CMSC 723: Computational Linguistics I — Session #12

MapReduce and Data Intensive NLP

Jimmy Lin and Nitin Madnani
University of Maryland

Wednesday, November 18, 2009

Three Pillars of Statistical NLP

- Algorithms and models
- Features
- Data

Why big data?

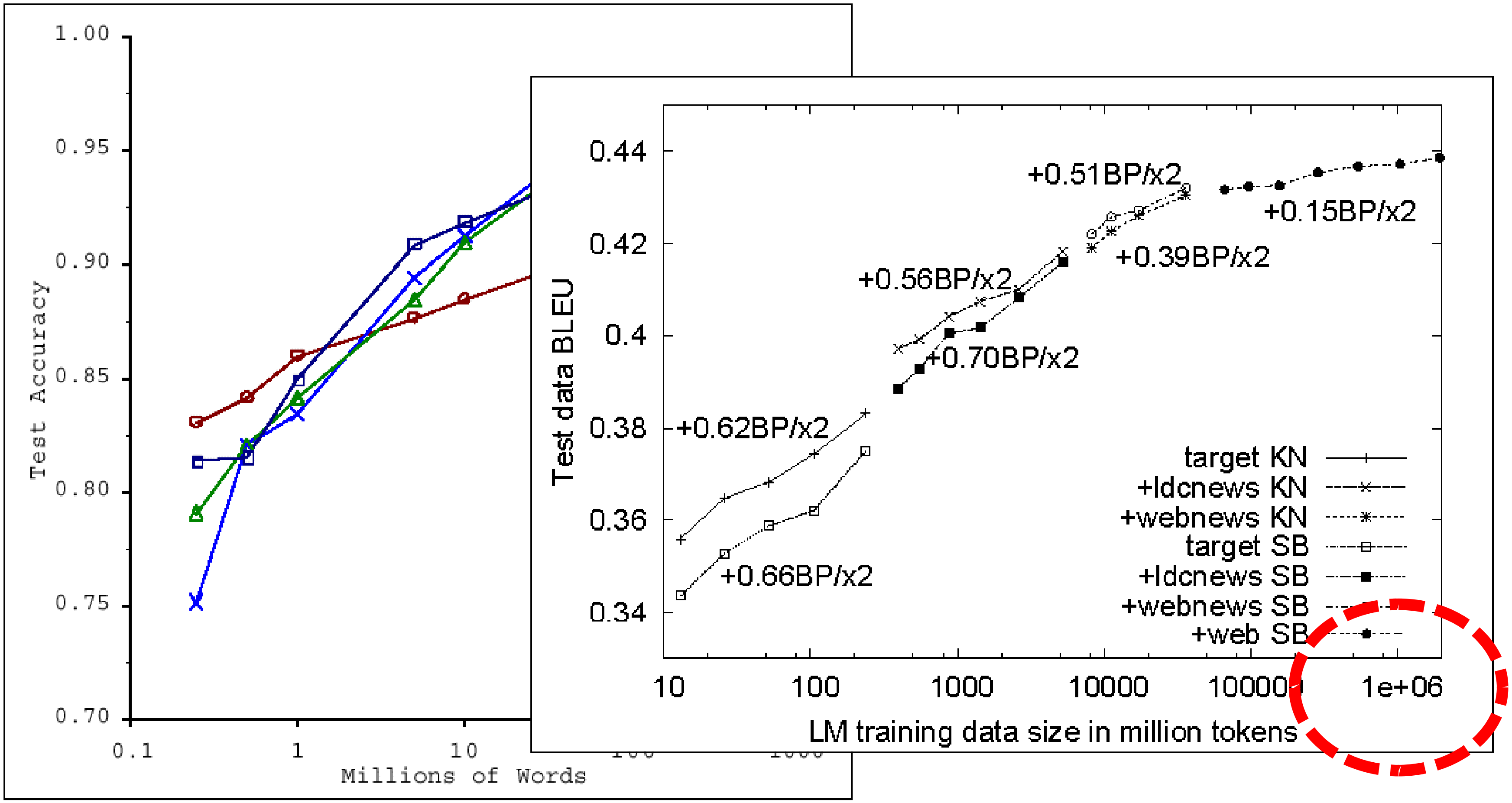
- Fundamental fact of the real world
- Systems improve with more data

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 (??)

No data like more data!

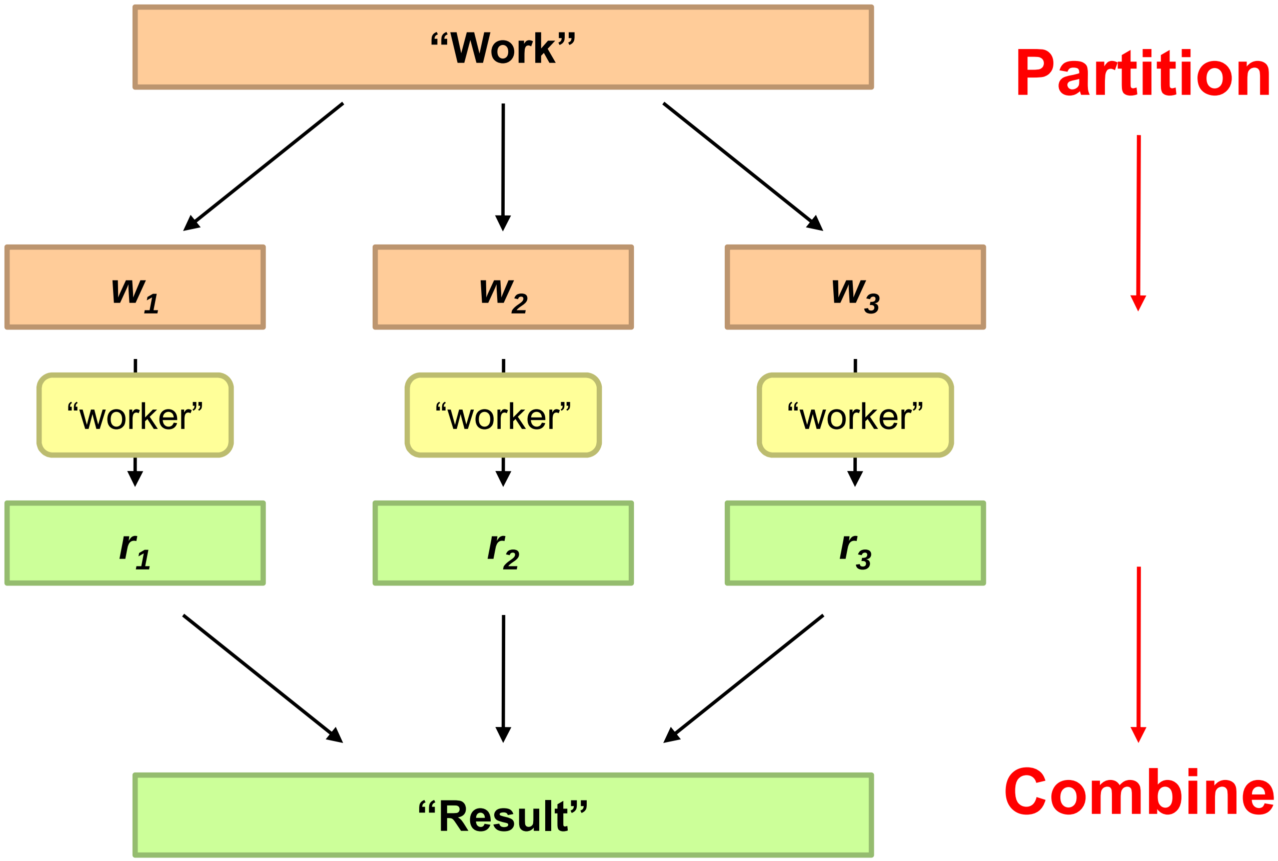
s/knowledge/data/g;



How do we get here if we're not Google?

How do we scale up?

