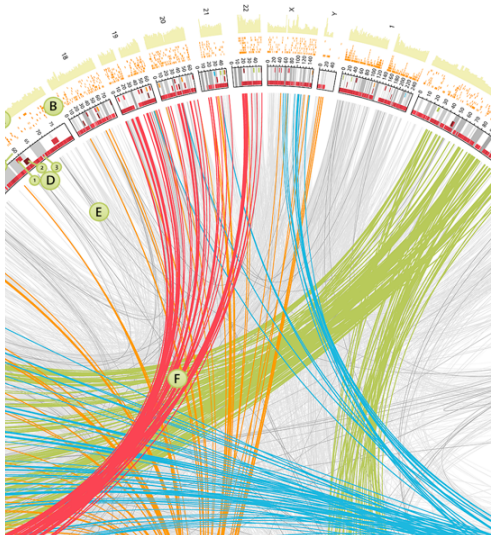
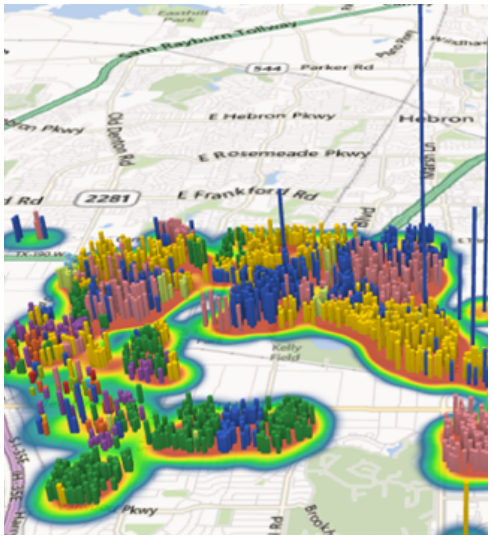
Big Data Analytics — Basic algorithm design





Basic Hadoop API

Mapper and *Reducer*

- Defined in package `org.apache.hadoop.mapreduce`
- Mapper
  - `void setup(Mapper.Context context)`
    Called once at the beginning of the task
  - `void map(K key, V value, Mapper.Context context)`
    Called once for each key/value pair in the input split
  - `void cleanup(Mapper.Context context)`
    Called once at the end of the task

```
Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
  Mapper()
  setup(Context) : void
  map(KEYIN, VALUEIN, Context) : void
  cleanup(Context) : void
  run(Context) : void
  Context
```

- Reducer/Combiner
  - `void setup(Reducer.Context context)`
    Called once at the start of the task
  - `void reduce(K key, Iterable<V> values, Reducer.Context context)`
    Called once for each key
  - `void cleanup(Reducer.Context context)`
    Called once at the end of the task

```
Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
  Reducer()
  setup(Context) : void
  reduce(KEYIN, Iterable<VALUEIN>, Context) : void
  cleanup(Context) : void
  run(Context) : void
  Context
```

Basic Hadoop API

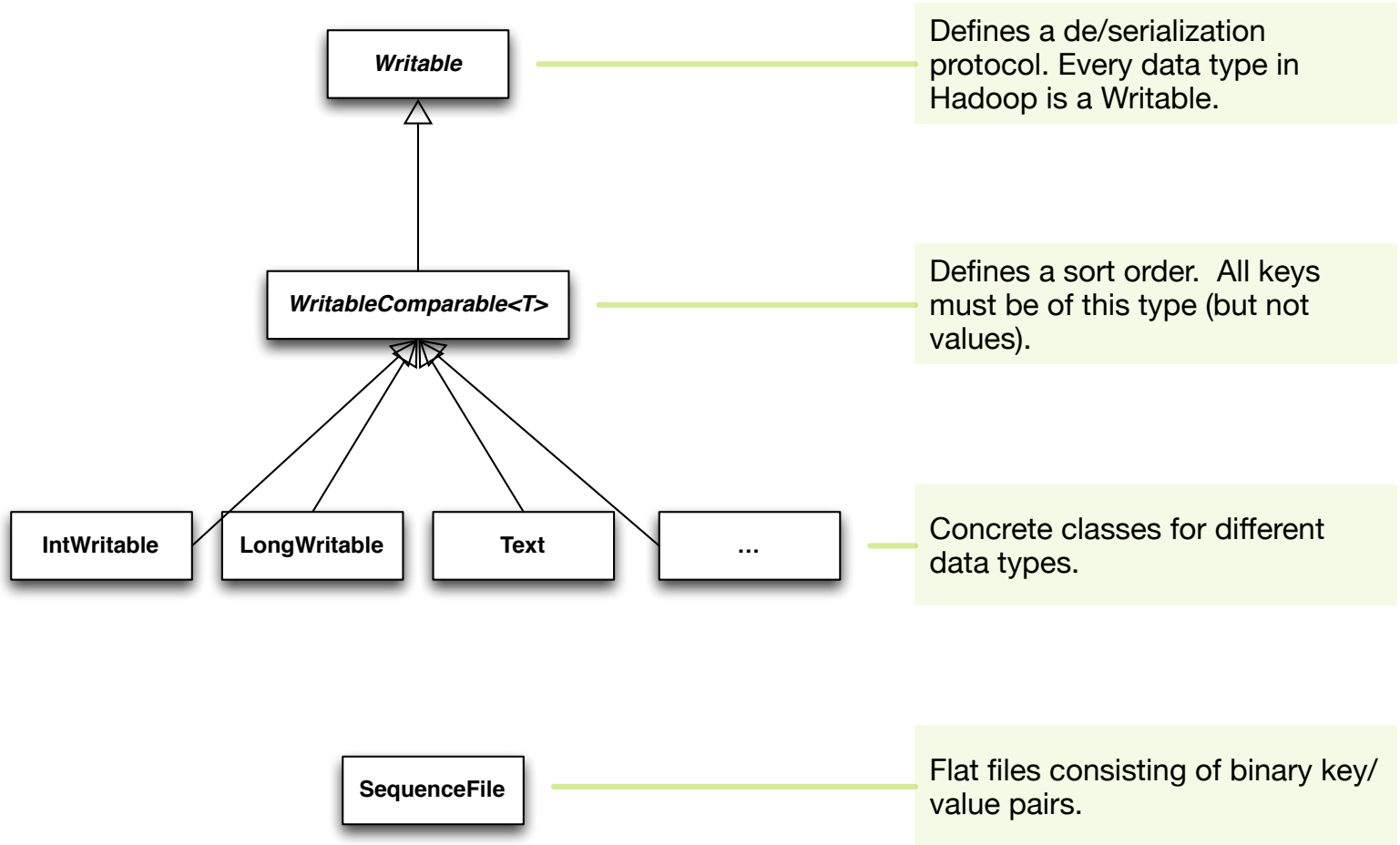
Partitioner and *Job*

- Partitioner
  - `int getPartition(K key, V value, int numPartitions)`
    Get the partition number given total number of partitions
- Job
  - Represents a packaged Hadoop job for submission to cluster
  - Need to specify input and output paths
  - Need to specify input and output formats
  - Need to specify mapper, reducer, combiner, partitioner classes
  - Need to specify intermediate and final key-value classes
  - Need to specify number of reducers (but not mappers, why?)
  - Don't depend of defaults!

