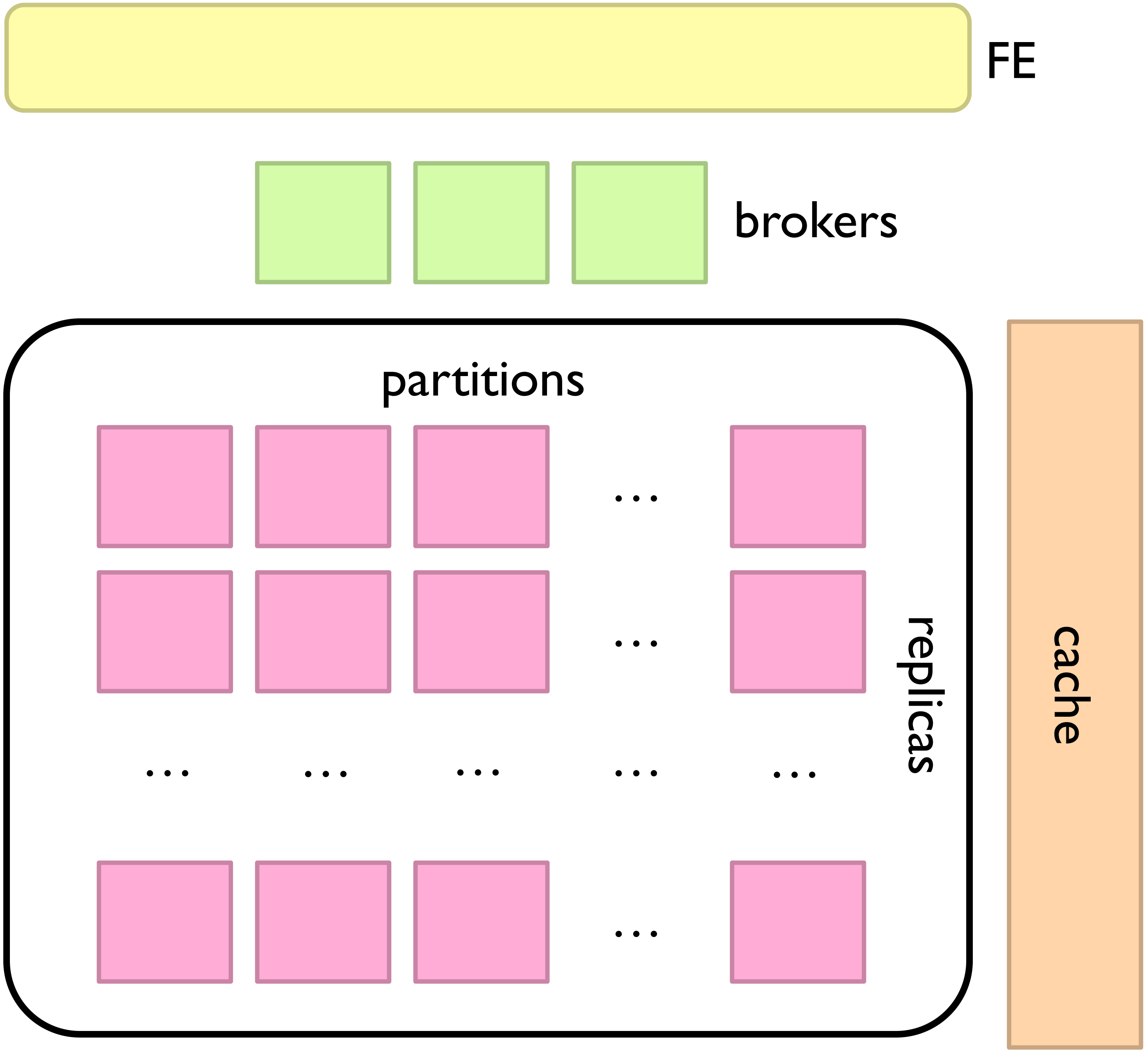
Important Ideas

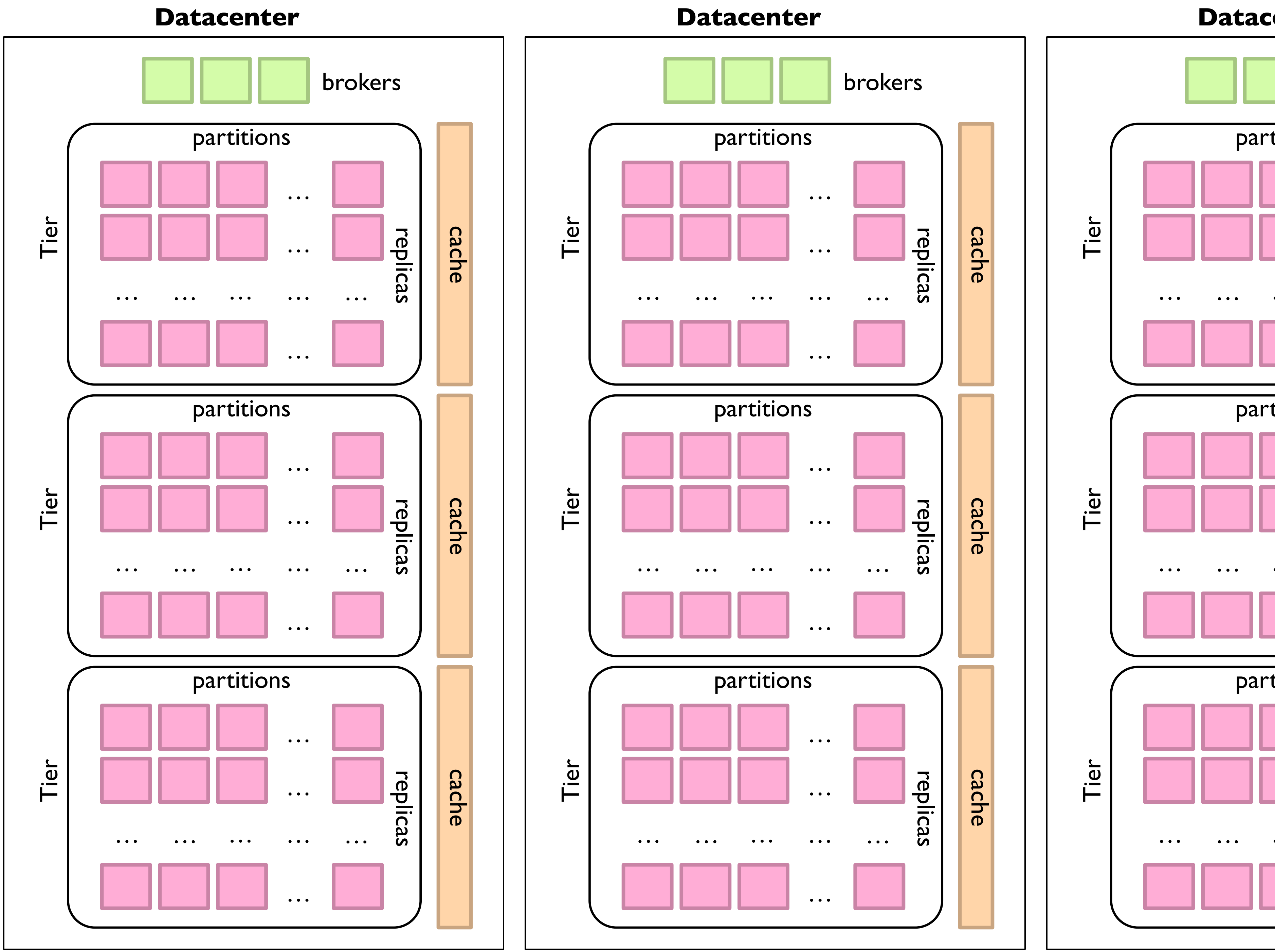
- Partitioning (for scalability)
- Replication (for redundancy)
- Caching (for speed)
- Routing (for load balancing)

The rest is just details!