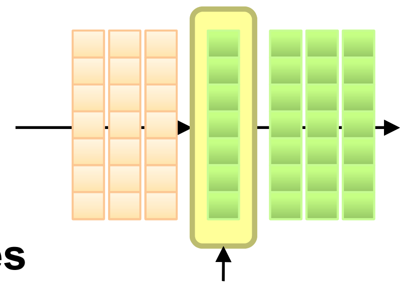
Divide and Conquer



It’s a bit more complex…

Fundamental issues

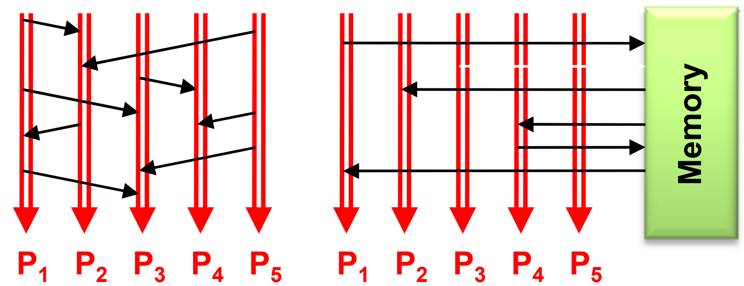
scheduling, data distribution, synchronization, inter-process communication, robustness, fault tolerance, …



Architectural issues

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

Different programming models



Different programming constructs

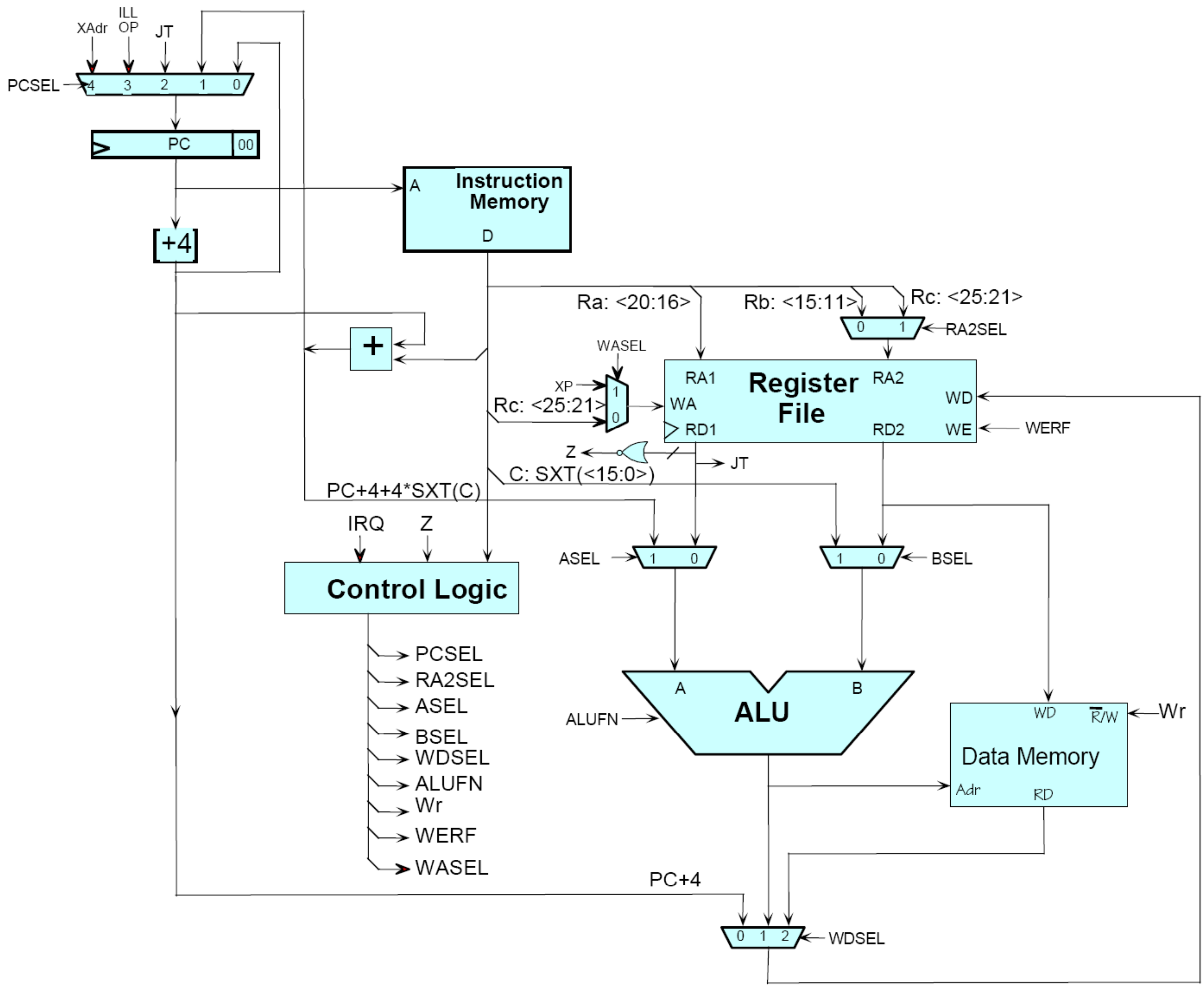
mutexes, conditional variables, barriers, …
masters/slaves, producers/consumers, work queues, …

Common problems

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

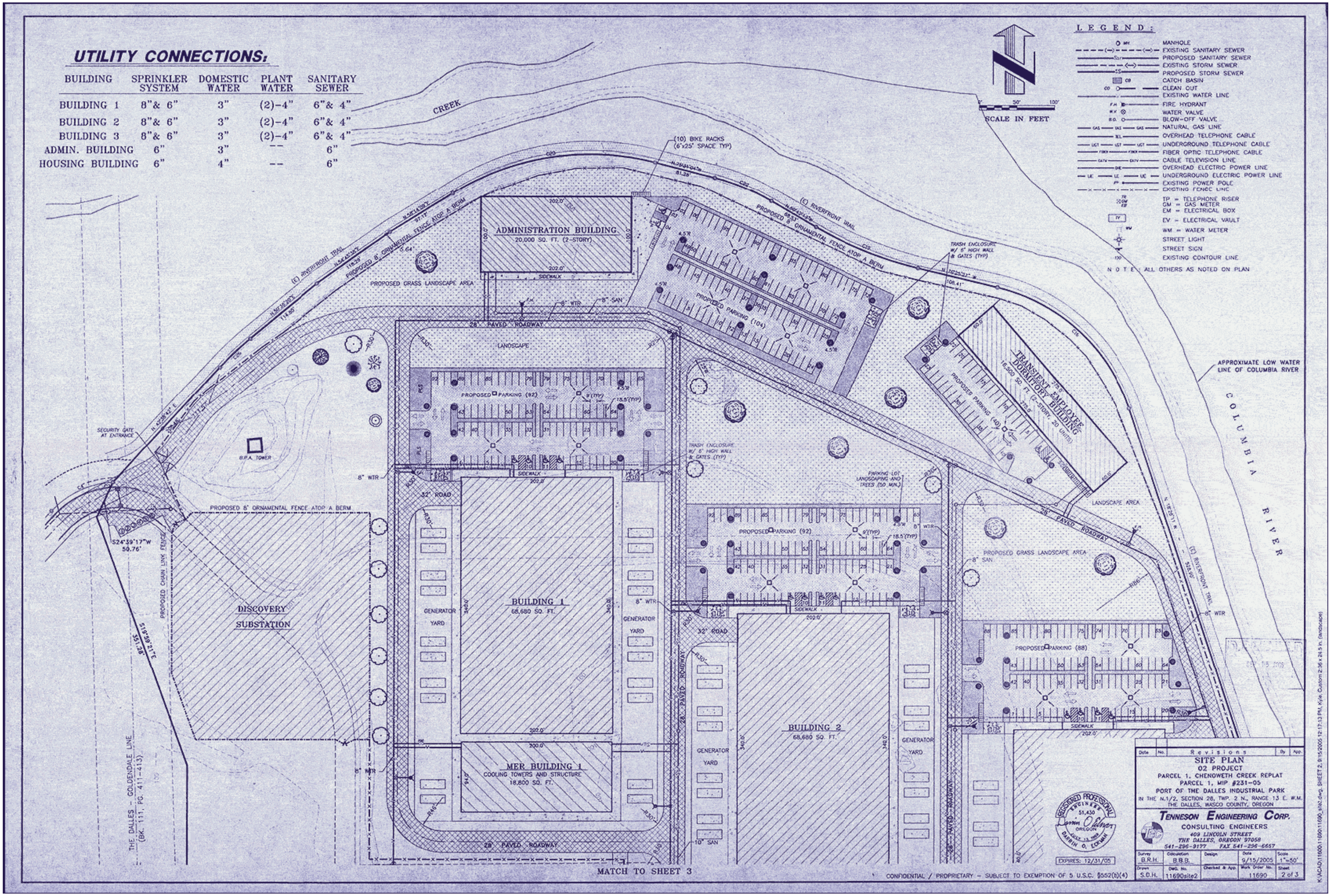
The reality: programmer shoulders the burden of managing concurrency…