Data types in Hadoop

Keys and values

- Defined in package `org.apache.hadoop.io`



Basic Hadoop API

The old and the new Java MapReduce APIs

- There are currently three Hadoop release lines that are developed in parallel:
  - Release line 0.x started in 2007-09
  - Release line 1.x started in 2011-12
  - Release line 2.x started in 2012-05
- The 1.x MapReduce API is used in this course, but on the web you will find references to the 0.x API.
  - (Release line 2.x supports the 1.x API)
- 1.x (**new**) API
  - Defined in package `org.apache.hadoop.`**`mapreduce`**
  - Introduced in Hadoop 0.20.0 which became the 1.x series.
  - Aka "Context Objects"
- 0.x (**old**) API
  - Defined in package `org.apache.hadoop.`**`mapred`**
  - Still supported in 1.x series.

Scalable Hadoop algorithms

General recommendations

- Avoid object creation whenever possible
  - A new object has costs in object creation and eventual garbage collection.
  - For example reuse *Writable* objects, change the payload.
    - Instead of
      ```
      context.write(new Text(word), new IntWritable(1));
      ```
    - Use
      ```
      public final static IntWritable ONE = new IntWritable(1);
      [...]
      context.write(new Text(word), WordCount.ONE);
      ```
- Avoid accumulating data in memory (buffering), instead write it out regularly
  - Heap size is limited.
  - In `map()` method do not try to accumulate all data in memory and only write it out in `cleanup()` method. Works for small datasets, but won't scale.

Local aggregation
Introduction

- Ideally we want a MapReduce program that
  - When processing twice the data takes twice the running time.
  - When having twice the number of nodes available takes half the running time.
- Why can't we achieve this?
  - Synchronization requires transfer of data.
  - Transfer of data kills performance.
- Therefore... avoid transfer of data!
  - Reduce the amount of intermediate data via local aggregation.
  - Combiners can help.

Local aggregation

Example: Word Count — Baseline and version 1.1

- We start with the basic Word Count implementation and try to improve its performance.
- Baseline implementation:
  - class Mapper

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

  - class Reducer

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

- Version 1.1: Use Combiners.
  - What is their impact?

Local aggregation

Example: Word Count — Version 2.0

- In version 2.0 we improve the Mapper.
  - Mapper remembers the word counts in a line of text using a HashMap.
  - Reducer remains unchanged.
- Version 2.0:
  - class Mapper

```
    method Map(String docid, String text):
        HashMap<String, Int> h;
        for each word w in text:
            h.put(w, h.get(w) + 1);
        for each key w in h:
            Emit(w, h.get(w));
```

- Are combiners still needed?

Local aggregation

Example: Word Count — Version 3.0

- In version 3.0 we improve the Mapper further.
  - Mapper remembers the word counts in the whole document. Let the HashMap persist between lines.
  - Reducer remains unchanged.
- Version 3.0:
  -
```
class Mapper
    method Initialize:
        HashMap<String, Int> h;
    method Map(String docid, String text):
        for each word w in text:
            h.put(w, h.get(w) + 1);
    method Close:
        for each key w in h:
            Emit(w, h.get(w));
```

- Are combiners still needed?

Local aggregation

Summary: *In-mapper combining* design pattern

- Version 3.0 of the Word Count example uses the so-called *In-mapper combining* design pattern.
  - Fold the functionality of the combiner into the mapper.
  - Preserve state across multiple map calls to combine the results.
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - The preserved state consumes memory.
    - Explicit memory management required.
  - Potential for order-dependent bugs.

Efficient counting of co-occurrences

How to resolve ambiguous terms

Maybe she'll change her name to Halliburton. Just to see.

3/18/2011 at 9:00 PM | 17 Comments

Mentions of the Name 'Anne Hathaway' May Drive Berkshire Hathaway Stock

By Patrick Huguenin

SHARE 3 TWEET 0

The Huffington Post recently pointed out that whenever Anne Hathaway is in the news, the stock price for Warren Buffett's Berkshire Hathaway goes up. Really. When *Bride Wars* opened, the stock rose 2.61 percent. (*Rachel Getting Married* only kicked it up 0.44 percent, but, you know, that one was so light on plot compared to *Bride Wars*.)

What is happening here?

Source: http://nymag.com/daily/intelligencer/2011/03/mentions_of_the_name_anne_hath.html

Efficient counting of co-occurrences

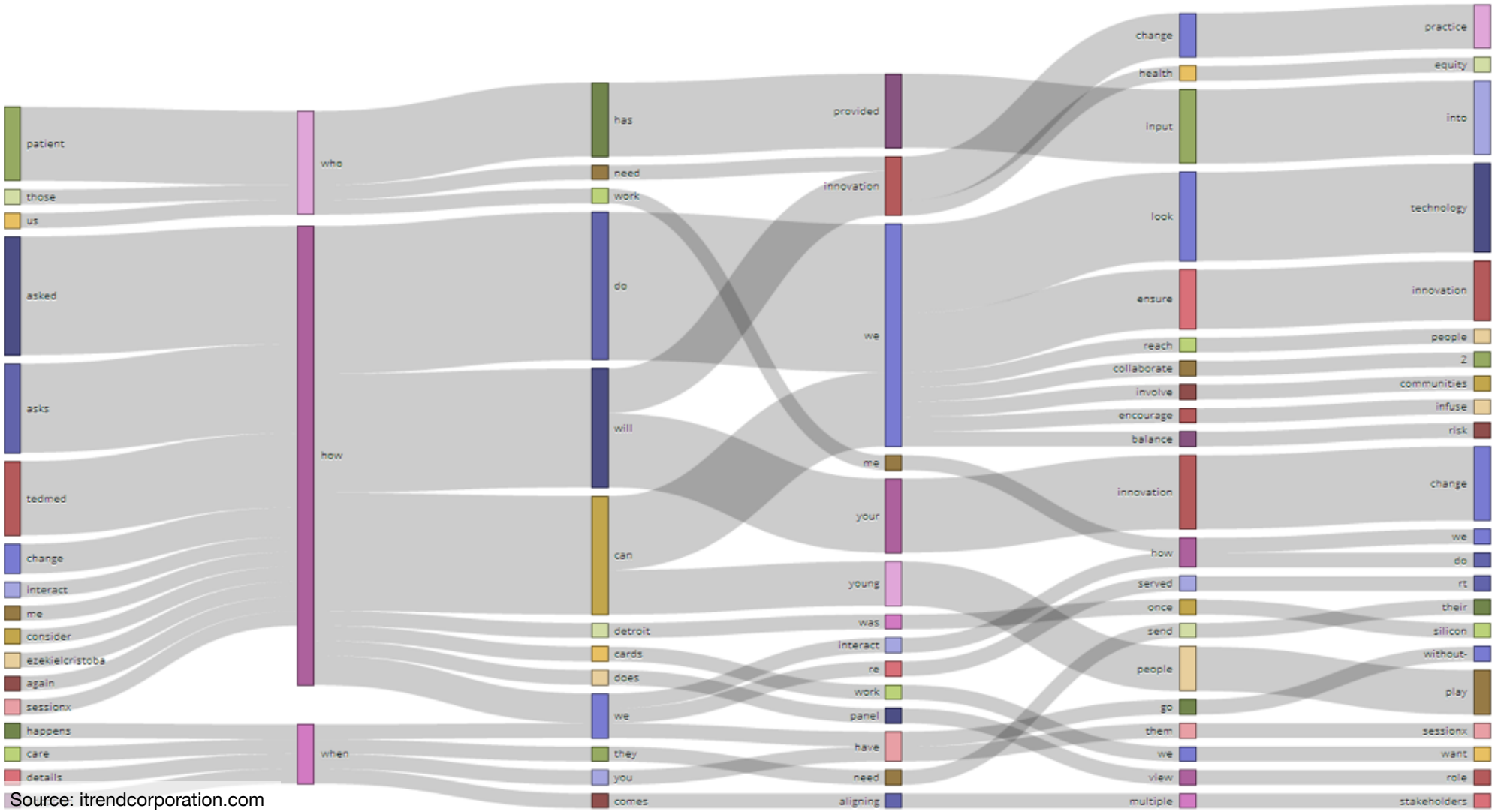
How to resolve ambiguous terms

- Automated stock trading algorithms do market prediction.
  - Sift through the Internet to see what people are talking about.
  - Use statistics to make predictions about the price of stocks.
  - Algorithms learned that in the past when the Internet talked about "Hathaway", the stock of the Berkshire Hathaway company went up.
  - Simplistic analysis confused two semantically separated concepts ("the company Berkshire Hathaway" and "the actress Anne Hathaway") that share the same word.

