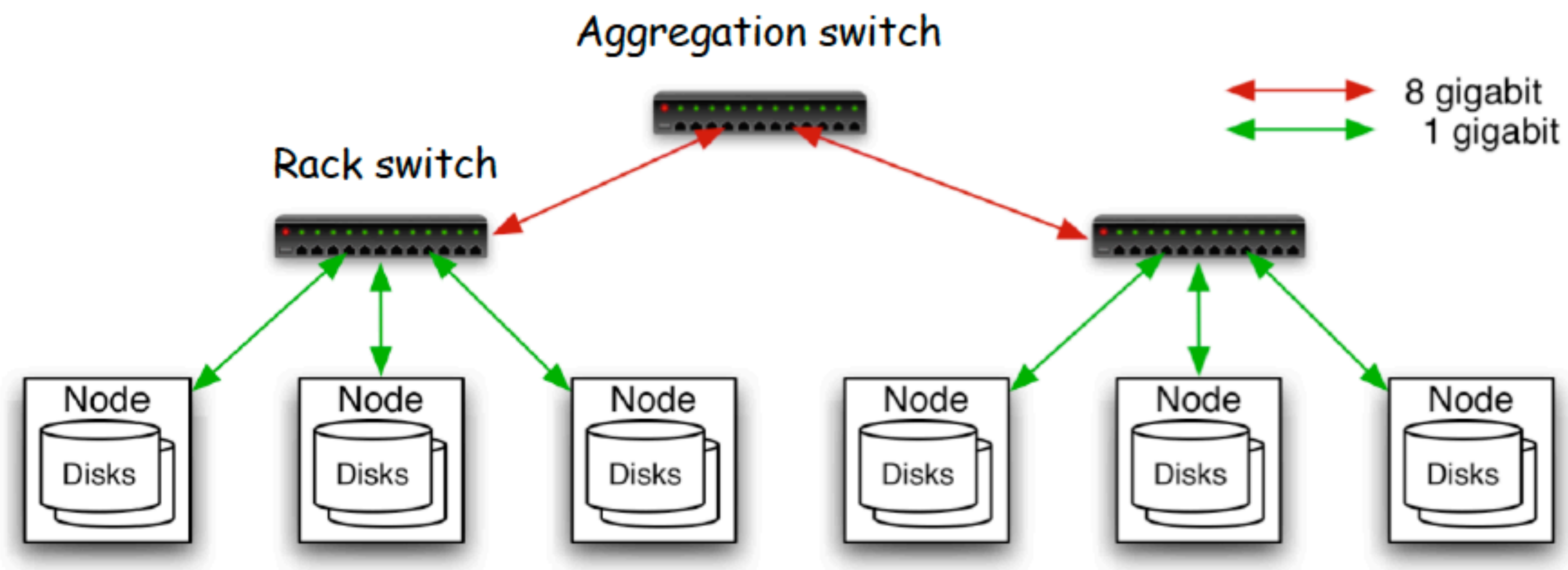
MapReduce
A High-level Programming Model (with a Complex Runtime System)

Irene Finocchi

Commodity clusters

- Cannot mine tens to hundreds of Terabytes of data on a single server
- Standard architecture emerging:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnections
- How to organize computations on these architectures?
- How to program these architectures?
- How to mask issues such as hardware failures in these architectures?

Cluster architecture



- Each rack contains 10/64 nodes
- Sample node configuration: 8 x 2GHz cores, 8 GB RAM, 4 disks (4 TB)

Real cluster architecture

Stable storage

- First order problem: if nodes can fail, how can we store data persistently?
- Answer: Distributed File System
  - Provides global file namespace
  - Google GFS; Hadoop HDFS; Kosmix KFS
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

Distributed file system

- Chunk servers
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- Master node
  - Stores metadata
  - Might be replicated
  - (a.k.a. Name Node in HDFS)
- Client library for file access
  - Talk to master to find chunk servers
  - Connect directly to chunk servers to access data