Source: Ricardo Guimarães Herrmann



Source: MIT Open Courseware

Source: MIT Open Courseware



Source: Harper's (Feb, 2008)

MapReduce

Typical Large-Data Problem

- Iterate over a large number of records
- Extract something of interest from each

Map

- Shuffle and sort intermediate results
- Aggregate intermediate results

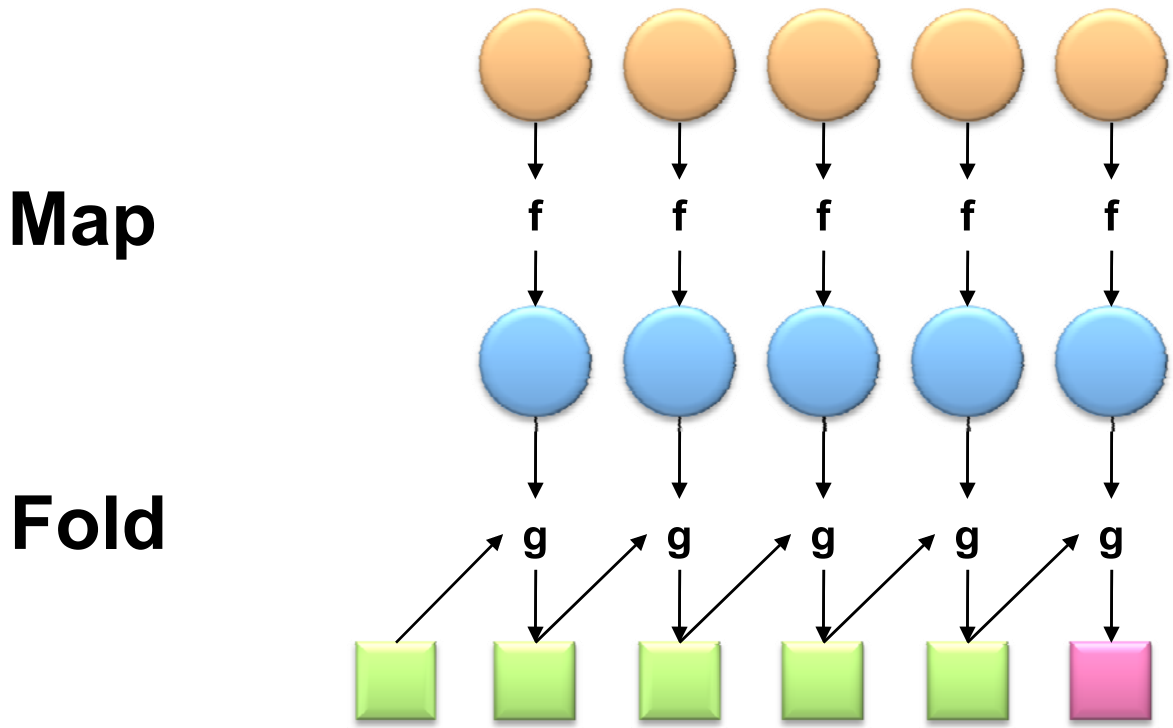
Reduce

- Generate final output

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

Roots in Functional Programming



MapReduce

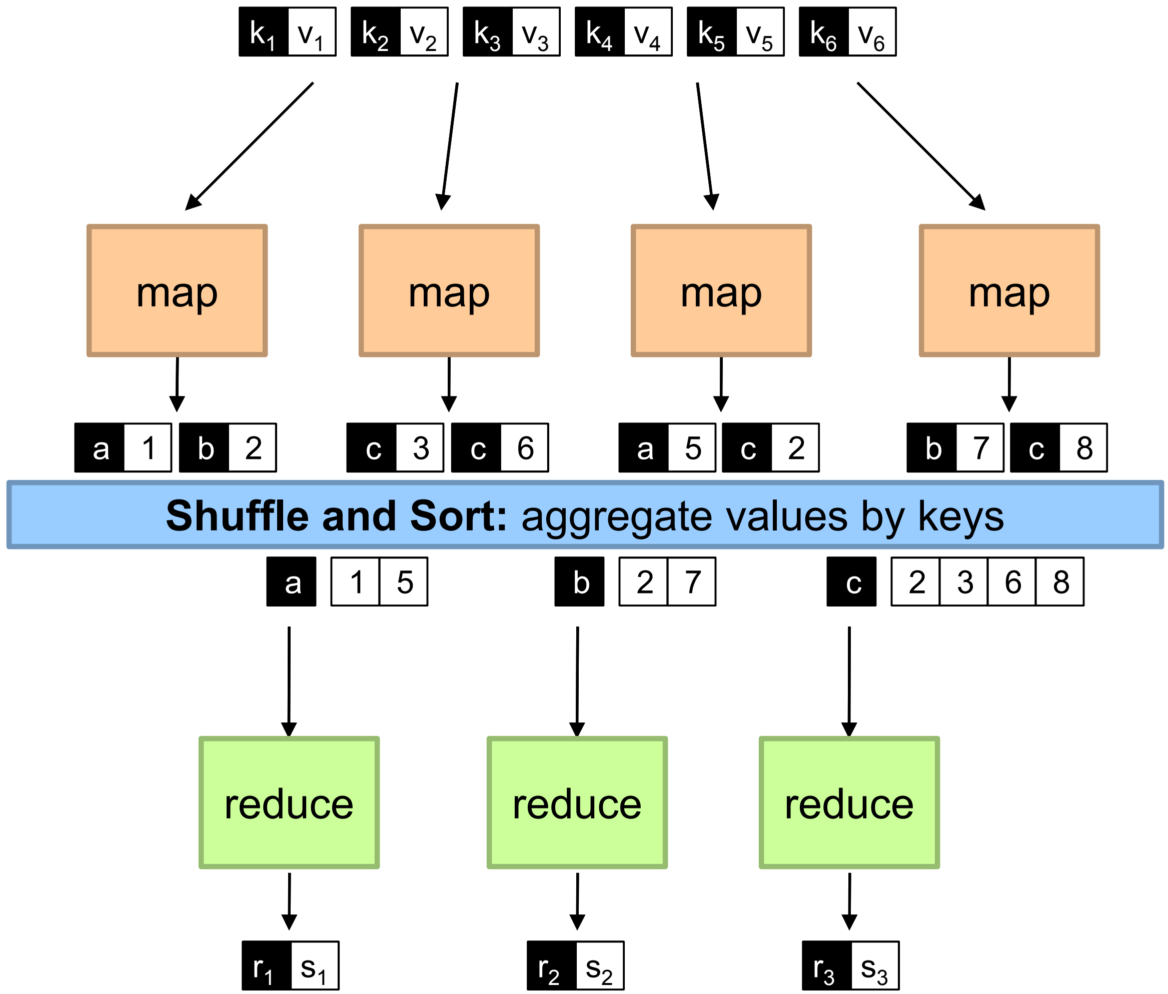
- Programmers specify two functions:

  map (k, v) → <k’, v’>*

  reduce (k’, v’) → <k’, v’>*

  - All values with the same key are reduced together

- The execution framework handles everything else…



k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

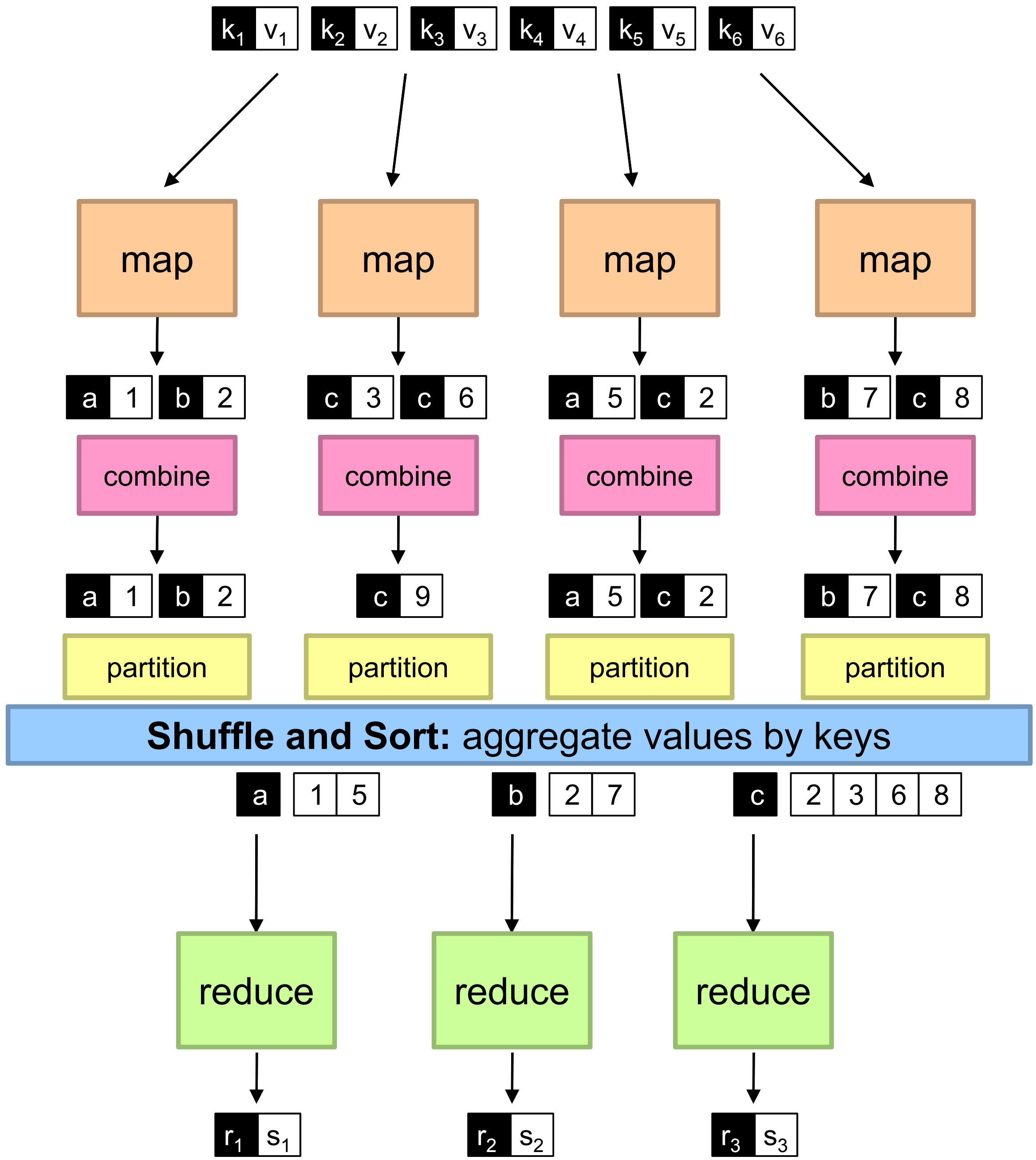
  - All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:

  partition (k', number of partitions) → partition for k'

  - Often a simple hash of the key, e.g., hash(k') mod n
  - Divides up key space for parallel reduce operations