- How to avoid?
  - Instead of words, consider co-occurrence of several words.
    - Words co-occurring immediately one after the other (bigrams)
    - Words co-occurring in the same sentence
    - Words co-occurring in the same paragraph
    - Words co-occurring in the same document

Efficient counting of co-occurrences

How to resolve ambiguous terms — N-gram analysis



Source: itrendcorporation.com

Efficient counting of co-occurrences

Introduction to running example

- In the following we will calculate a *term co-occurrence matrix* M for a text collection
  - M is a N x N matrix, where N is the size of the vocabulary (= number of terms)
  - Matrix element M_{ij}: number of times i and j co-occur in some context

Efficient counting of co-occurrences

Introduction to running example

- Example corpus:
  - "Hathaway starred in dramatic films. Talented and beautiful actress Anne Hathaway."
- Bigram matrix of the corpus:

	actress	and	anne	beautiful	dramatic	films	hathaway	in	starred	talented
actress			1							
and				1						
anne							1			
beautiful	1									
dramatic						1				
films										
hathaway									1	
in					1					
starred								1		
talented		1								

Efficient counting of co-occurrences

Large counting problems

- Term co-occurrence is an example 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

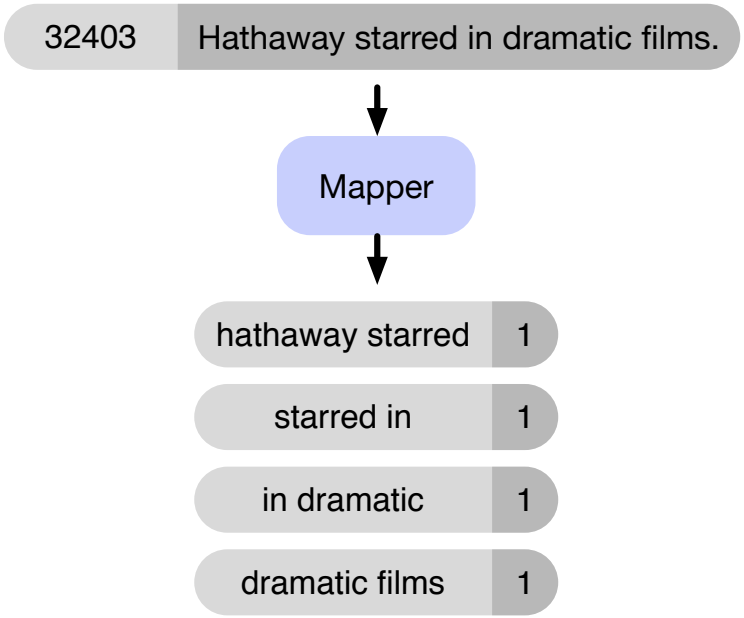
- Basic approach
  - Mappers generate partial counts
  - Reduce aggregate partial counts

- How do we aggregate partial counts efficiently?

Efficient counting of co-occurrences

Version 1.0: "Pairs"

- We start with a straightforward implementation.
  - Each mapper takes a sentence:
    - Generate all co-occurring term pairs.
    - For each pair, emit a key-value pair
  - Reducers sum up counts associated with these pairs.
- Use combiners!



Efficient counting of co-occurrences

Version 1.0: "Pairs"

- Pseudo-code

```
class Mapper
    method Map(String docid, String text):
        for each term t in text:
            for each term u in neighbors(t):
                Emit(pair(t, u), 1);                    // emit count for each co-occurrence


class Reducer
    method Reduce(pair p, Iterator<Int> values):
        int sum = 0;
        for each v in values:
            sum += v;                                   // sum co-occurrences
        Emit(pair p, sum);
```

Efficient counting of co-occurrences

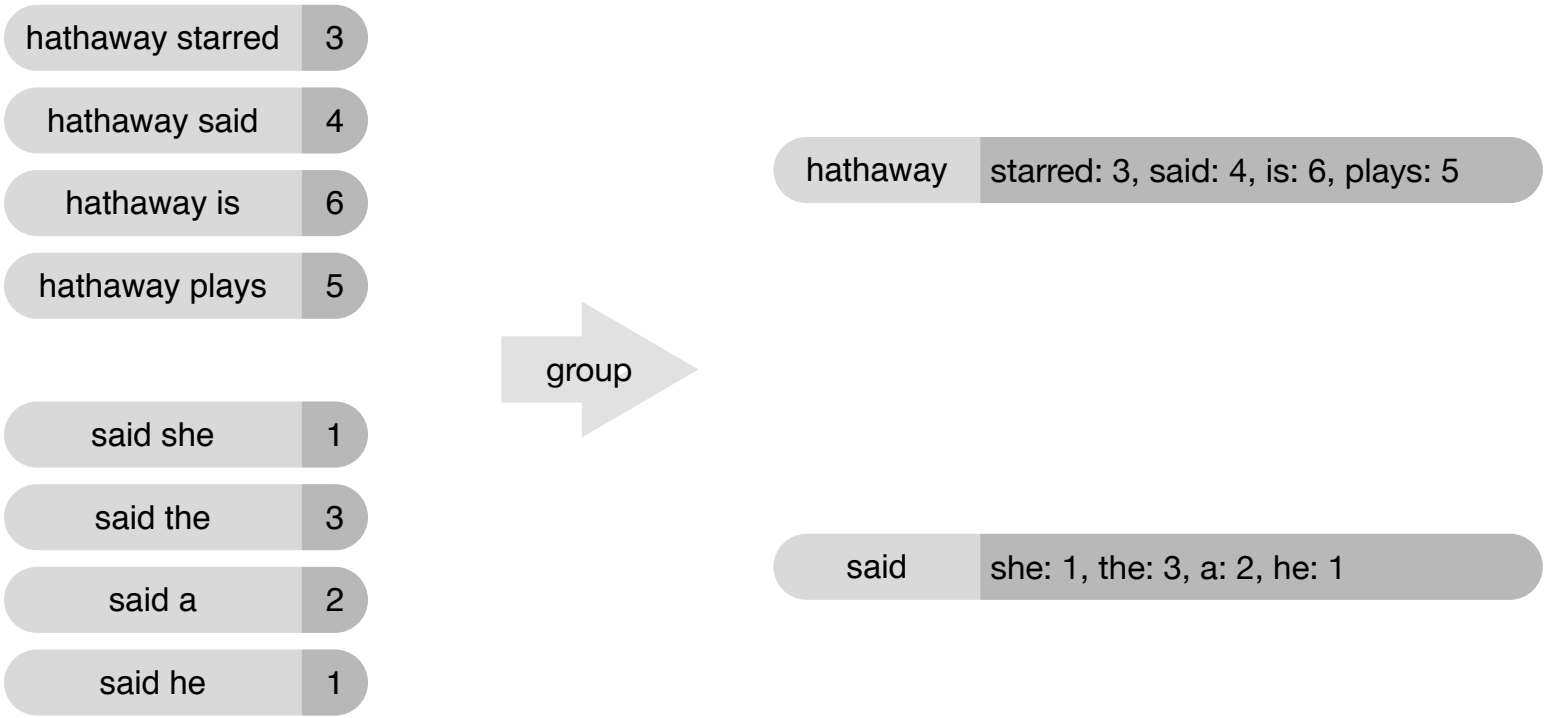
Version 1.0: "Pairs" — Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)
  - Not many opportunities for combiners to work