Term vs. Document Partitioning

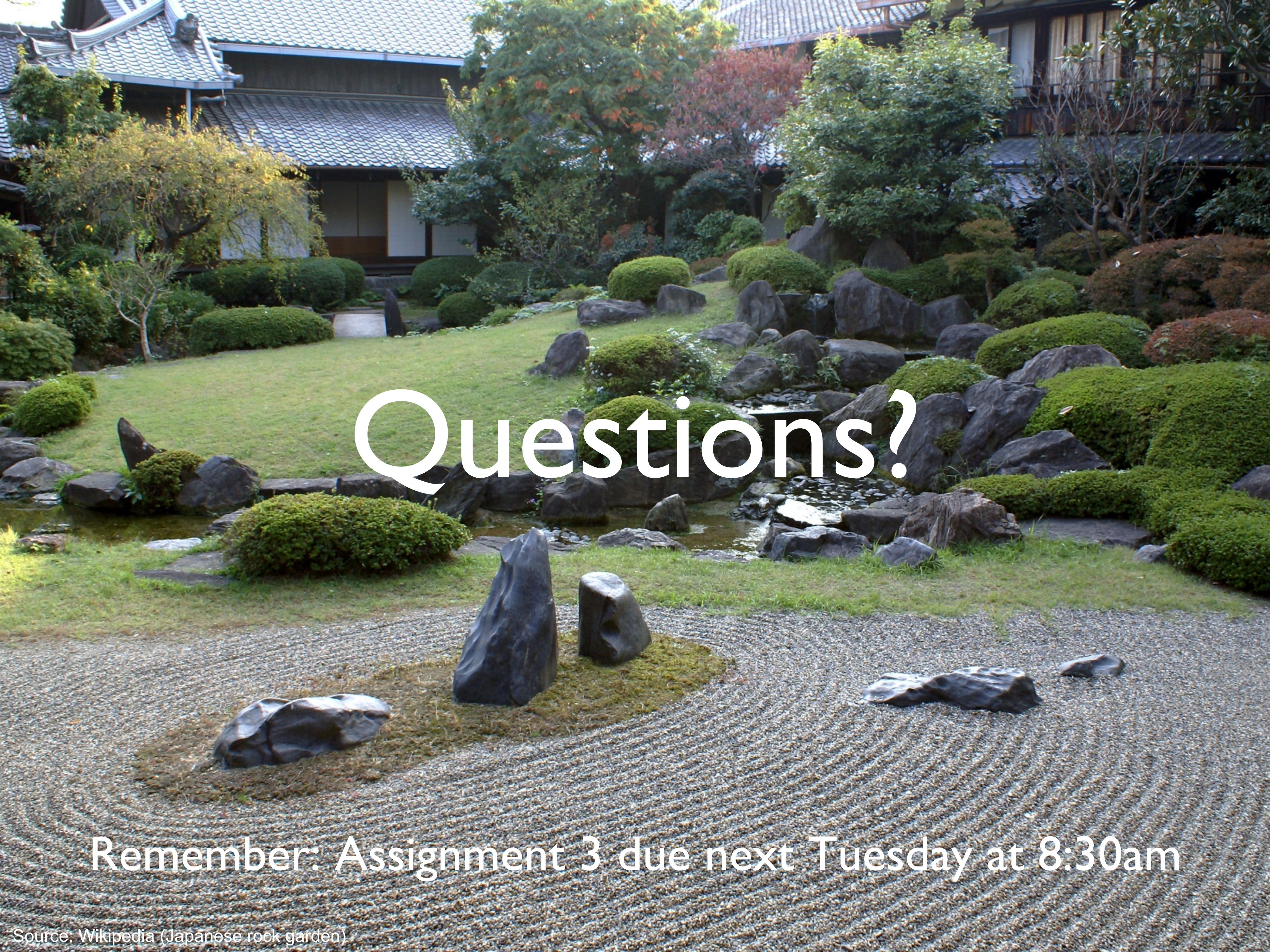



FE
brokers
partitions
replicas
cache


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

Important Ideas

- Partitioning (for scalability)
- Replication (for redundancy)
- Caching (for speed)
- Routing (for load balancing)


Questions?
Remember: Assignment 3 due next Tuesday at 8:30am
Source: Wikipedia (Japanese rock garden)