  combine (k', v') → <k', v'>*

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

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 errors and 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);
```

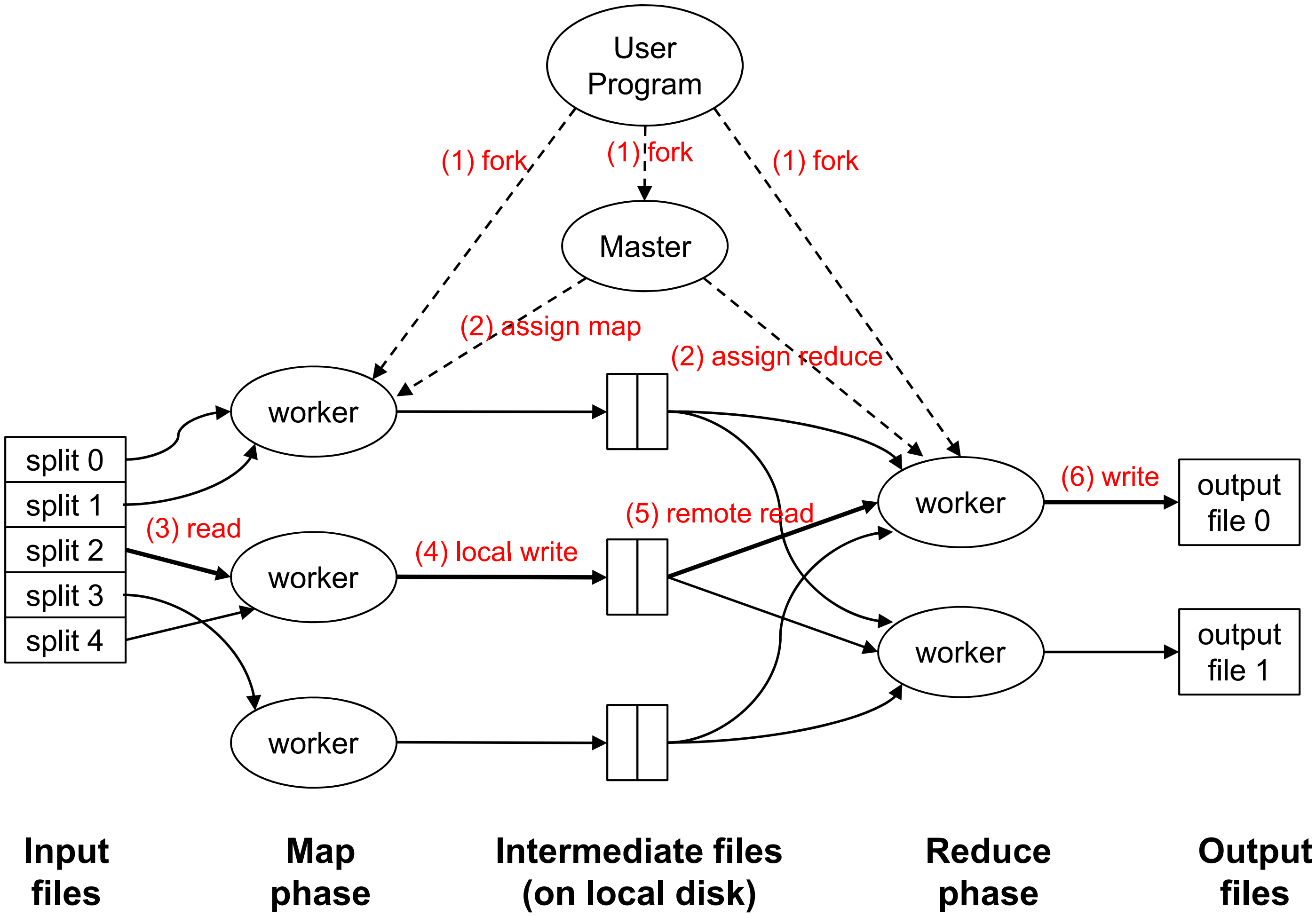
MapReduce can refer to…

- The programming model
- The execution framework (aka "runtime")
- The specific implementation

Usage is usually clear from context!

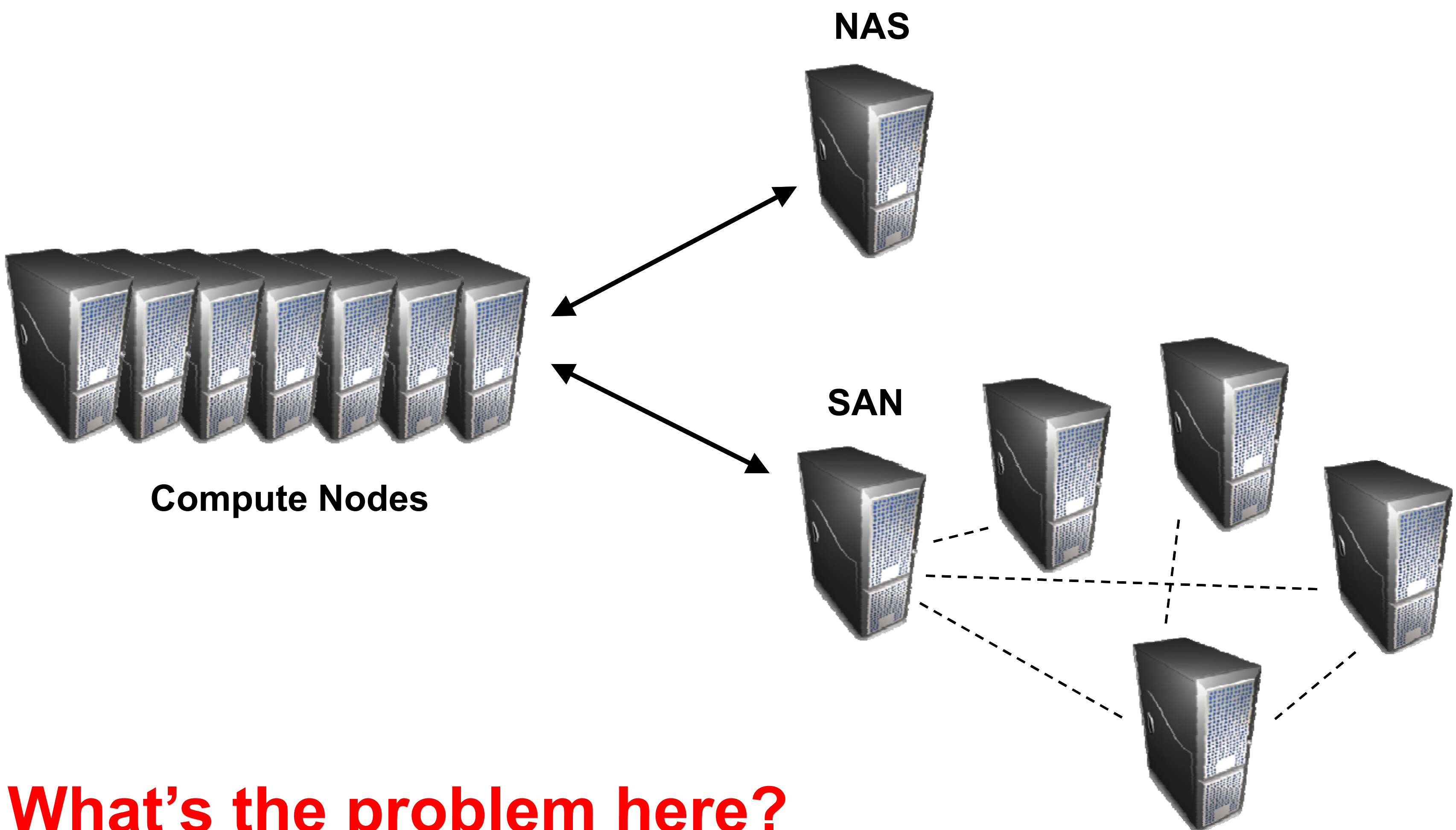
MapReduce Implementations

- 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
- Lots of custom research implementations
  - For GPUs, cell processors, etc.



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

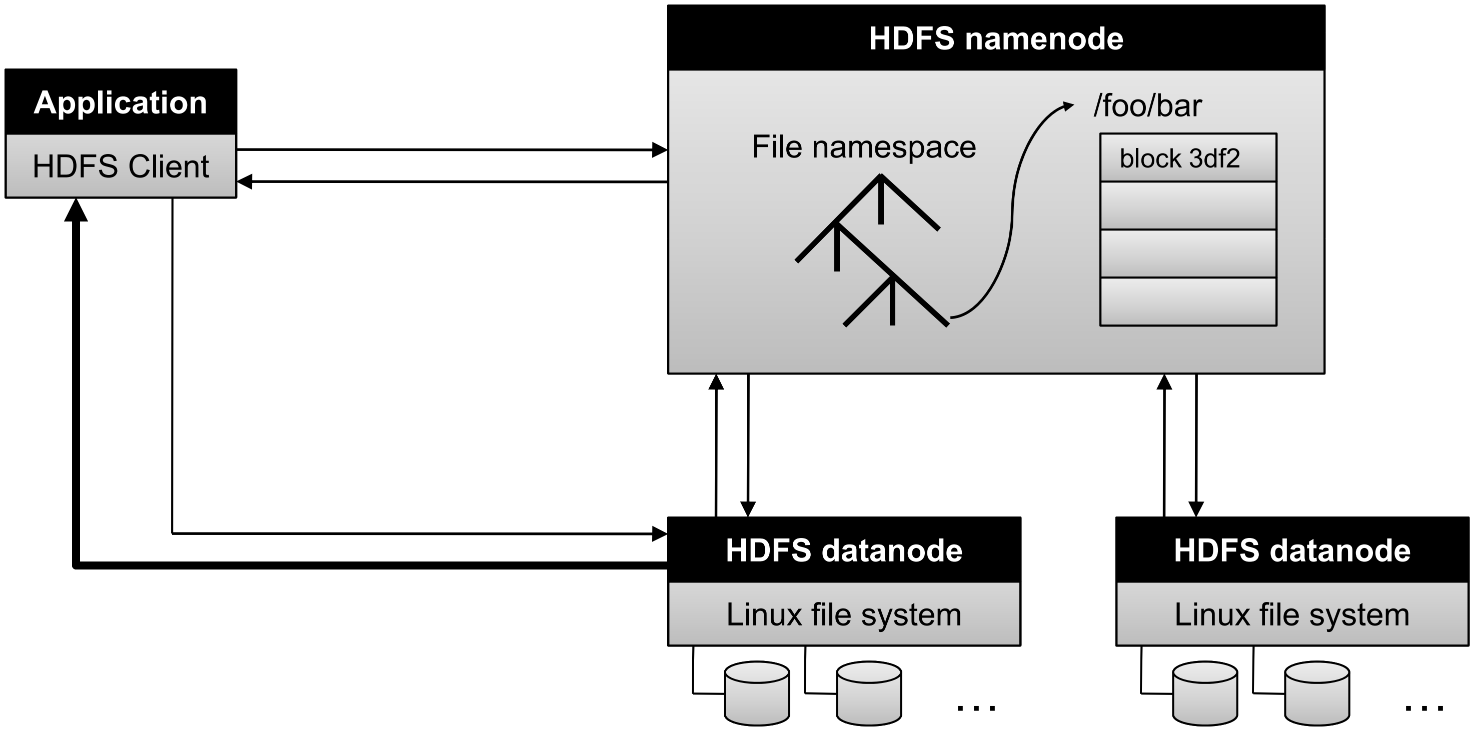
- Commodity hardware over “exotic” hardware
  - Scale out, not up
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of huge files
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
- High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

HDFS = GFS clone (same basic ideas)

HDFS Architecture



Adapted from (Ghemawat et al., SOSP 2003)

Master's Responsibilities

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

MapReduce Algorithm Design

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)

Motivating Example

- Term co-occurrence matrix for a text collection
  - M = N x N matrix (N = vocabulary size)
  - M_{ij}: number of times *i* and *j* co-occur in some context
    (for concreteness, let's say context = sentence)
- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks

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

  $(a, b) \rightarrow 1$
  $(a, c) \rightarrow 2$
  $(a, d) \rightarrow 5$
  $(a, e) \rightarrow 3$
  $(a, f) \rightarrow 2$

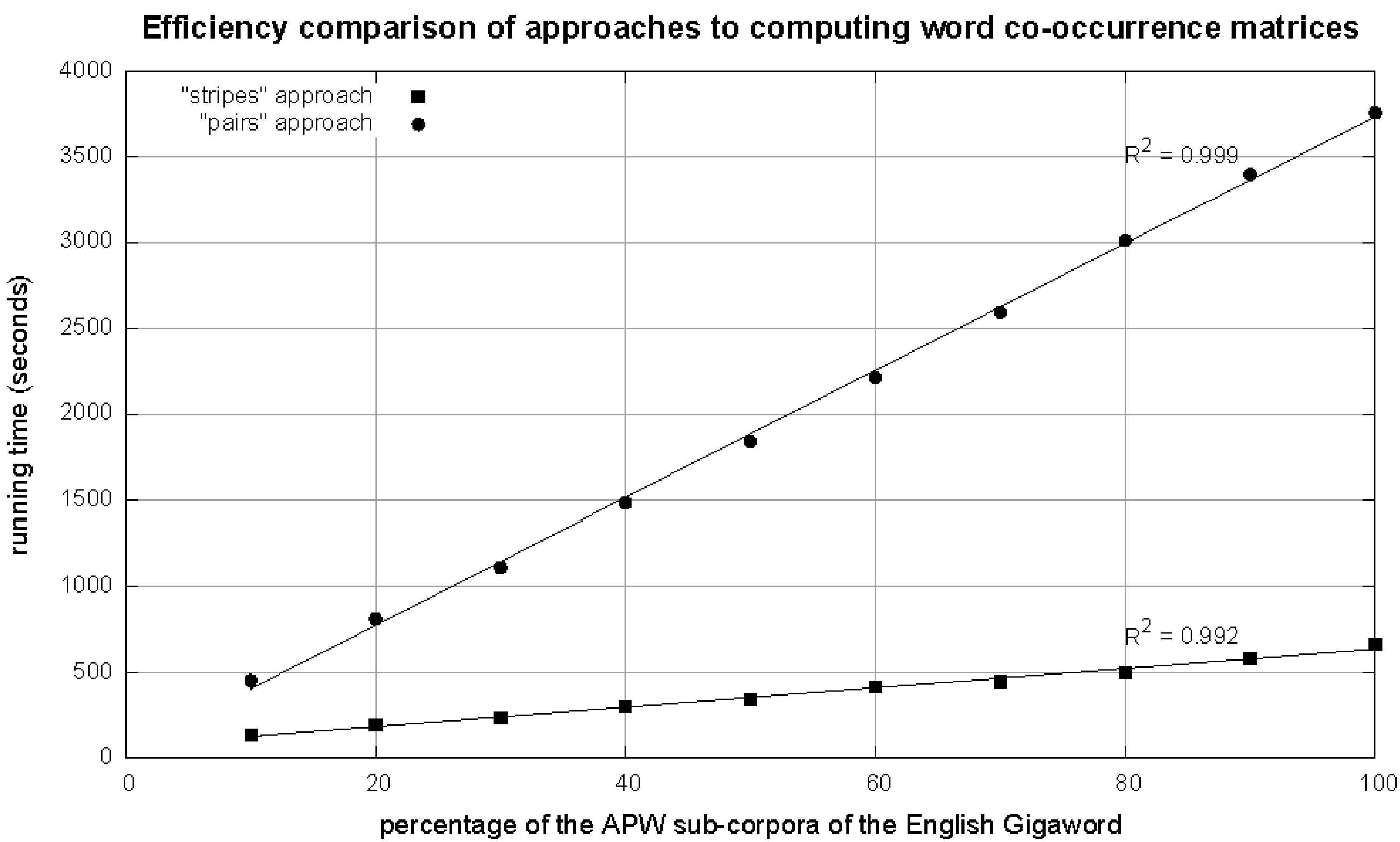
  $a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$

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

$$
\begin{array}{rl}
 & a \rightarrow \{ b: 1, d: 5, e: 3 \} \\
+ & a \rightarrow \{ b: 1, c: 2, d: 2, f: 2 \} \\
\hline
 & a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \}
\end{array}
$$

“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



Cluster size: 38 cores
Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)

Conditional Probabilities

- How do we estimate conditional probabilities from counts?

$$P(B \mid A) = \frac{\text{count}(A,B)}{\text{count}(A)} = \frac{\text{count}(A,B)}{\sum_{B'} \text{count}(A,B')}$$

- Why do we want to do this?
- How do we do this with MapReduce?