Efficient counting of co-occurrences

Version 2.0: "Stripes"

- Idea: group together pairs into an associative array (= hashmap)



- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term a, emit (a, {b: $count_{ab}$, c: $count_{ac}$, d: $count_{ad}$, ... })

Efficient counting of co-occurrences

Version 2.0: "Stripes"

- Reducers perform element-wise sum of associative arrays

	hathaway	starred: 2, is: 3, plays: 5
+	hathaway	starred: 1, said: 4, is: 5
	hathaway	starred: 4, said: 4, is: 8, plays: 5

Efficient counting of co-occurrences

Version 2.0: "Stripes"

- Pseudo-code

```
class Mapper
    method Map(String docid, String text):
        for each term t in text:
            HashMap h;
            for each term u in neighbors(t):
                h.put(h.get(u) + 1);                    // tally words co-occurring with t
            Emit(t, h);


class Reducer
    method Reduce(String t, Iterator<HashMap> stripes):
        HashMap h;
        for each stripe in stripes:
            sum(h, stripe);                             // element-wise sum
        Emit(t, h);
```

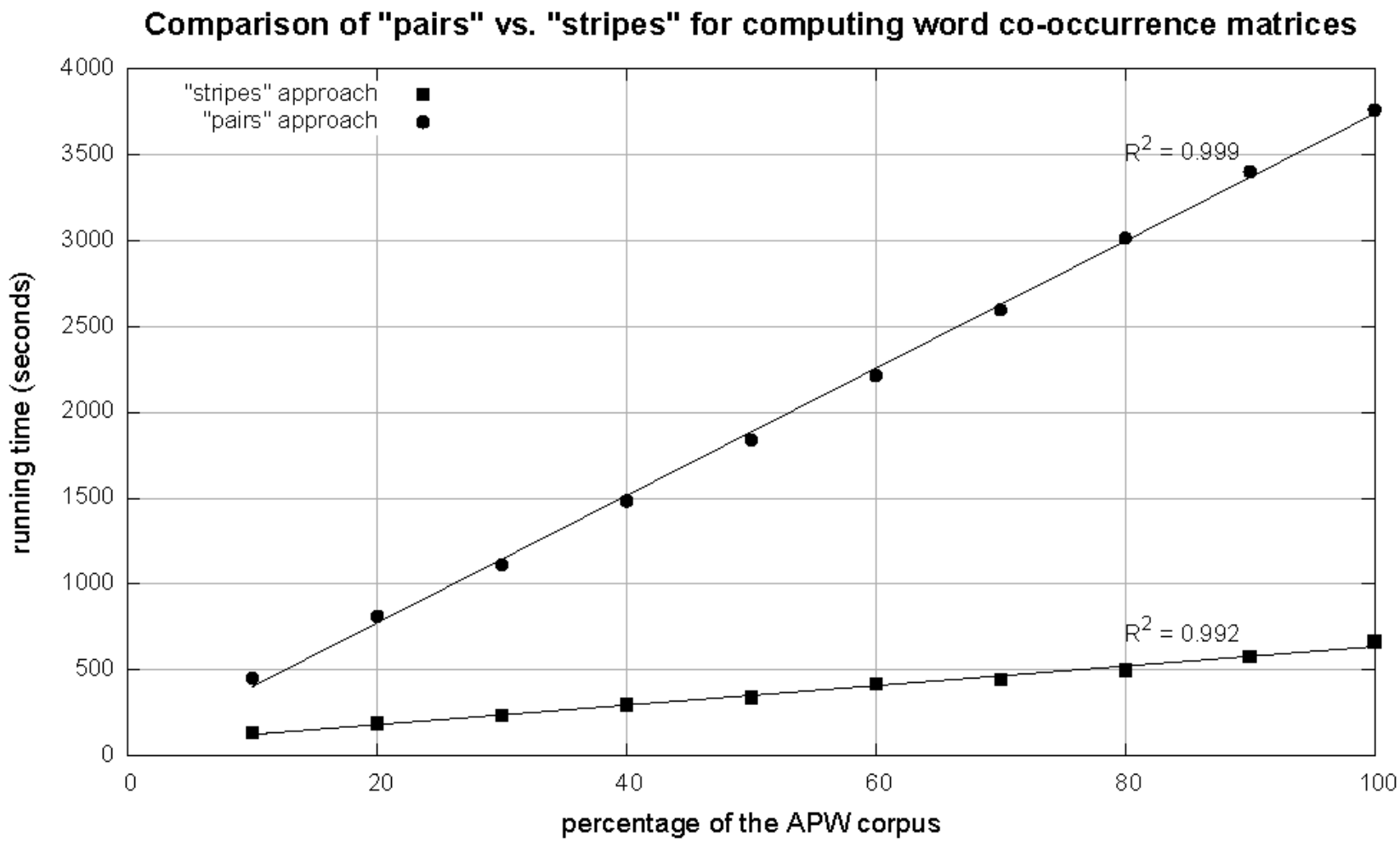
Efficient counting of co-occurrences

Version 2.0: "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 more heavyweight
  - Fundamental limitation in terms of size of event space