Warm up: word count

- We have a large corpus of documents, one word per line
- Count the number of times each distinct word occurs in the corpus
  - `words(docs/*) | sort | uniq -c`
  - where **`words`** takes a file and outputs the words in it, one to a line

- Sample application: analyze web server logs to find popular URLs
- The above captures the essence of MapReduce
  - Great thing is it is naturally parallelizable

MapReduce

- A novel programming model
- Everything built on top of <key,value> pairs
  - Keys and values are user-defined: can be anything
- Only two user-defined functions:
  - Map
    - $map(k_1,v_1) \rightarrow list(k_2,v_2)$
    - given input data $<k_1,v_1>$, produce intermediate data v_2 labeled with key k_2
  - Reduce
    - $reduce(k_2, list(v_2)) \rightarrow list(v_3)$ preserves key
    - given a list of values $list(v_2)$ associated with a key k_2, return a list of values $list(v_3)$ associated with the same key

Parallelism in MapReduce

- All mappers in parallel
- All reducers in parallel
- Different pairs transparently distributed across available machines

$\text{map}(k_1,v_1) \rightarrow \text{list}(k_2,v_2)$

Shuffle: group values with the same key to be passed to a single reducer

$\text{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3)$

Runtime system

- The underlying runtime system:
  - automatically parallelizes the computation across large-scale clusters of machines
  - handles machine failures
  - schedules inter-machine communication to make efficient use of the network and disks

Implementations

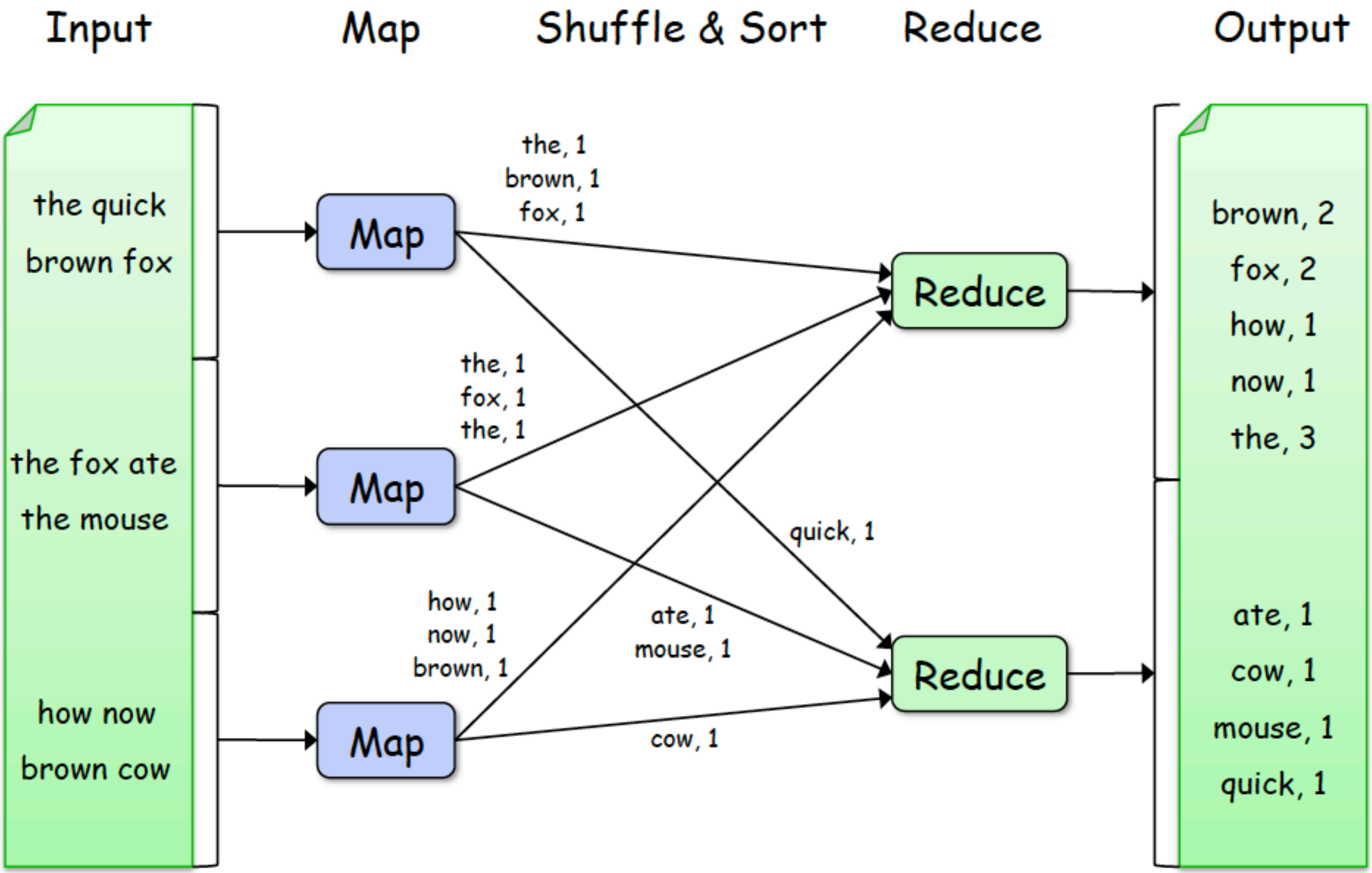
- Google MapReduce
  - Not available outside Google
- Hadoop: an Apache project
  - Open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://lucene.apache.org/hadoop/
- Aster Data
  - Cluster-optimized SQL Database that also implements MapReduce

