Information Retrieval Processing with MapReduce

Based on Jimmy Lin's Tutorial at the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2009)

How much data?

- Google processes 20 PB a day (2008)
- Wayback Machine has 3 PB + 100 TB/month (3/2009)
- Facebook has 2.5 PB of user data + 15 TB/day (4/2009)
- eBay has 6.5 PB of user data + 50 TB/day (5/2009)
- CERN's LHC will generate 15 PB a year (??)

640K ought to be enough for anybody.

MapReduce

e.g., Amazon Web Services

cheap commodity clusters (or utility computing)
+ simple distributed programming models
+ availability of large datasets
= data-intensive IR research for the masses!

ClueWeb09

ClueWeb09

- NSF-funded project, led by Jamie Callan (CMU/LTI)
- It's big!
  - 1 billion web pages crawled in Jan./Feb. 2009
  - 10 languages, 500 million pages in English
  - 5 TB compressed, 25 uncompressed
- It's available!
  - Available to the research community
  - Test collection coming (TREC 2009)

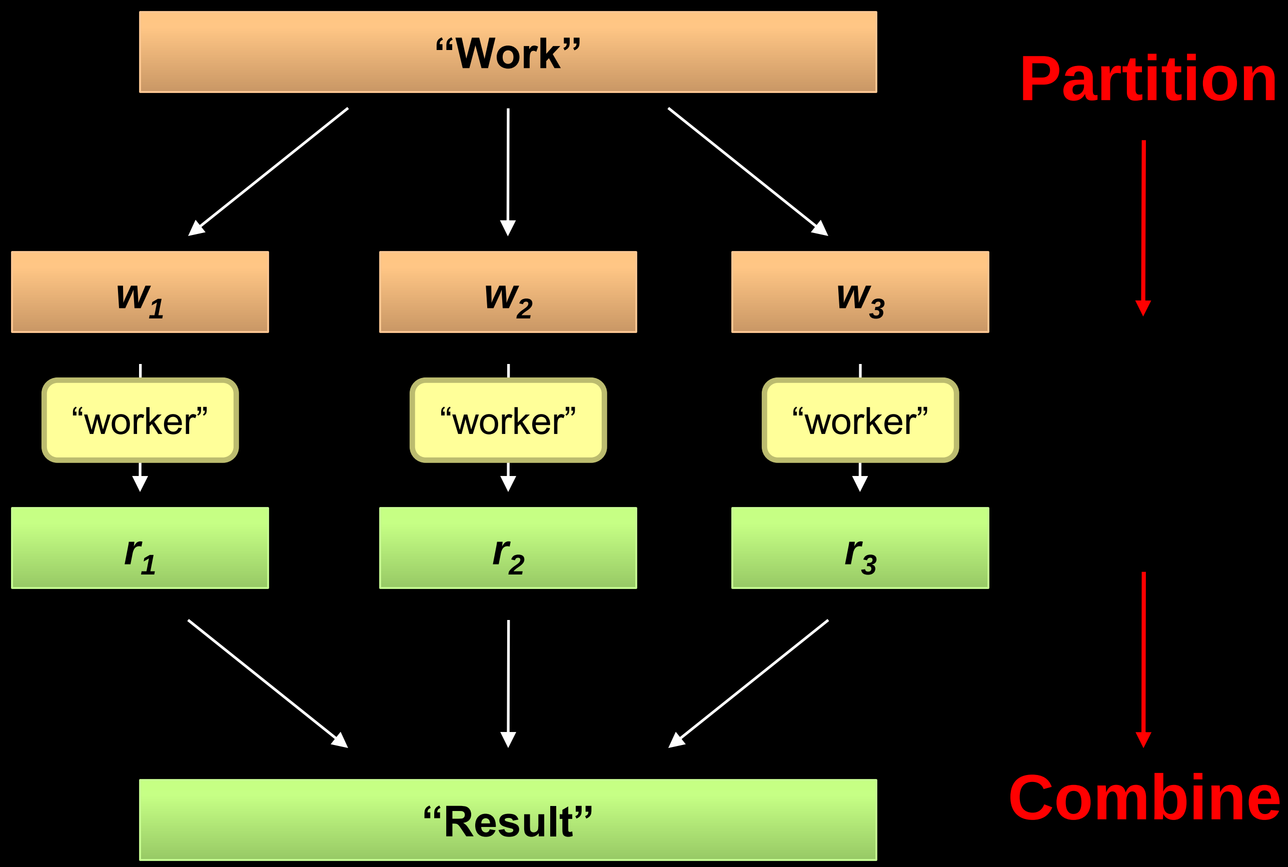
Ivory and SMRF

- Collaboration between:
  - University of Maryland
  - Yahoo! Research
- Reference implementation for a Web-scale IR toolkit
  - Designed around Hadoop from the ground up
  - Written specifically for the ClueWeb09 collection
  - Implements some of the algorithms described in this tutorial
  - Features SMRF query engine based on Markov Random Fields
- Open source
  - Initial release available now!