P(B|A): "Stripes"

$a \rightarrow \{b_1{:}3,\ b_2{:}12,\ b_3{:}7,\ b_4{:}1,\ \ldots\}$

- Easy!
  - One pass to compute (a, *)
  - Another pass to directly compute P(B|A)

P(B|A): “Pairs”

$(a, *) \rightarrow 32$ **Reducer holds this value in memory**

$(a, b_1) \rightarrow 3$		$(a, b_1) \rightarrow 3 / 32$
$(a, b_2) \rightarrow 12$	⇨	$(a, b_2) \rightarrow 12 / 32$
$(a, b_3) \rightarrow 7$		$(a, b_3) \rightarrow 7 / 32$
$(a, b_4) \rightarrow 1$		$(a, b_4) \rightarrow 1 / 32$
…		…

- For this to work:
  - Must emit extra (a, *) for every b_n in mapper
  - Must make sure all a’s get sent to same reducer (use partitioner)
  - Must make sure (a, *) comes first (define sort order)
  - Must hold state in reducer across different key-value pairs

Synchronization in Hadoop

- Approach 1: turn synchronization into an ordering problem
  - Sort keys into correct order of computation
  - Partition key space so that each reducer gets the appropriate set of partial results
  - Hold state in reducer across multiple key-value pairs to perform computation
  - Illustrated by the “pairs” approach
- Approach 2: construct data structures that “bring the pieces together”
  - Each reducer receives all the data it needs to complete the computation
  - Illustrated by the “stripes” approach

Issues and Tradeoffs

- Number of key-value pairs
  - Object creation overhead
  - Time for sorting and shuffling pairs across the network
- Size of each key-value pair
  - De/serialization overhead
- Combiners make a big difference!
  - RAM vs. disk vs. network
  - Arrange data to maximize opportunities to aggregate partial results

Case Study: LMs with MR

Language Modeling Recap

- **Interpolation**: Consult *all* models at the same time to compute an interpolated probability estimate.
- **Backoff**: Consult the highest order model first and backoff to lower order model *only if* there are no higher order counts.
- **Interpolated Kneser Ney** (state-of-the-art)
  - Use absolute discounting to save some probability mass for lower order models.
  - Use a novel form of lower order models (count *unique* single word contexts instead of occurrences)
  - Combine models into a true probability model using interpolation

$$P_{KN}(w_3|w_1, w_2) = \frac{C_{KN}(w_1 w_2 w_3) - D}{C_{KN}(w_1 w_2)} + \lambda(w_1 w_2) P_{KN}(w_3|w_2)$$

Questions for today

Can we efficiently train an IKN LM with terabytes of data?

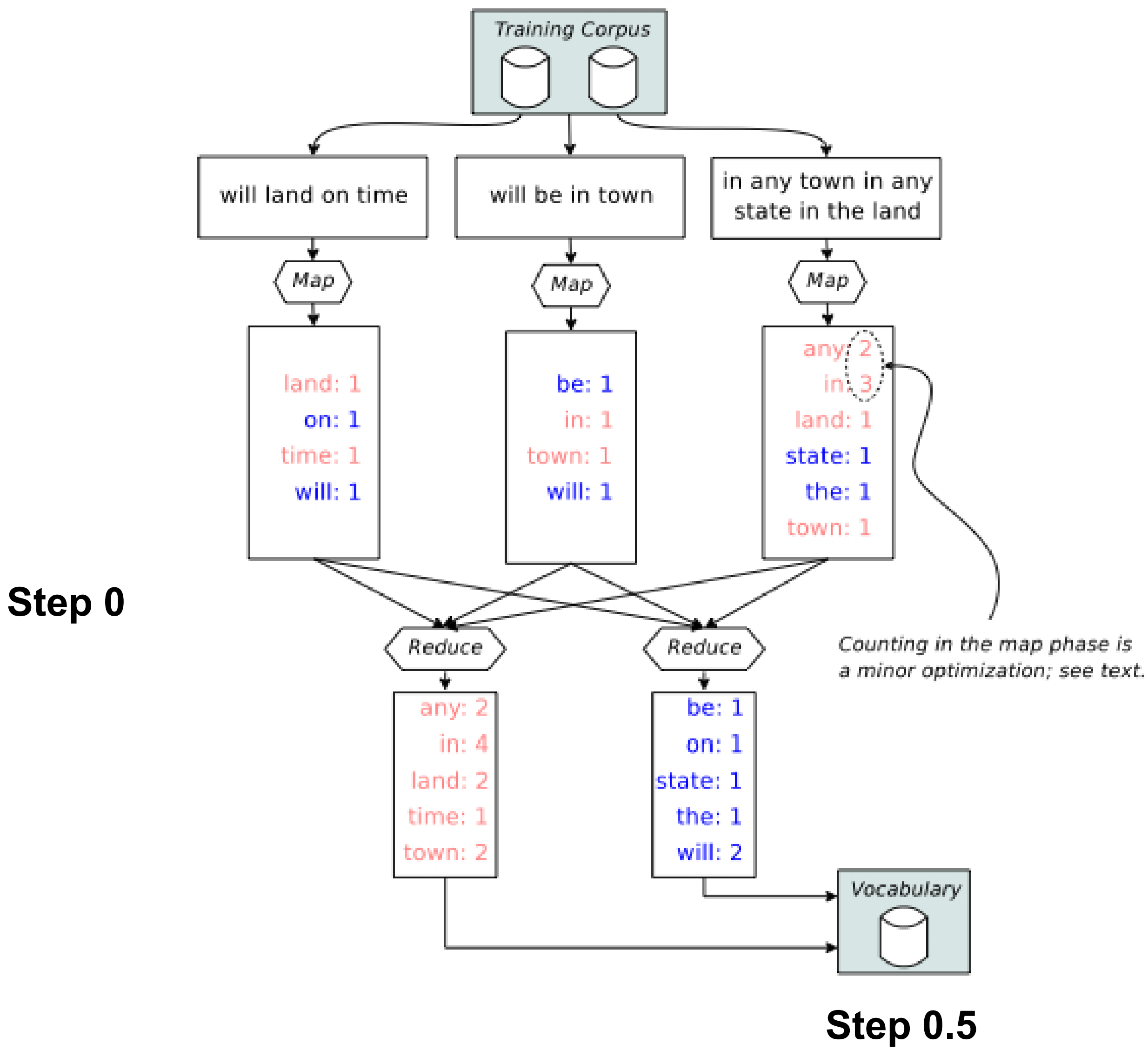
Does it really matter?