THE MapReduce example: WordCount

```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
        emit(w, 1)

reduce(key, values):
// key: a word; value: an iterator over counts
    result = 0
    for each count v in values:
        result += v
    emit(result)
```

WordCount data flow



A programmer's perspective

> *The beauty of MapReduce is that any programmer can understand it, and its power comes from being able to harness thousands of computers behind that simple interface.*
>
> David Patterson

Implementation sketch: WordCount again

```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
        emit(w, 1)

reduce(key, values):
// key: a word; value: an iterator over counts
    result = 0
    for each count v in values:
        result += v
    emit(result)
```

WordCount in Hadoop (sketch)

The class org.apache.hadoop.mapreduce.Job

Job is the class used to submit a MapReduce task to the cluster:

```
Job job = Job.getInstance(new Configuration());
job.setJarByClass(MyJob.class);

// Specify various job-specific parameters
job.setJobName("myjob");

job.setInputPath(new Path("in"));
job.setOutputPath(new Path("out"));

job.setMapperClass(MyJob.MyMapper.class);
job.setReducerClass(MyJob.MyReducer.class);

/* Submit the job, then poll for progress until
 *  the job is complete */
job.waitForCompletion(true);
```

WordCount in Hadoop (sketch)

The Mapper and Reducer classes

The way to define what a Job should do is to assign it with our custom Mapper and Reducer subclasses:

The Mapper class:

```
org.apache.hadoop.mapreduce.Mapper
          <KEYIN,VALUEIN,KEYOUT,VALUEOUT>
```

The Reducer class:

```
org.apache.hadoop.mapreduce.Reducer
          <KEYIN,VALUEIN,KEYOUT,VALUEOUT>
```

WordCount in Hadoop (sketch)

Methods of the Mapper class

```
protected void setup(Context context) throws
      IOException, InterruptedException { }

protected void cleanup(Context context) throws
      IOException, InterruptedException { }

protected void map(KEYIN key, VALUEIN value,
            Context context) throws
      IOException, InterruptedException { }
```

WordCount in Hadoop (sketch)

Methods of the Reducer class

```
protected void setup(Context context) throws
      IOException, InterruptedException { }

protected void cleanup(Context context) throws
      IOException, InterruptedException { }

protected void reduce(KEYIN key, Iterable<VALUEIN> value,
            Context context) throws
      IOException, InterruptedException { }
```

WordCount in Hadoop (sketch)

The Context(s)