Cloud^9

- Set of libraries originally developed for teaching MapReduce at the University of Maryland
  - Demos, exercises, etc.
- “Eat you own dog food”
  - Actively used for a variety of research projects

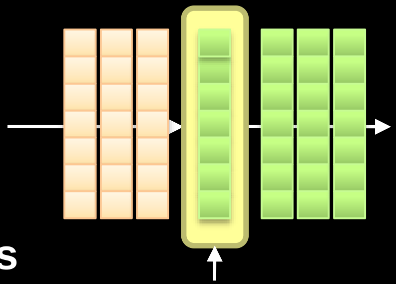
Divide and Conquer



It's a bit more complex…

Fundamental issues

scheduling, data distribution, synchronization, inter-process communication, robustness, fault tolerance, …

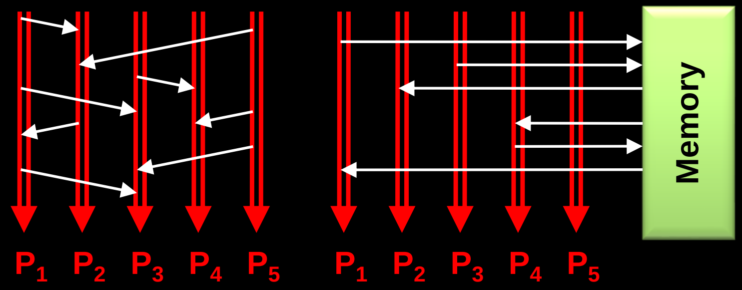


Architectural issues

Flynn's taxonomy (SIMD, MIMD, etc.), network typology, bisection bandwidth UMA vs. NUMA, cache coherence

Different programming models

Message Passing Shared Memory



Different programming constructs

mutexes, conditional variables, barriers, …
masters/slaves, producers/consumers, work queues, …

Common problems

livelock, deadlock, data starvation, priority inversion…
dining philosophers, sleeping barbers, cigarette smokers, …

The reality: programmer shoulders the burden of managing concurrency…

Source: Ricardo Guimarães Herrmann

Typical Large-Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

Reduce

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

MapReduce Runtime

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS (later)

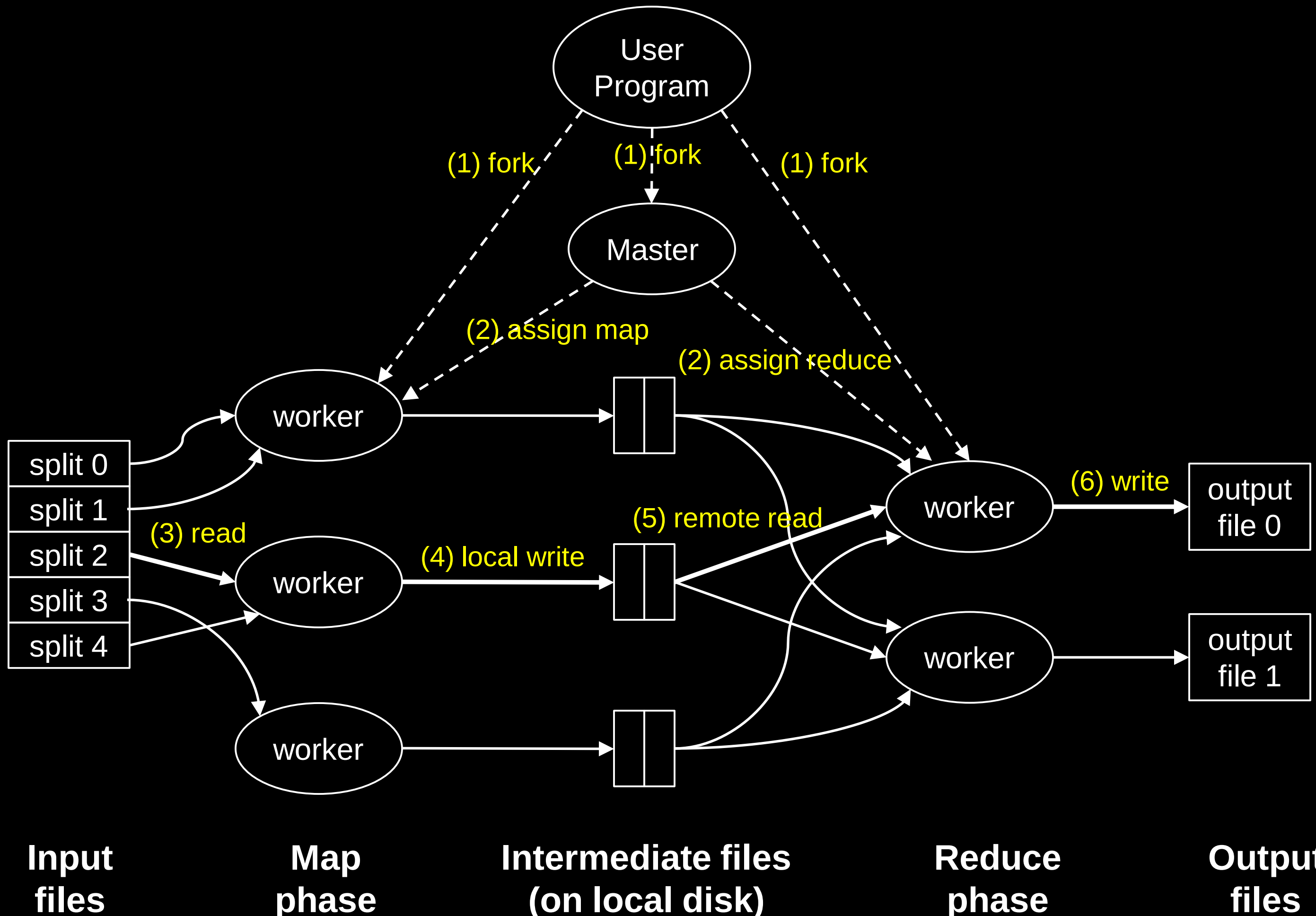
“Hello World”: Word Count

```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
        Emit(term, value);
```