Efficient counting of co-occurrences

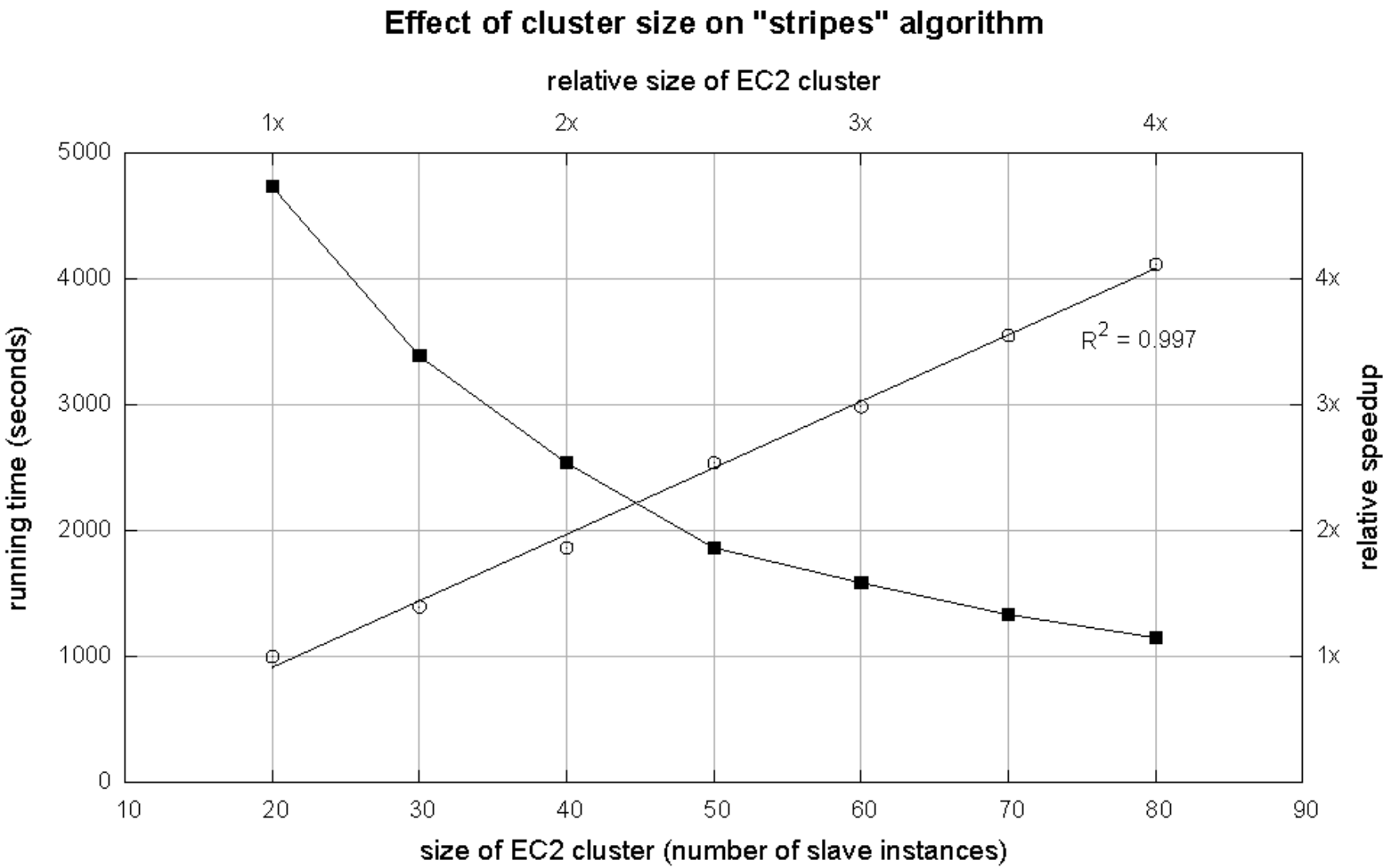
Pairs vs. Stripes performance



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)

Efficient counting of co-occurrences

Stripes scalability



Cloud Computing

Introduction

- **Definition:** Cloud computing is a model for
  - **enabling** ubiquitous, convenient, on-demand network **access to** a
  - shared pool of configurable **computing resources** (e.g., networks, servers, storage, applications, and services)
    - that can be rapidly provisioned and released with minimal management effort or service provider interaction.

- Five **essential characteristics**:
  - **On-demand self service** (automatic provisioning without requiring human interaction)
  - **Broad network access** (access via standardized protocols from a variety of clients)
  - **Resource pooling** (multi-tenant model, dynamic resource assignment, location independence)
  - **Rapid elasticity** (rapid provisioning/deprovisioning to scale out/in, seemingly unlimited capacity)
  - **Measured service** (usage is monitored and controlled, providing transparency)

- Three **service models**:
  - **Infrastructure as a Service** (IaaS)
  - **Platform as a Service** (PaaS)
  - **Software as a Service** (SaaS)

- Four **deployment models**:
  - **Private** cloud
  - **Community** cloud
  - **Public** cloud
  - **Hybrid** cloud

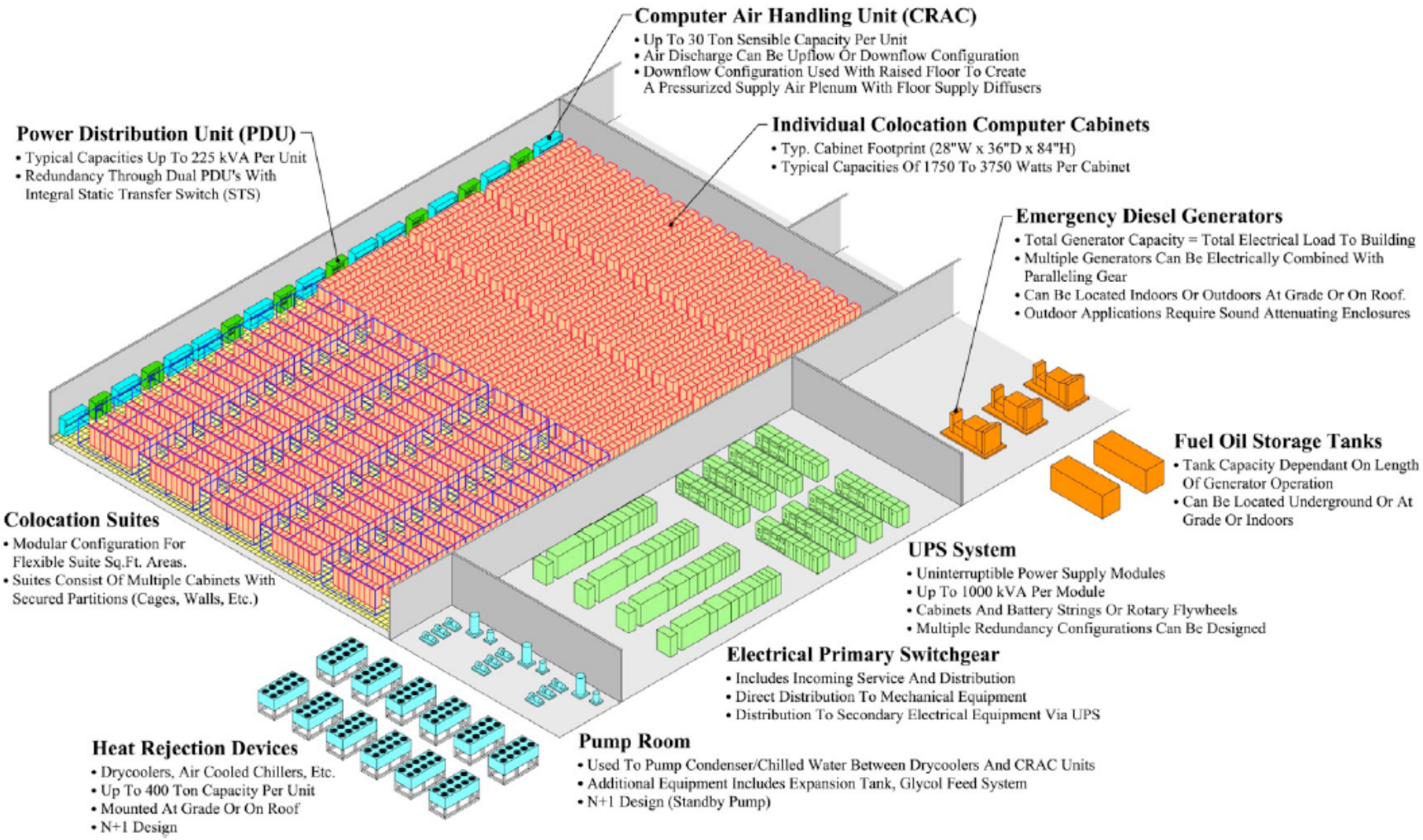
Cloud Computing

What is a data center?