Finally we should consider the `Context` classes.
These are inner classes of the `Mapper` and `Reducer` classes.
We are interested in the method:

```
write(KEYOUT, VALUEOUT)
```

(inherited from `org.apache.hadoop.mapreduce.TaskInputOutputContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT>`)

WordCount in Hadoop (sketch)

Missing parts

- MyMapper fields:

```
private final static IntWritable one =
    new IntWritable(1);
private Text word = new Text();
```

- MyMapper map function body:

```
Scanner scanner = new Scanner(value.toString());
scanner.useDelimiter(" ");
while (scanner.hasNext()) {
    word.set(scanner.next());
    context.write(word, one);
}
scanner.close();
```

- MyReducer reduce function body:

```
int sum = 0;
for(IntWritable value : values) {
    sum += value.get();
}
context.write(key, new IntWritable(sum));
```

Cloud computing

- Ability to rent computing by the hour
  - Additional services: e.g., persistent storage

- E.g., Amazon Web Services
  - Elastic Compute Cloud: EC2
  - Persistent storage: S3
  - Elastic MapReduce: run Hadoop on EC2

Big data computing course: free AWS access, run your MapReduce apps on EC2 (up to 16 nodes)

AWS+Hadoop: a success story

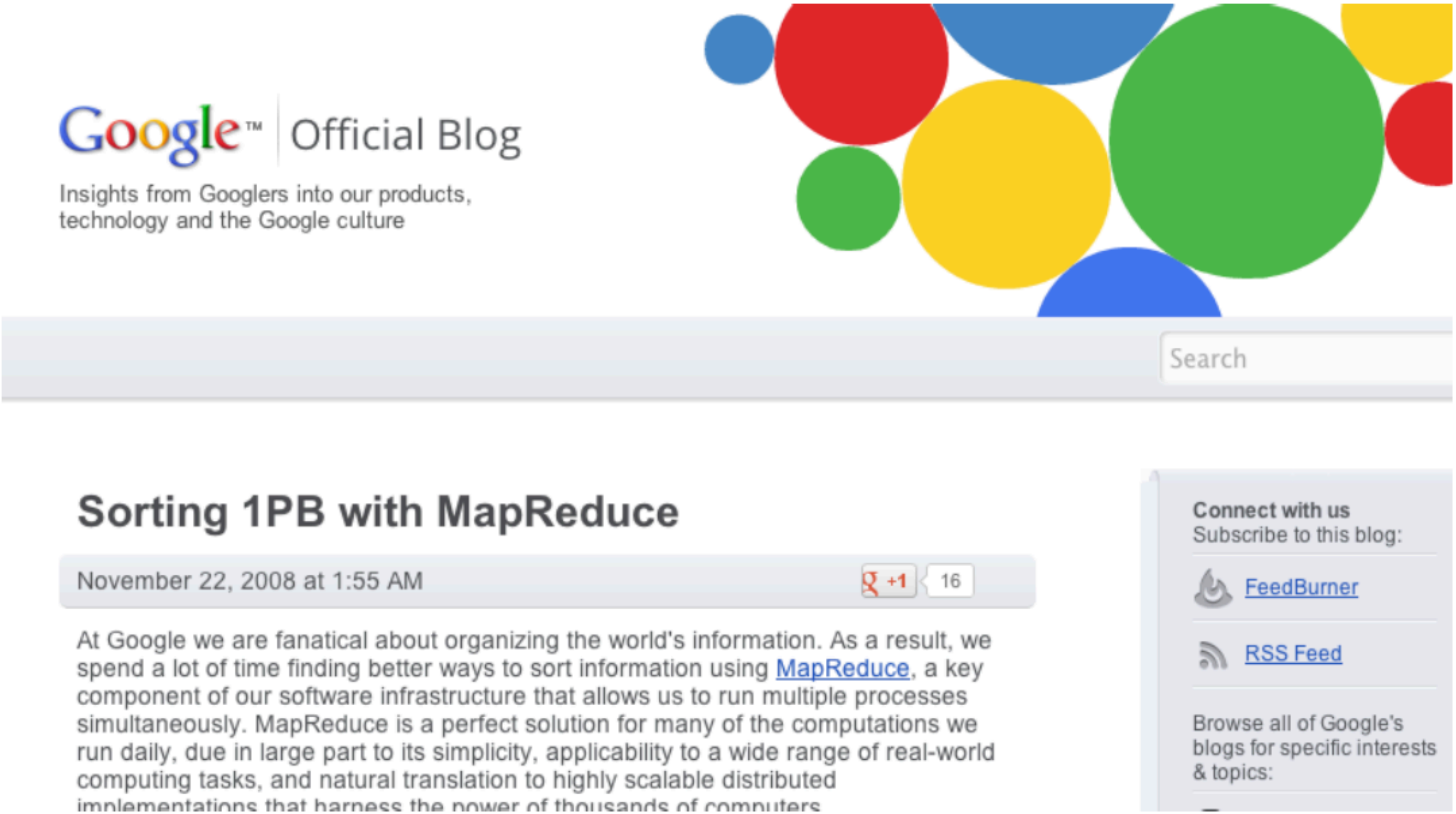
- The New York Times needed to generate PDF files for 11,000,000 articles (every article from 1851-1980) in the form of images scanned from the original paper
- Each article composed of numerous TIFF images scaled and glued together
- Code for generating PDF quite straightforward