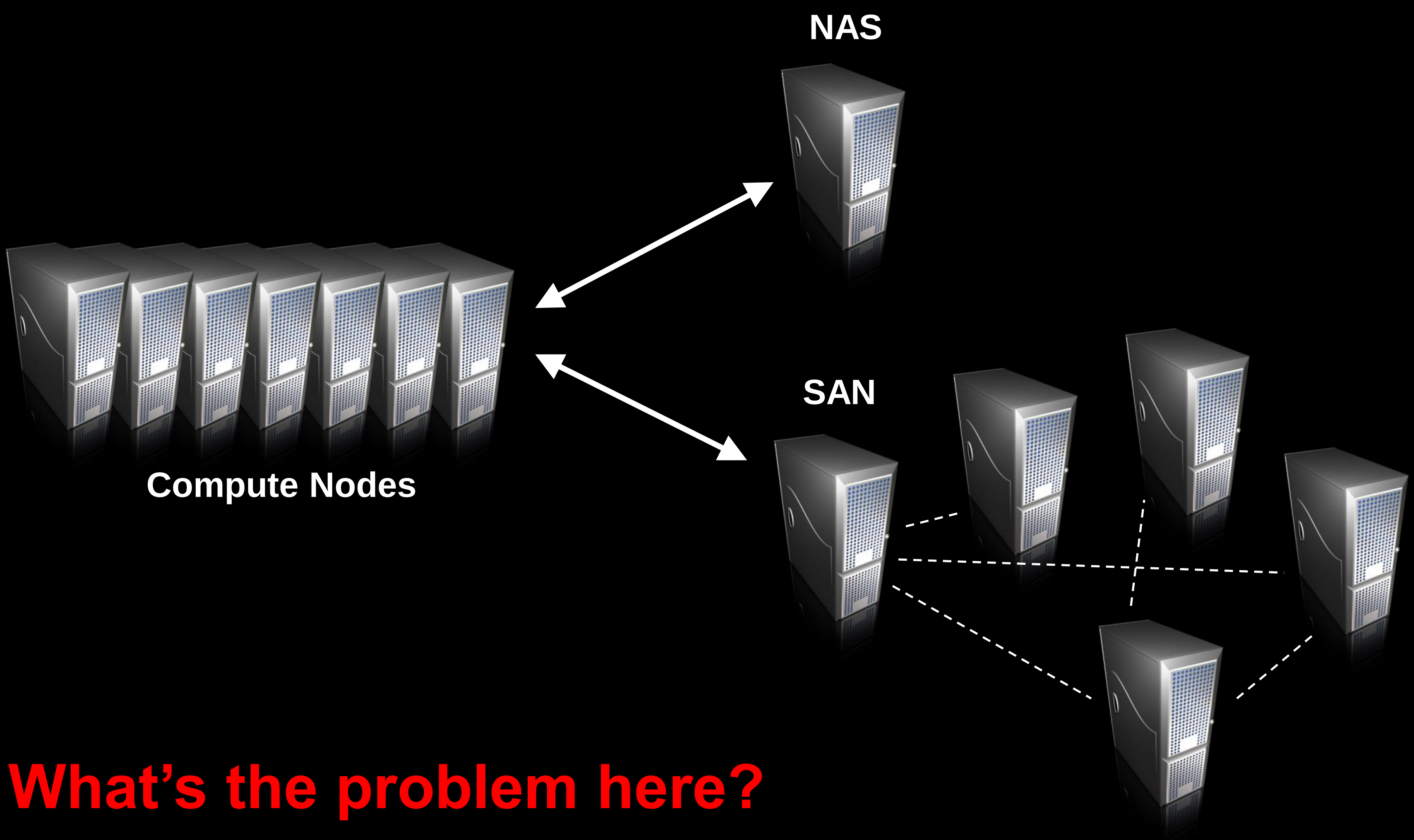
MapReduce Implementations

- MapReduce is a programming model
- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Project led by Yahoo, used in production
  - Rapidly expanding software ecosystem



Redrawn from (Dean and Ghemawat, OSDI 2004)

How do we get data to the workers?