- A data center
  - Hosts computers and associated components, such as networks and storage systems, cooling systems, air filters, uninterruptible power supplies, ...
  - Hosts typically a large number of interconnected computing systems
  - Can occupy a room in a building, one or more floors, or an entire building.
    - The largest data centers occupy 65'000 m^2, ~9 soccer fields
- Estimations for an Amazon data center of average size
  - Hosts 46'000 servers
  - Costs $88M to build
  - Consumes 8 MW electrical power
  - Operational costs:
    - 57% servers (amortized over 3 years)
    - 18% energy distribution and cooling
    - 13% energy

Cloud Computing

Anatomy of a datacenter



Source: Barroso and Hölzle (2009)

Cloud Computing

A data center

Amazon Web Services

Introduction

- Amazon Web Services (AWS) is a collection of remote infrastructure services that together form an *Infrastructure as a Service* (IaaS) offering

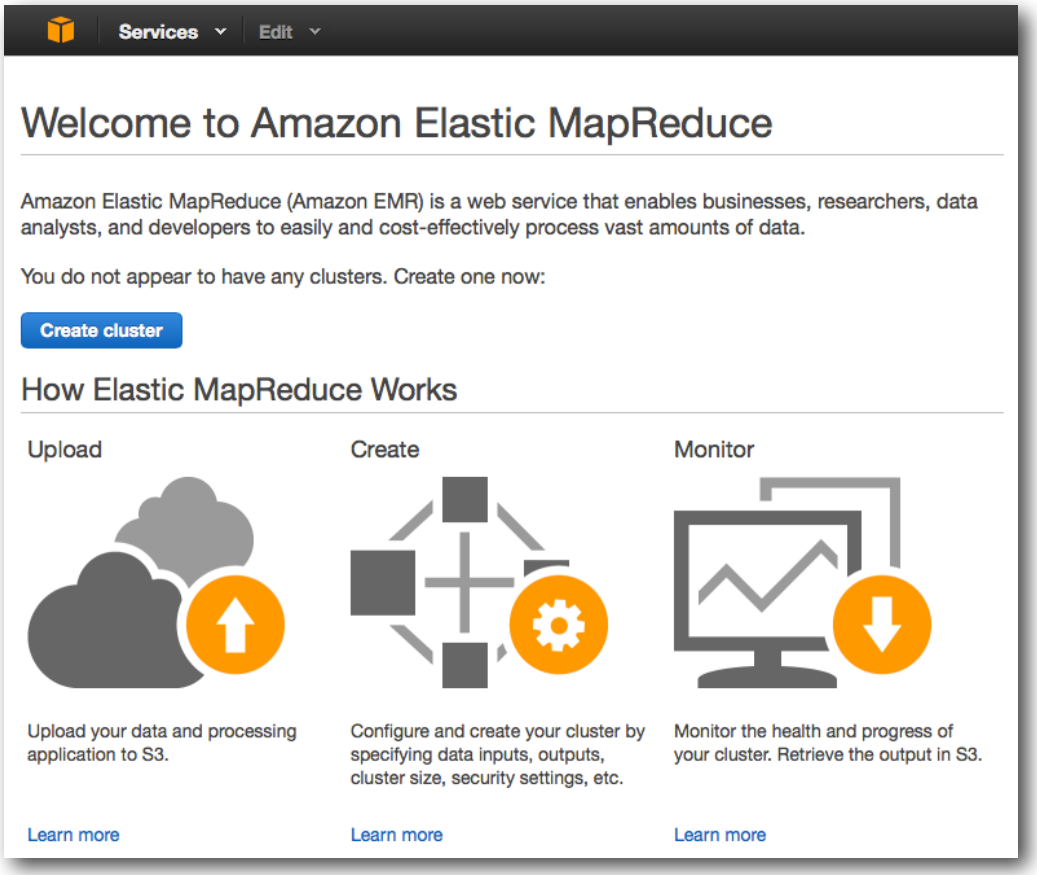
- AWS offers the following categories of services
  - **Compute** — for example *Elastic Compute Cloud*
  - **Storage** — for example *Simple Storage Service*
  - **Database** — for example *Relational Database Service*
  - **Networking** — for example *Virtual Private Cloud*
  - ...

- The services are targeted towards *operations engineers* and *developers*.

Amazon Web Services

Elastic MapReduce

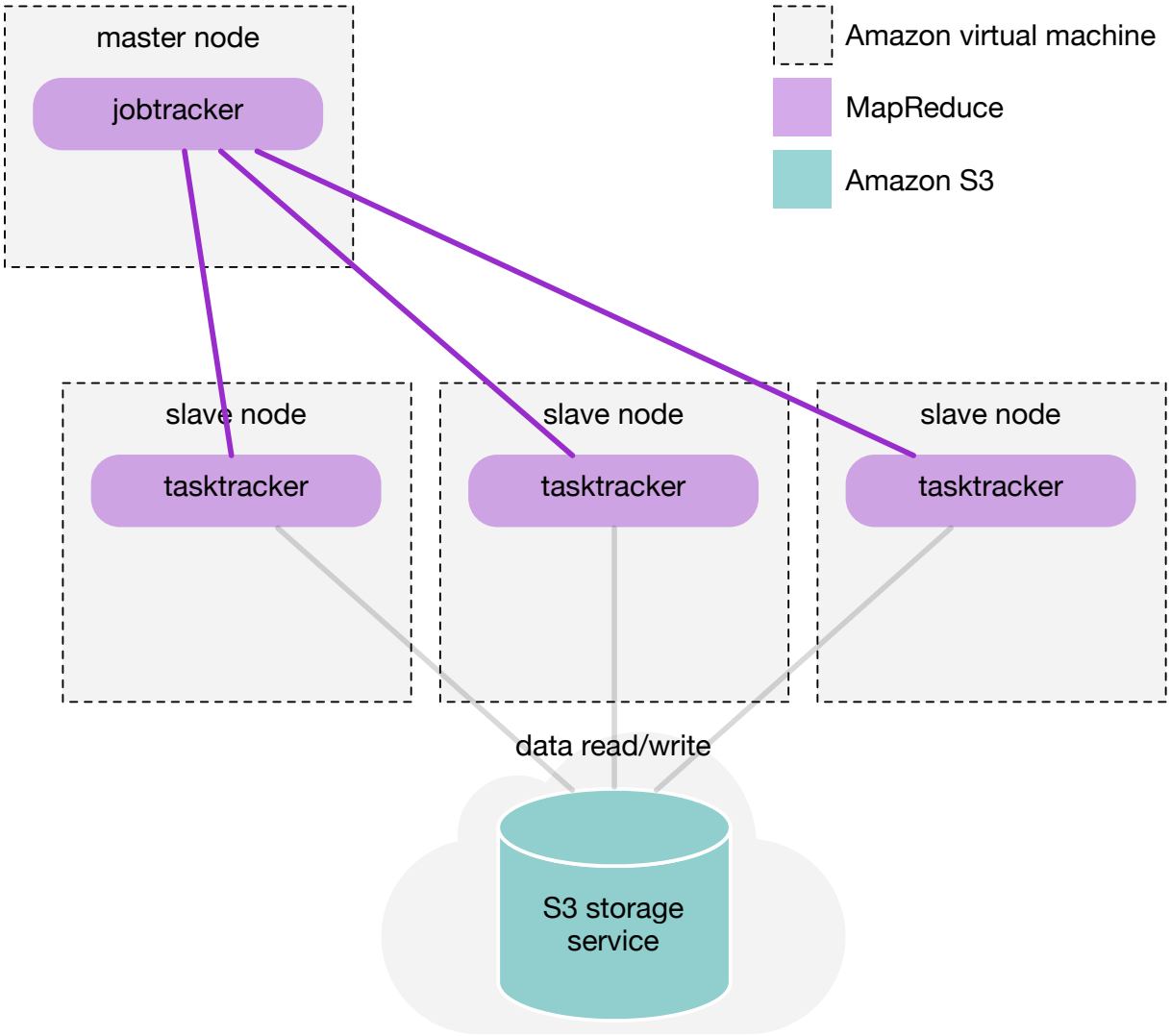
- Amazon offers Hadoop Clusters as a Service, called *Elastic MapReduce* (EMR).
- EMR is accessible via
  - a Web-based user interface
  - an API / command line
- EMR is based on virtual machines (called *instances*) provided by the *Elastic Compute Cloud* (EC2) service.
  - Virtual machines are charged by the hour, so it is a good idea to release them when they are no longer needed for computation.
  - When a virtual machine is released its data is gone.