NYT technologies and results

- 4TB of scanned articles sent to Amazon S3
- A cluster of EC2 machines configured to distribute the PDF generation via Hadoop
- Using 100 EC2 instances, in 24 hours the New York Times was able to convert the 4TB of scanned articles into 1.5TB of PDF documents

- Embarrassingly parallel problem

Another success: sorting on commodity clusters



Google™ | Official Blog

Insights from Googlers into our products, technology and the Google culture

Search

Sorting 1PB with MapReduce

November 22, 2008 at 1:55 AM +1 16

At Google we are fanatical about organizing the world's information. As a result, we spend a lot of time finding better ways to sort information using MapReduce, a key component of our software infrastructure that allows us to run multiple processes simultaneously. MapReduce is a perfect solution for many of the computations we run daily, due in large part to its simplicity, applicability to a wide range of real-world computing tasks, and natural translation to highly scalable distributed implementations that harness the power of thousands of computers

Connect with us
Subscribe to this blog:

FeedBurner

RSS Feed

Browse all of Google's blogs for specific interests & topics:

Sorting on commodity clusters

- Nov 2008: 1TB, 1000 computers, 68 secs
  - Previous record: 910 computers, 209 secs
- Nov 2008: 1PB, 4000 computers, 6 h, 48k harddisks
- Sept 2011: 1PB, 8000 computers, 33 m
- Sept 2011: 10PB, 8000 computers, 6 ½ h

10PB?

Metric prefixes						
Prefix	Symbol	1000^m	10^n	Decimal	English word[n 1]	Since[n 2]
yotta	Y	1000^8	10^{24}	1 000 000 000 000 000 000 000 000	septillion	1991
zetta	Z	1000^7	10^{21}	1 000 000 000 000 000 000 000	sextillion	1991
exa	E	1000^6	10^{18}	1 000 000 000 000 000 000	quintillion	1975
peta	P	1000^5	10^{15}	1 000 000 000 000 000	quadrillion	1975
tera	T	1000^4	10^{12}	1 000 000 000 000	trillion	1960
giga	G	1000^3	10^9	1 000 000 000	billion	1960
mega	M	1000^2	10^6	1 000 000	million	1960

Readings

- J. Leskovec, A. Rajaraman & J. Ullman

  Mining of massive data sets

  Chapters 1 and 2 (Sections 2.1 & 2.2)

  http://i.stanford.edu/~ullman/mmds.html

- Jeffrey Dean and Sanjay Ghemawat,

  MapReduce: Simplified Data Processing on Large Clusters

  http://labs.google.com/papers/mapreduce.html