What's the problem here?

Distributed File System

- Don’t move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System)
  - HDFS for Hadoop (= GFS clone)

GFS: Assumptions

- Commodity hardware over "exotic" hardware
  - Scale out, not up
- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of HUGE files
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
- High sustained throughput over low latency

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client

Master’s Responsibilities

- Metadata storage
- Namespace management/locking
- Periodic communication with chunkservers
- Chunk creation, re-replication, rebalancing
- Garbage collection

Managing Dependencies

- Remember: Mappers run in isolation
  - You have no idea in what order the mappers run
  - You have no idea on what node the mappers run
  - You have no idea when each mapper finishes
- Tools for synchronization:
  - Ability to hold state in reducer across multiple key-value pairs
  - Sorting function for keys
  - Partitioner
  - Cleverly-constructed data structures

Slides in this section adapted from work reported in (Lin, EMNLP 2008)

MapReduce: Large Counting Problems

- Term co-occurrence matrix for a text collection = specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of observations (the collection itself)
  - Goal: keep track of interesting statistics about the events
- Basic approach
  - Mappers generate partial counts
  - Reducers aggregate partial counts

How do we aggregate partial counts efficiently?

First Try: "Pairs"

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For all pairs, emit (a, b) → count
- Reducers sums up counts associated with these pairs
- Use combiners!

“Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)

Another Try: "Stripes"

- Idea: group together pairs into an associative array

(a, b) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
(a, f) → 2

a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

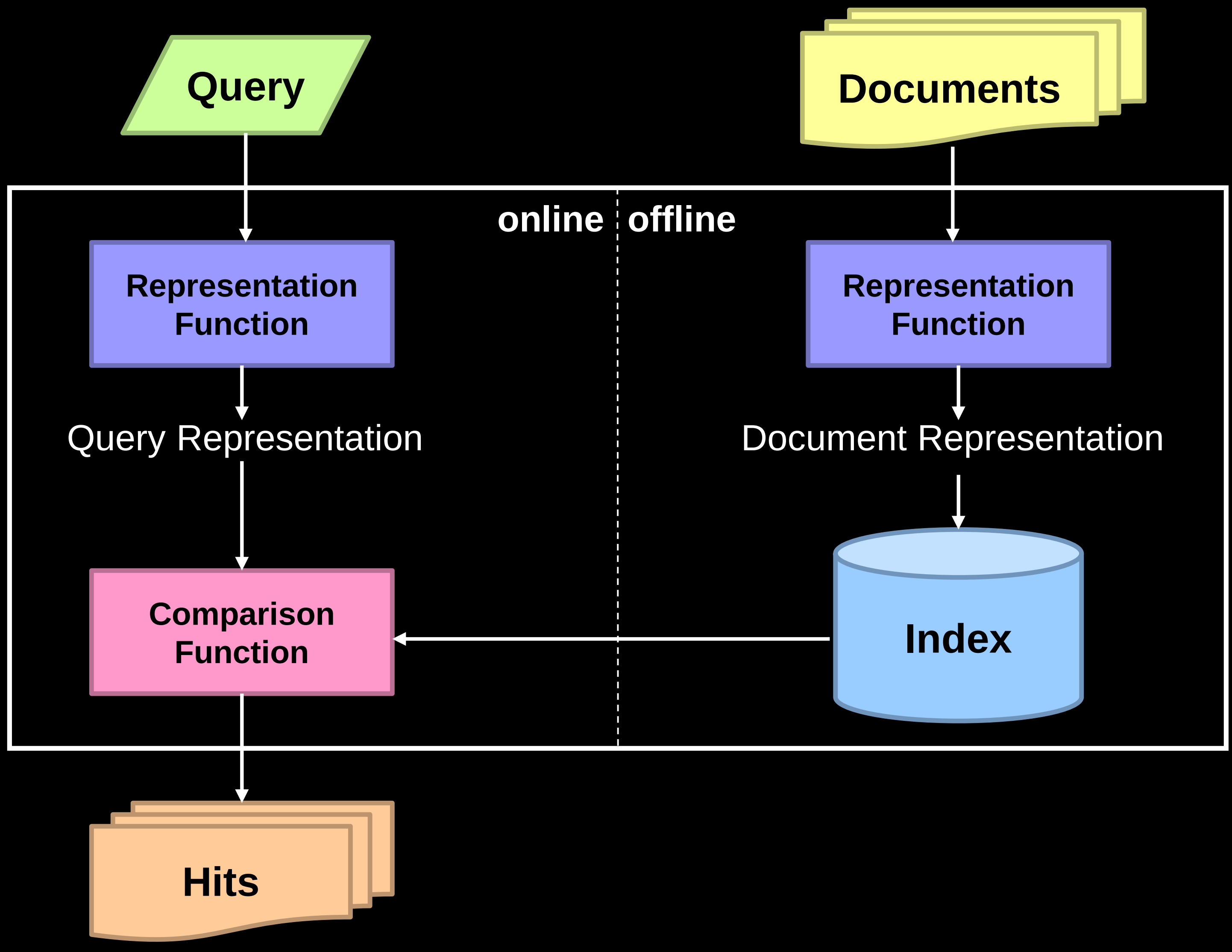
- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit a → { b: count_b, c: count_c, d: count_d … }
- Reducers perform element-wise sum of associative arrays