Amazon Web Services

Elastic MapReduce architecture

- Elastic MapReduce does not use HDFS to store the data, but S3, which is Amazon's cloud-based storage service.
- The reason: one wants to release the virtual machines after the MapReduce job is finished, otherwise they continue to cost money. VMs are not good for storing data permanently.
- Data is stored in an S3 *bucket*. A bucket resides in a particular *region*. The Hadoop cluster should reside in the same region, otherwise Amazon will bill for the data transfer between two regions.



Amazon Web Services

Elastic MapReduce pricing

- Elastic MapReduce jobs require one instance to control the cluster plus 1 to n instances to perform the work.
- Amazon charges from the time the cluster begins processing until it is terminated.
  - **Partial hours count as full hours.**
- Prices for On-Demand instances in US East region (General Purpose family only)

Instance type	Proc. arch.	vCPU	ECU	Memory (GiB)	Price (EC2 + EMR)
m1.small	32-bit or 64-bit	1	1	1.7	$0.075 per hour
m1.medium	32-bit or 64-bit	1	2	3.75	$0.15 per hour
m1.large	64-bit	2	4	7.5	$0.30 per hour
m1.xlarge	64-bit	4	8	15	$0.60 per hour

Amazon Web Services

S3 pricing

- The prices for S3 have three components:
  - Storage: Each GB of data stored is charged per month.
  - Request: Each file read or write request is charged.
  - Data transfer: Each GB of data transferred *out* is charged. Data transferred *in* is free.

Storage	
First 1 TB / month	$0.0300 per GB
Next 49 TB / month	$0.0295 per GB
Next 450 TB / month	$0.0290 per GB
Next 500 TB / month	$0.0285 per GB
Next 4000 TB / month	$0.0280 per GB
Over 5000 TB / month	$0.0275 per GB

Requests	
PUT, COPY, POST, or LIST Requests	$0.005 per 1,000 requests
Delete Requests	Free
GET and all other Requests	$0.004 per 10,000 requests

Data transfer out	
First 1 GB / month	$0.000 per GB
Up to 10 TB / month	$0.120 per GB
Next 40 TB / month	$0.090 per GB
Next 100 TB / month	$0.070 per GB
Next 350 TB / month	$0.050 per GB

Relative frequencies

Improving co-occurrence counts

- So far we have computed how often two words co-occur in absolute counts.
- If the bigram "Euro crisis" occurs 10 times in a text, is that a significant occurrence?
  - Depends on how often the individual words occur in the text.
    - If "Euro" occurs only 12 times, it is certainly significant.
- *Relative frequency* of *B* in the context of *A*:

$$f(B|A) = \frac{N(A,B)}{N(A)}$$

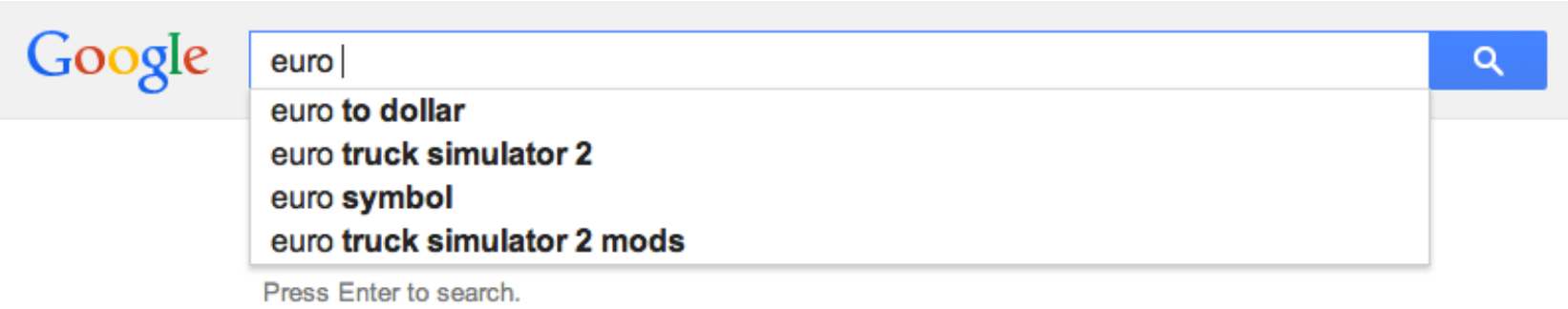
- How to obtain *N(A)*?
  - Write a two-step job: First compute occurrence of single words, then of Bigrams.
  - Single-step job: Use the Bigram counts to compute single word counts:

$$N(A) = \Sigma_{B'} N(A, B')$$

  - *N(A)* is also known as a *marginal count*, *N(A, B)* as *joint count*.

Relative frequencies

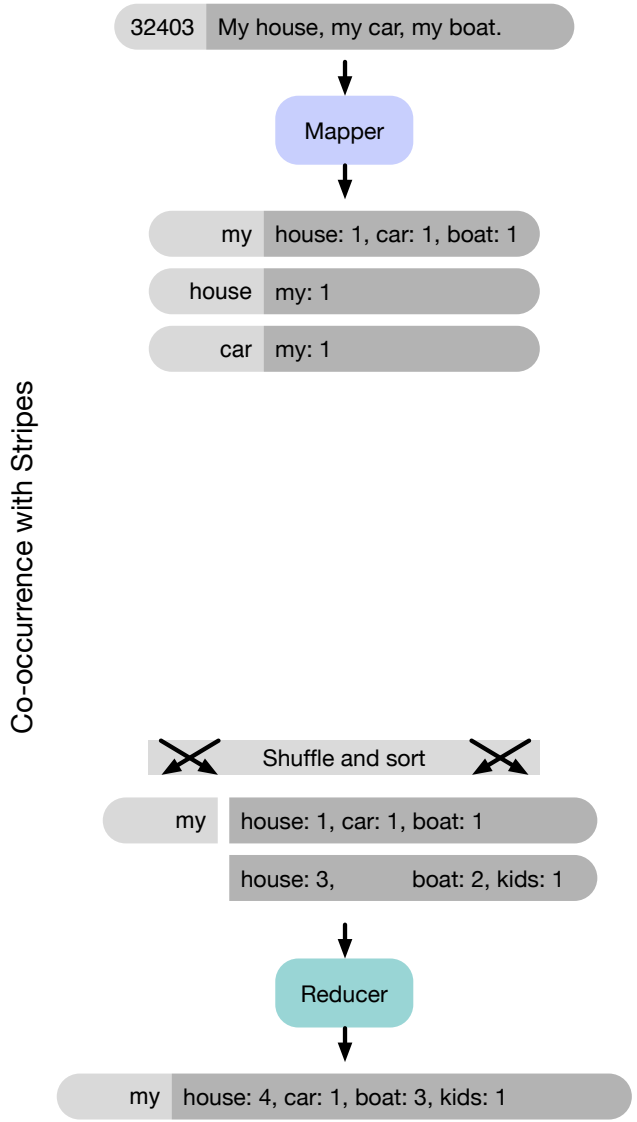
Application: Auto-completion in search bar