Using MapReduce to Train IKN

- Step 0: Count words [MR]
- Step 0.5: Assign IDs to words [vocabulary generation]
  (more frequent → smaller IDs)
- Step 1: Compute *n*-gram counts [MR]
- Step 2: Compute lower order context counts [MR]
- Step 3: Compute unsmoothed probabilities and interpolation weights [MR]
- Step 4: Compute interpolated probabilities [MR]

[MR] = MapReduce job

Steps 0 & 0.5


Training Corpus
will land on time
will be in town
in any town in any state in the land
Map
Map
Map
land: 1
on: 1
time: 1
will: 1
be: 1
in: 1
town: 1
will: 1
any: 2
in: 3
land: 1
state: 1
the: 1
town: 1
Reduce
Reduce
Counting in the map phase is a minor optimization; see text.
any: 2
in: 4
land: 2
time: 1
town: 2
be: 1
on: 1
state: 1
the: 1
will: 2
Vocabulary
Step 0
Step 0.5

Steps 1-4

		Step 1	Step 2	Step 3	Step 4
Mapper Input	Input Key	DocID	*n*-grams "a b c"	"a b c"	"a b"
	Input Value	Document	C_{total}("a b c")	C_{KN}("a b c")	_Step 3 Output_
Mapper Output Reducer Input	Intermediate Key	*n*-grams "a b c"	"a b c"	"a b" (history)	"c b a"
	Intermediate Value	C_{doc}("a b c")	C'_{KN}("a b c")	("c", C_{KN}("a b c"))	(P'("a b c"), λ("a b"))
	Partitioning	"a b c"	"a b c"	"a b"	"c b"
Reducer Output	Output Value	C_{total}("a b c")	C_{KN}("a b c")	("c", P'("a b c"), λ("a b"))	(P_{KN}("a b c"), λ("a b"))
		Count n-grams	**Count contexts**	**Compute unsmoothed probs AND interp. weights**	**Compute Interp. probs**

All output keys are always the *same* as the intermediate keys
I only show trigrams here but the steps operate on bigrams and unigrams as well

Steps 1-4

		Step 1	Step 2	Step 3	Step 4
Mapper Input	Input Key	DocID	*n*-grams "a b c"	"a b c"	"a b"
	Input Value	Document	C_{total}("a b c")	C_{KN}("a b c")	_Step 3 Output_
Mapper Output Reducer Input					
	Intermediate Key				"c b a"
	Intermediate Value				"a b c"), λ("a b"))
	Partitioning				"c b"
Reducer Output	Output Value	C_{total}("a b c")	C_{KN}("a b c")	("c", P'("a b c"), λ("a b"))	(P_{KN}("a b c"), λ("a b"))
		Count n-grams	**Count contexts**	**Compute unsmoothed probs AND interp. weights**	**Compute Interp. probs**

Details are not important!

5 MR jobs to train IKN (expensive)!

IKN LMs are big!
(interpolation weights are context dependent)

Can we do something that has better behavior at scale in terms of time and space?

All output keys are always the *same* as the intermediate keys
I only show trigrams here but the steps operate on bigrams and unigrams as well