	a → { b: 1, d: 5, e: 3 }
+	a → { b: 1, c: 2, d: 2, f: 2 }
	a → { b: 2, c: 2, d: 7, e: 3, f: 2 }

“Stripes” Analysis

- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Can make better use of combiners
- Disadvantages
  - More difficult to implement
  - Underlying object is more heavyweight
  - Fundamental limitation in terms of size of event space

Abstract IR Architecture



MapReduce it?

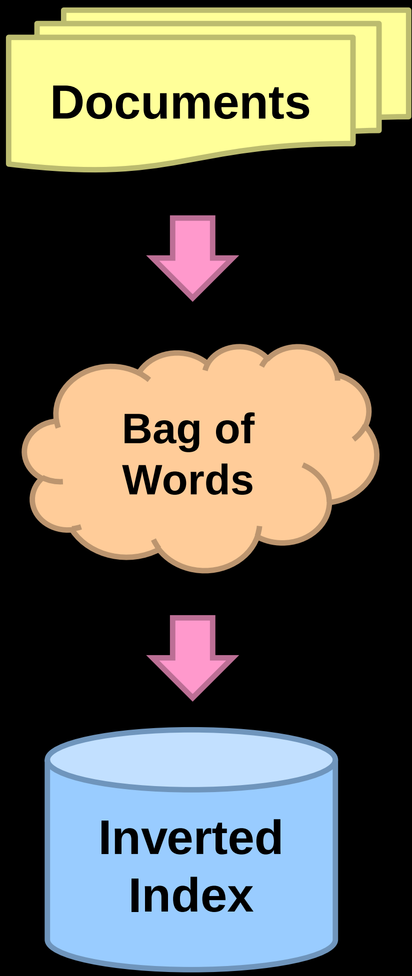
- The indexing problem
  - Scalability is critical
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
  - Incremental updates may or may not be important
  - For the web, crawling is a challenge in itself

Perfect for MapReduce!

- The retrieval problem
  - Must have sub-second response time
  - For the web, only need relatively few results

Uh… not so good…

Counting Words...



case folding, tokenization, stopword removal, stemming

~~syntax~~, ~~semantics~~, ~~word knowledge~~, etc.