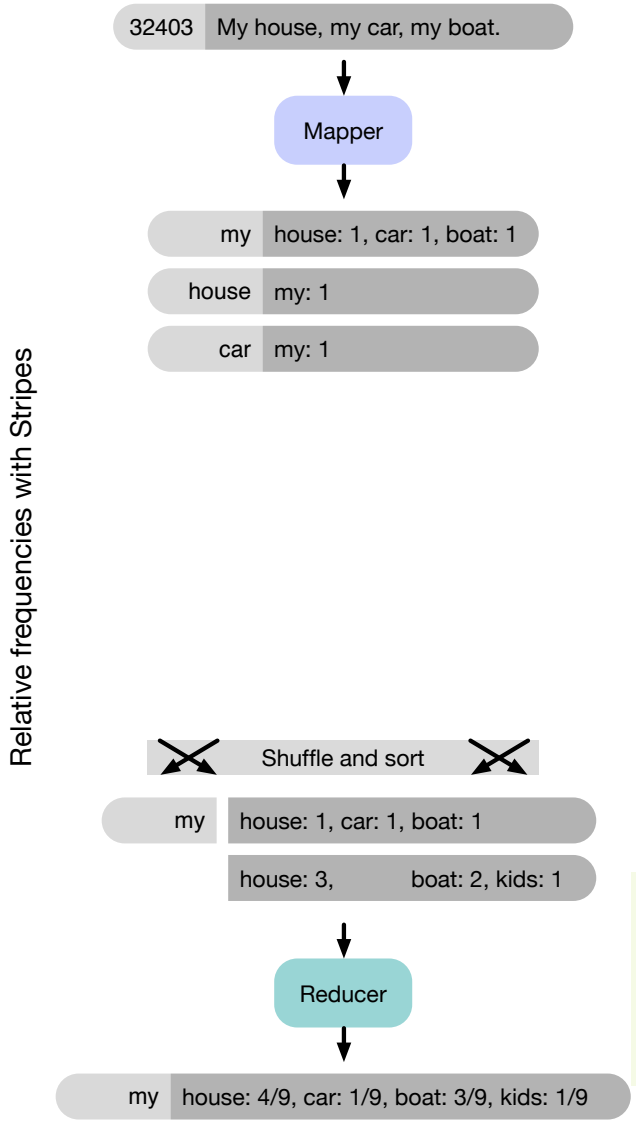


- As soon as the user types a search term, the auto-complete function suggests additional terms likely to be wanted by the user.
- A simplified auto-complete function could work as follows:
  - Remember the history of search terms entered for all users. This is the corpus.
  - Compute relative frequencies of n-grams in this corpus.
  - When the user has entered a term, pick among the n-grams starting with this term the ones with the highest relative frequencies. These become the suggestions.

Relative frequencies

How to compute using Stripes





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

Relative frequencies

How to compute using Pairs

- What's the issue?
  - Computing relative frequencies requires marginal counts.
  - But the marginal cannot be computed until you see all counts.
  - Buffering is a bad idea!
- Solution:
  - What if we could get the marginal count to arrive at the reducer first?

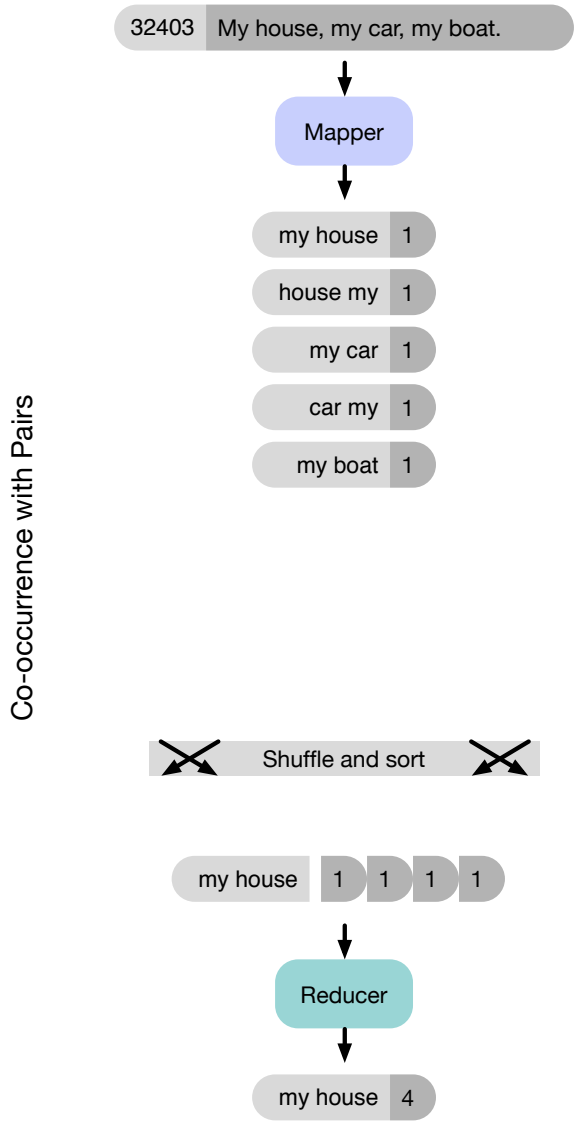
Relative frequencies

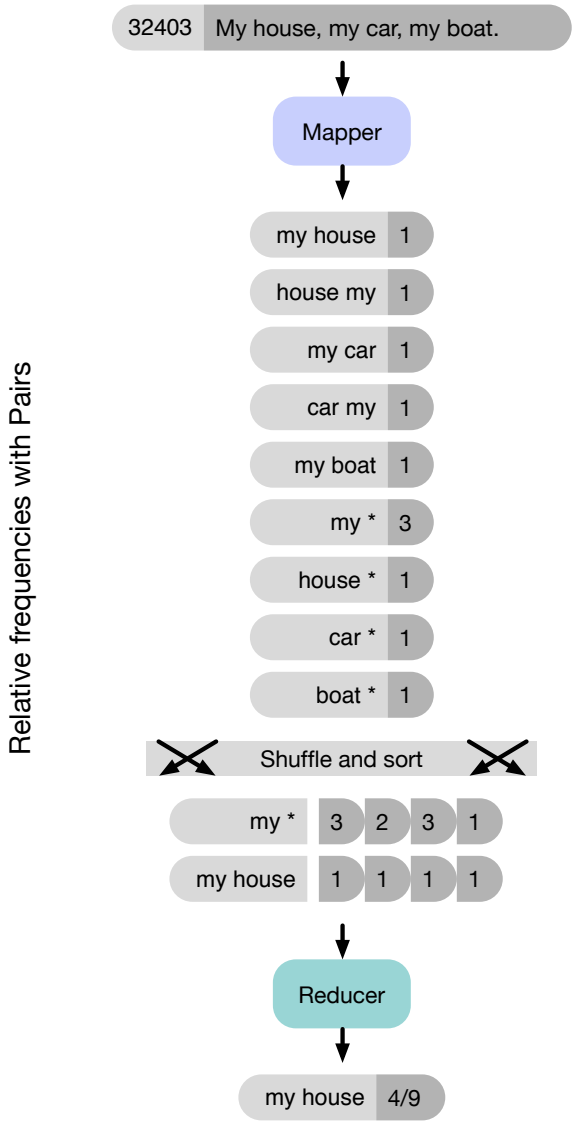
How to compute using Pairs

- 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.

Relative frequencies

How to compute using Pairs





Relative frequencies

Order Inversion design pattern

- Common design pattern:
  - Take advantage of sorted key order at reducer to sequence computations.
  - Get the marginal counts to arrive at the reducer before the joint counts.
- Optimization:
  - Apply in-memory combining pattern to accumulate marginal counts.

Relative frequencies

Synchronization approaches: Pairs vs. Stripes

- 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 partial results together
  - Each reducer receives all the data it needs to complete the computation.
  - Illustrated by the “stripes” approach.