Let's try something stupid!

- Simplify backoff as much as possible!
- Forget about trying to make the LM be a true probability distribution!
- Don't do any discounting of higher order models!
- Have a single backoff weight independent of context!
  [α(•) = α]

$$S(w_3|w_2, w_1) = \frac{c(w_1 w_2 w_3)}{c(w_1 w_2)} \quad \text{if } c(w_1 w_2 w_3) > 0$$

$$= \alpha S(w_3|w_2) \quad \text{otherwise}$$

$$S(w_3) = \frac{c(w_3)}{N} \quad \text{(recursion ends at unigrams)}$$

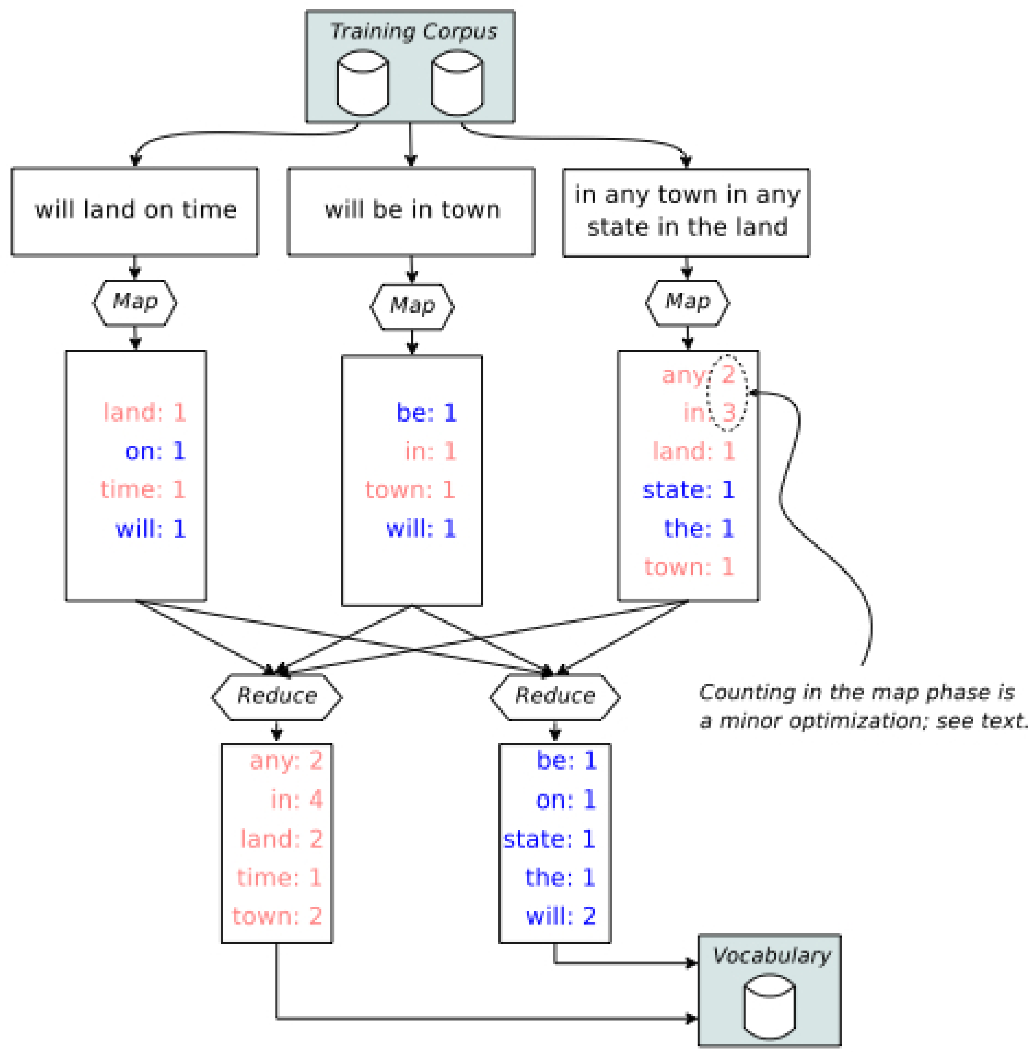
"Stupid Backoff (SB)"

Using MapReduce to Train SB

- Step 0: Count words [MR]
- Step 0.5: Assign IDs to words [vocabulary generation]
  (more frequent → smaller IDs)
- Step 1: Compute *n*-gram counts [MR]
- Step 2: Generate final LM "scores" [MR]

[MR] = MapReduce job

Steps 0 & 0.5



Step 0

Step 0.5

Steps 1 & 2

		Step 1	Step 2
Mapper Input	Input Key	DocID	*n*-grams "a b c"
	Input Value	Document	C_{total}("a b c")
Mapper Output Reducer Input	Intermediate Key	*n*-grams "a b c"	"a b c"
	Intermediate Value	C_{doc}("a b c")	S("a b c")
	Partitioning	first two words "a b"	last two words "b c"
Reducer Output	Output Value	C_{total}("a b c")	S("a b c") [write to disk]
		Count n-grams	**Compute LM scores**

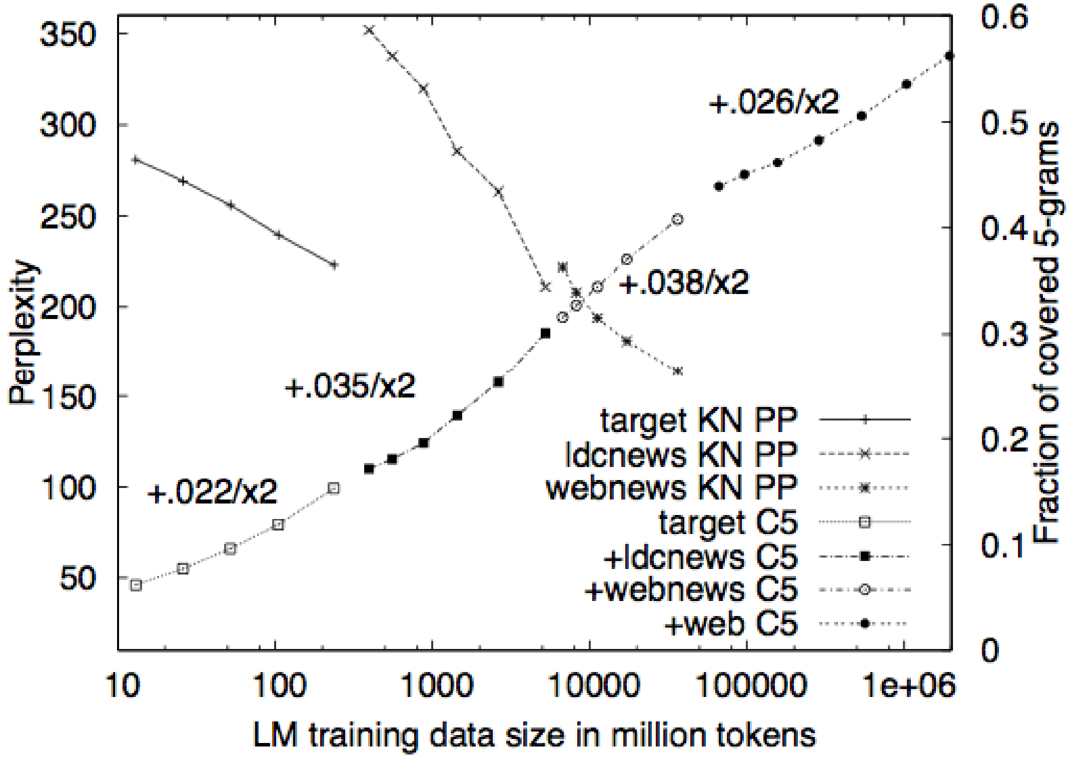
The clever partitioning in Step 2 is the key to efficient use at runtime!

Which one wins?

	target	*webnews*	*web*
# tokens	237M	31G	1.8T
vocab size	200k	5M	16M
# n-grams	257M	21G	300G
LM size (SB)	2G	89G	1.8T
time (SB)	20 min	8 hours	1 day
time (KN)	2.5 hours	2 days	–
# machines	100	400	1500

Table 2: Sizes and approximate training times for 3 language models with Stupid Backoff (SB) and Kneser-Ney Smoothing (KN).

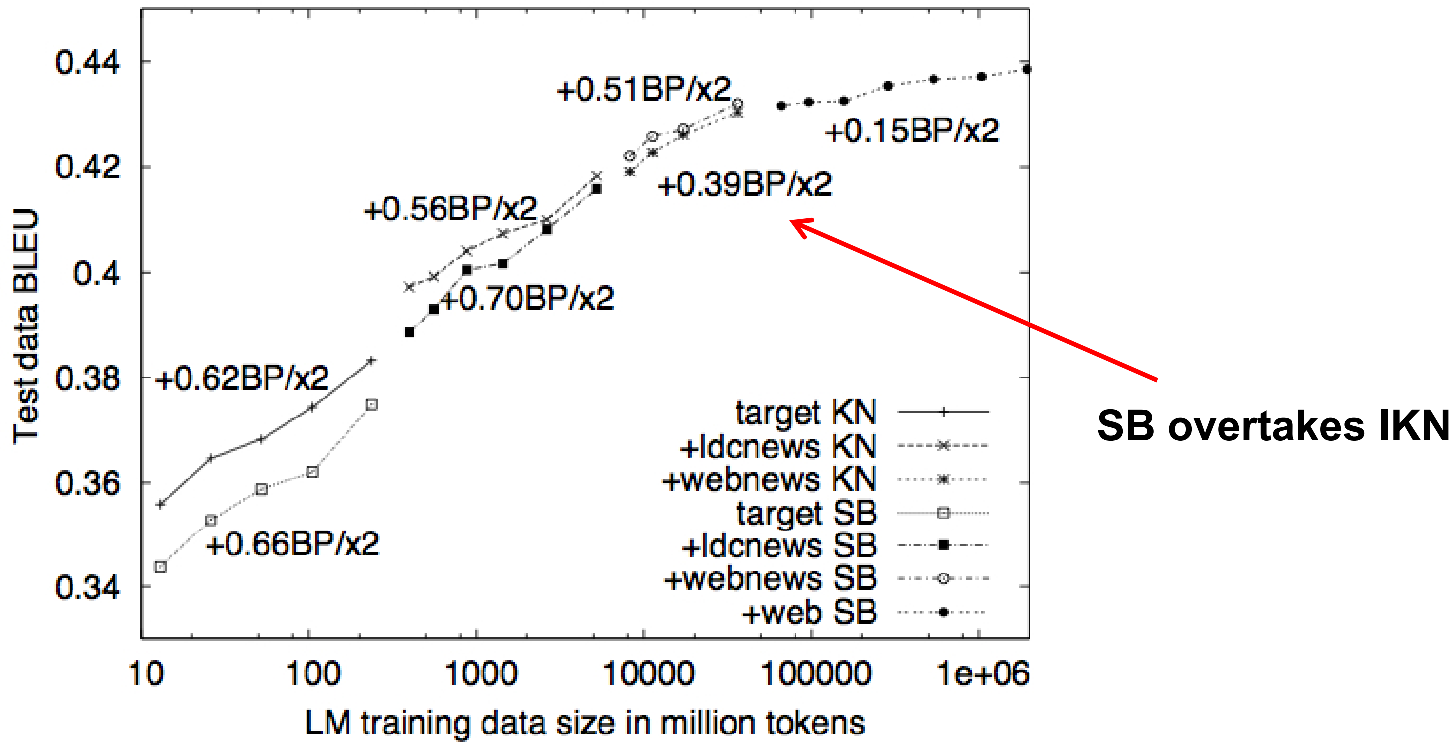
Which one wins?



Can't compute perplexity for SB. Why?

Why do we care about 5-gram coverage for a test set?

Which one wins?



BLEU is a measure of MT performance.

Not as stupid as you thought, huh?

Take away

- The MapReduce paradigm and infrastructure make it simple to scale algorithms to web scale data
- At Terabyte scale, efficiency becomes really important!
- When you have a lot of data, a more scalable technique (in terms of speed and memory consumption) can do better than the state-of-the-art even if it's stupider!

"The difference between genius and stupidity is that genius has its limits."
- Oscar Wilde

"The dumb shall inherit the cluster"
- Nitin Madnani

Back to the Beginning…

- Algorithms and models
- Features
- Data