Inverted Index: 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			

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

Inverted Index: Ranked 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

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

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

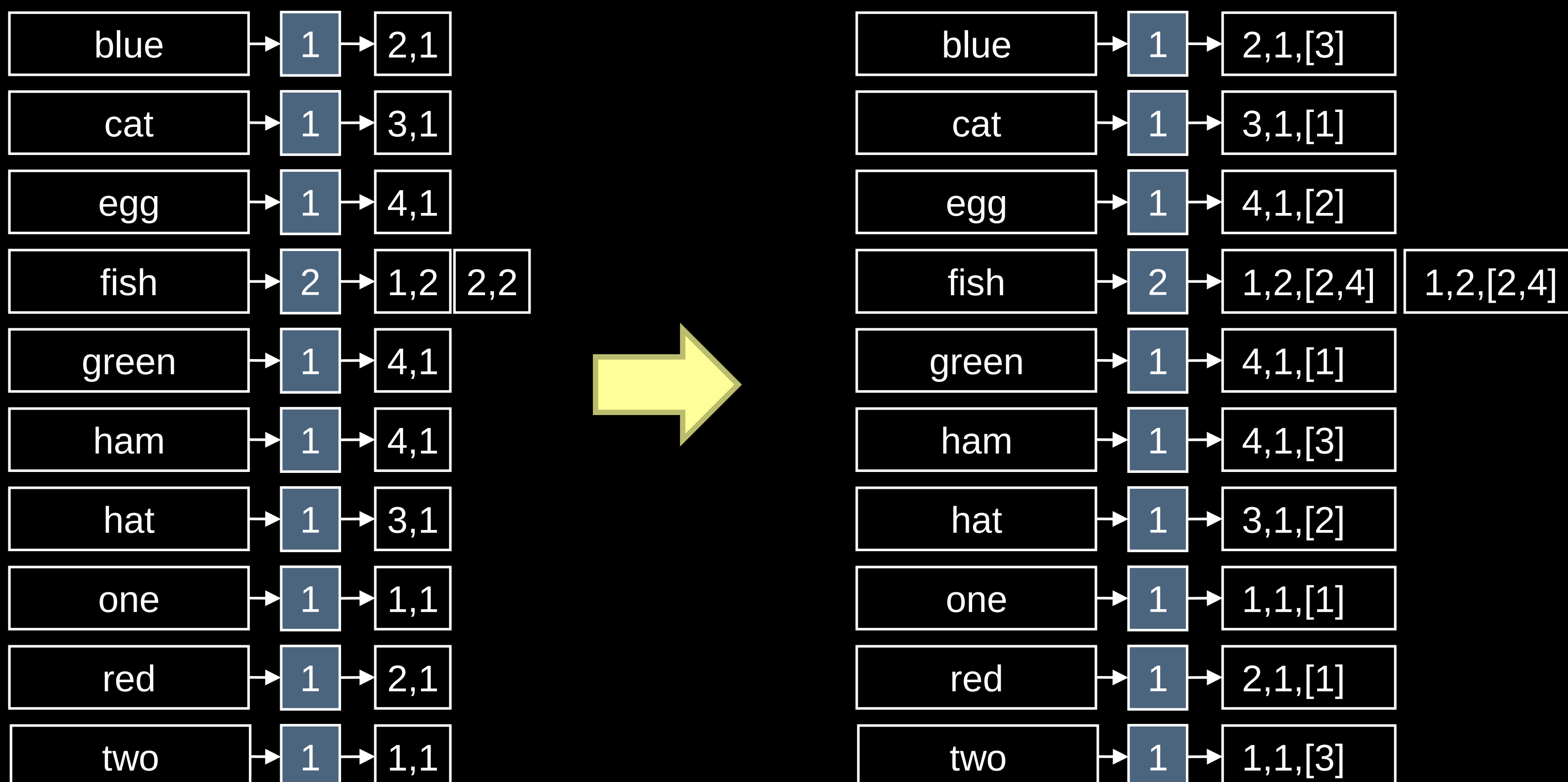
Inverted Index: Positional Information

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham



Indexing: Performance Analysis

- Fundamentally, a large sorting problem
  - Terms usually fit in memory
  - Postings usually don't
- How is it done on a single machine?
- How can it be done with MapReduce?
- First, let's characterize the problem size:
  - Size of vocabulary
  - Size of postings

MapReduce: Index Construction

- Map over all documents
  - Emit *term* as key, (*docno*, *tf)* as value
  - Emit other information as necessary (e.g., term position)
- Sort/shuffle: group postings by term
- Reduce
  - Gather and sort the postings (e.g., by *docno* or *tf*)
  - 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

procedure $\textsc{Map}(a, d)$
  $\textsc{Initialize.AssociativeArray}(H)$
  for all $t \in d$ **do**
    $H\{t\} \leftarrow H\{t\} + 1$
  for all $t \in H$ **do**
    $\textsc{Emit}(t, \langle a, H\{t\}\rangle)$

procedure $\textsc{Reduce}(t, [\langle a_1, f_1\rangle, \langle a_2, f_2\rangle \ldots])$
  $\textsc{Initialize.List}(P)$
  for all $\langle a, f\rangle \in [\langle a_1, f_1\rangle, \langle a_2, f_2\rangle \ldots]$ **do**
    $\textsc{Append}(P, \langle a, f\rangle)$
  $\textsc{Sort}(P)$
  $\textsc{Emit}(t, P)$

Positional Indexes

Map

Doc 1
one fish, two fish

one	1	1	[1]
two	1	1	[3]
fish	1	2	[2,4]

Doc 2
red fish, blue fish

red	2	1	[1]
blue	2	1	[3]
fish	2	2	[2,4]

Doc 3
cat in the hat

cat	3	1	[1]
hat	3	1	[2]

Shuffle and Sort: aggregate values by keys

Reduce

cat	3	1	[1]			
fish	1	2	[2,4]	2	2	[2,4]
one	1	1	[1]			
red	2	1	[1]			

blue	2	1	[3]
hat	3	1	[2]
two	1	1	[3]

Inverted Indexing: Pseudo-Code

procedure $\textsc{Map}(a, d)$
    $\textsc{Initialize.AssociativeArray}(H)$
    for all $t \in d$ **do**
        $H\{t\} \leftarrow H\{t\} + 1$
    for all $t \in H$ **do**
        $\textsc{Emit}(t, \langle a, H\{t\} \rangle)$

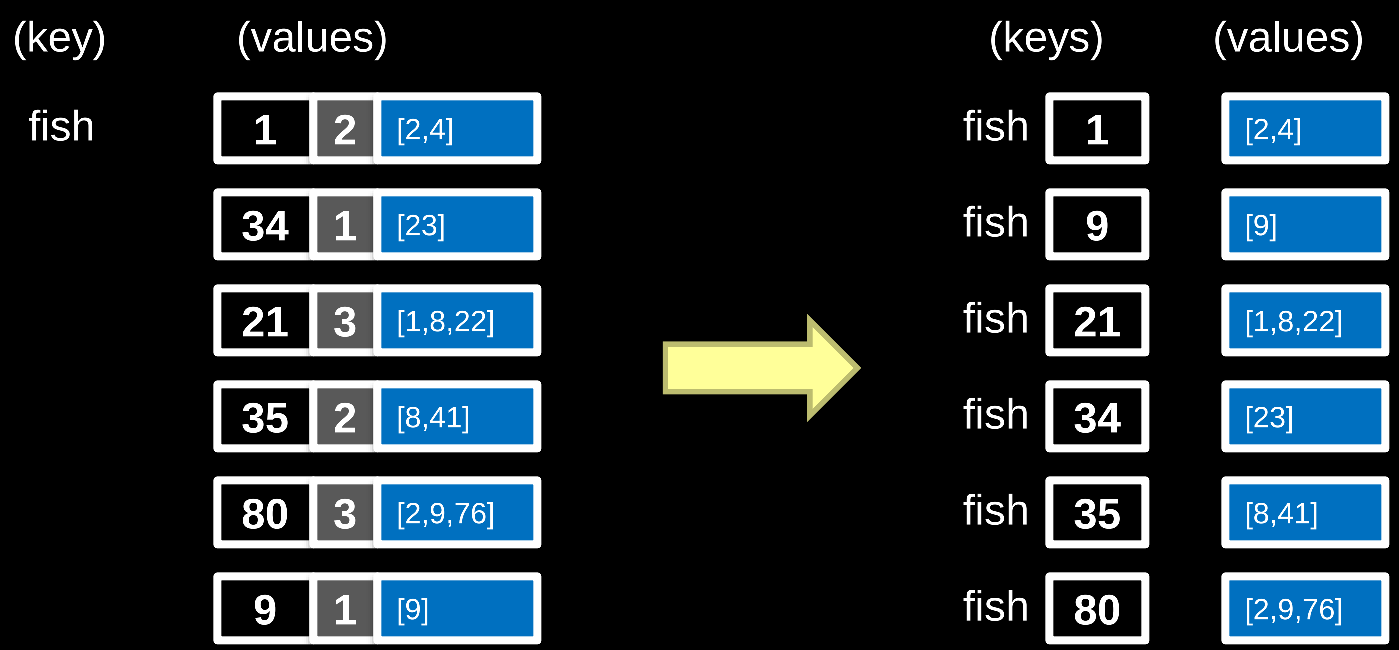
procedure $\textsc{Reduce}(t, [\langle a_1, f_1 \rangle, \langle a_2, f_2 \rangle \ldots])$
    $\textsc{Initialize.List}(P)$
    for all $\langle a, f \rangle \in [\langle a_1, f_1 \rangle, \langle a_2, f_2 \rangle \ldots]$ **do**
        $\textsc{Append}(P, \langle a, f \rangle)$
    $\textsc{Sort}(P)$
    $\textsc{Emit}(t, P)$

What's the problem?

Scalability Bottleneck

- 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?
- Uh oh!

Another Try…



How is this different?

- Let the framework do the sorting
- Term frequency implicitly stored
- Directly write postings to disk!

Wait, there’s more!

(but first, an aside)

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…

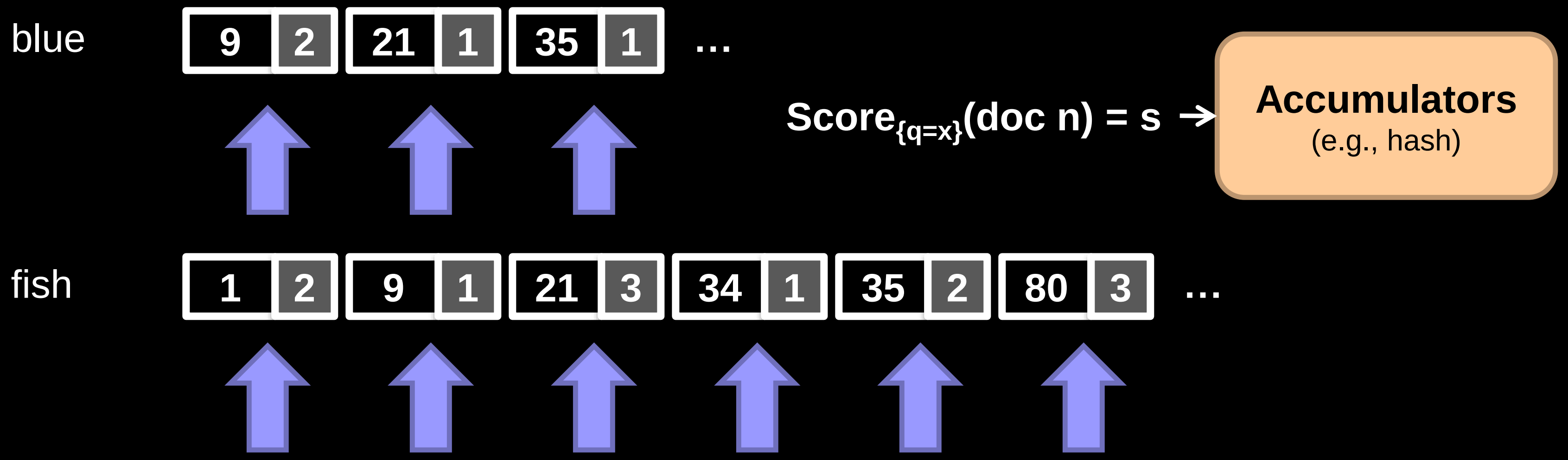
fish	1	2	8	1	12	3	13	1	1	2	45	3	…

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: Query-At-A-Time

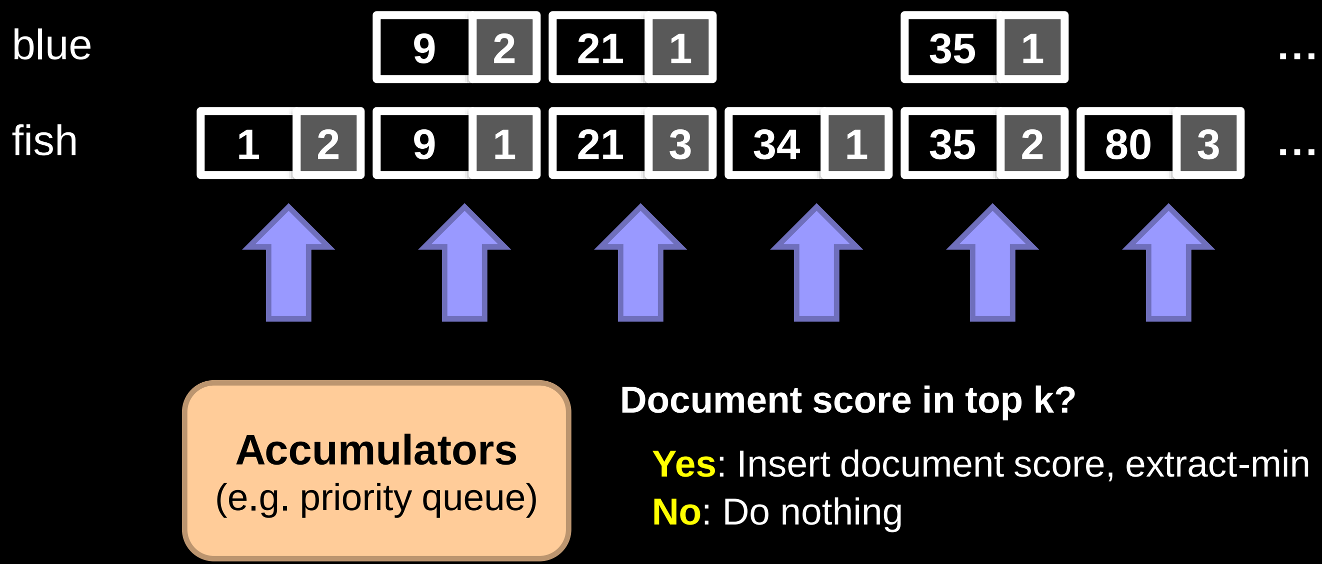
- Evaluate documents one query at a time
  - Usually, starting from most rare term (often with tf-scored postings)



- 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)



Document score in top k?

Yes: Insert document score, extract-min if queue too large

No: Do nothing

- Tradeoffs
  - Small memory footprint (good)
  - Must read through all postings (bad), but skipping possible
  - More disk seeks (bad), but blocking possible

Retrieval with MapReduce?

- MapReduce is fundamentally batch-oriented
  - Optimized for throughput, not latency
  - Startup of mappers and reducers is expensive
- MapReduce is not suitable for real-time queries!
  - Use separate infrastructure for retrieval…

Important Ideas

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

The rest is just details!

When is MapReduce appropriate?

- Lots of input data
  - (e.g., compute statistics over large amounts of text)
  - Take advantage of distributed storage, data locality, aggregate disk throughput
- Lots of intermediate data
  - (e.g., postings)
  - Take advantage of sorting/shuffling, fault tolerance
- Lots of output data
  - (e.g., web crawls)
  - Avoid contention for shared resources
- Relatively little synchronization is necessary

When is MapReduce less appropriate?

- Data fits in memory
- Large amounts of shared data is necessary
- Fine-grained synchronization is needed
- Individual operations are processor-intensive