MapReduce is

- a *programming model* for expressing distributed computations on massive datasets, and
- an *execution framework* for large-scale data processing on commodity clusters

Developed at Google in 2004 (Jeffrey Dean & Sanjay Ghemawat).

An open-source version called *Hadoop* was later developed at Yahoo.

Hadoop is now an Apache project.

*Amazon Elastic MapReduce* runs Hadoop on Amazon EC2.
MapReduce

MapReduce provides

- Simple API’s, and
- Automatic
  - Parallelization
  - Data distribution
  - Load balancing
  - Fault tolerance
Big Ideas behind MapReduce

Scale Out Instead of Scaling Up: A large number of commodity low-end servers is preferred over a small number of high-end servers.

Be Ready to Tackle Failures: Failures are the norm at warehouse scale computing.

Move Code to the Data: Code transfer is much cheaper than transferring massive amounts of data.

Process Data Sequentially: Random accesses to data stored on disks are much costlier than sequential accesses.

Hide System-Level Details from Programmers: Provide a simple abstraction that is easy to reason about.

Seamless Scalability: A simple programming model to approach ideal scaling characteristics in many circumstances.
A Simplified View of MapReduce

Input Files
- Input key-value pairs

Map
- mapper
- Intermediate key-value pairs

Shuffle
- Shuffle and Sort: aggregate values by keys

Reduce
- reducer
- All values associated with the same key

Output Files
- X 5
- Y 7
- Z 9

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
A Simple Word Count Example

Count the number of occurrences of every word in a text collection.

```java
1: class MAPPER
2:    method MAP(docid a, doc d)
3:        for all term t ∈ doc d do
4:            EMIT(term t, count 1)

1: class REDUCER
2:    method REDUCE(term t, counts [c₁, c₂, ...])
3:        sum ← 0
4:        for all count c ∈ counts [c₁, c₂, ...] do
5:            sum ← sum + c
6:        EMIT(term t, count sum)
```

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Combiner & Partitioner

A mini reducer in the map phase
Determines which key goes to which reducer

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Word Count with In-Mapper Combining

```java
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         H ← new ASSOCIATIVEARRAY
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             EMIT(term t, count H{t}) ▷ Tally counts for entire document

1: class REDUCER
2:     method REDUCE(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         EMIT(term t, count sum)
```

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Word Count with Improved In-Mapper Combining

```java
class MAPPER {
  method INITIALIZE {
    H ← new ASSOCIATIVEARRAY
  }
  method MAP(docid a, doc d) {
    for all term t ∈ doc d do
      H{t} ← H{t} + 1
      ▶ Tally counts across documents
  }
  method CLOSE {
    for all term t ∈ H do
      EMIT(term t, count H{t})
  }
}

class REDUCER {
  method REDUCE(term t, counts [c_1, c_2, ...]) {
    sum ← 0
    for all count c ∈ counts [c_1, c_2, ...] do
      sum ← sum + c
    EMIT(term t, count sum)
  }
}
```

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Compute Mean of Values Associated with Each Key

```java
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, integer r)

1: class REDUCER
2:     method REDUCE(string t, integers [r₁, r₂, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r₁, r₂, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:             r_{avg} ← sum/cnt
9:             EMIT(string t, integer r_{avg})
```

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Mean of Values with a Separate Combiner

```
1: class MAPPER
2:    method MAP(string t, integer r)
3:        EMIT(string t, integer r)

1: class COMBINER
2:    method COMBINE(string t, integers [r₁, r₂, ...])
3:        sum ← 0
4:        cnt ← 0
5: for all integer r ∈ integers [r₁, r₂, ...] do
6:    sum ← sum + r
7:    cnt ← cnt + 1
8: EMIT(string t, pair (sum, cnt)) ▷ Separate sum and count

1: class REDUCER
2:    method REDUCE(string t, pairs [(s₁, c₁), (s₂, c₂) ...])
3:    sum ← 0
4:    cnt ← 0
5: for all pair (s, c) ∈ pairs [(s₁, c₁), (s₂, c₂) ...] do
6:    sum ← sum + s
7:    cnt ← cnt + c
8:  r_{avg} ← sum/cnt
9: EMIT(string t, integer r_{avg})
```

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Mean of Values with a Separate Combiner

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, pair (r, 1))

1: class COMBINER
2:     method COMBINE(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         EMIT(string t, pair (sum, cnt))

1: class REDUCER
2:     method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         EMIT(string t, integer r_avg)
```

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Mean of Values with an In-Mapper Combiner

```java
1: class Mapper
2:   method initialize
3:     S ← new AssociativeArray
4:     C ← new AssociativeArray
5:   method map(string t, integer r)
6:     S{t} ← S{t} + r
7:     C{t} ← C{t} + 1
8:   method close
9:     for all term t ∈ S do
10:        emit(term t, pair (S{t}, C{t}))
1: class Reducer
2:   method reduce(string t, pairs \[(s_1, c_1), (s_2, c_2)\ldots\])
3:     sum ← 0
4:     cnt ← 0
5:     for all pair (s, c) ∈ pairs \[(s_1, c_1), (s_2, c_2)\ldots\] do
6:        sum ← sum + s
7:        cnt ← cnt + c
8:     r_{avg} ← sum/cnt
9:     emit(string t, integer r_{avg})

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Computing Word Co-occurrences

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Word Co-occurrences (Stripes Approach)

```java
1: class MAPPER
2:   method MAP(docid a, doc d)
3:     for all term w ∈ doc d do
4:       H ← new ASSOCIATIVEARRAY
5:       for all term u ∈ NEIGHBORS(w) do
6:         H{u} ← H{u} + 1
7:       EMIT(Term w, Stripe H)

1: class REDUCER
2:   method REDUCE(term w, stripes [H₁, H₂, H₃,...])
3:     H_f ← new ASSOCIATIVEARRAY
4:     for all stripe H ∈ stripes [H₁, H₂, H₃,...] do
5:       SUM(H_f, H)
6:     EMIT(term w, stripe H_f)
```

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Baseline Inverted Indexing for Text Retrieval

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Baseline Inverted Indexing for Text Retrieval

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Scalable Inverted Indexing for Text Retrieval

```java
1: class MAPPER
2:  method MAP(docid n, doc d)
3:     H ← new ASSOCIATIVEARRAY
4:     for all term t ∈ doc d do
5:         H{t} ← H{t} + 1
6:     for all term t ∈ H do
7:         EMIT(tuple ⟨t, n⟩, tf H{t})

1: class REDUCER
2:  method INITIALIZE
3:     t_prev ← ∅
4:     P ← new POSTINGSLIST
5:  method REDUCE(tuple ⟨t, n⟩, tf [f])
6:     if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:         EMIT(term t, postings P)
8:         P.RESET()
9:     P.ADD(⟨n, f⟩)
10:    t_prev ← t
11:  method CLOSE
12:    EMIT(term t, postings P)
```

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Parallel Breadth-First Search

1: class MAPPER
2:   method MAP(nid n, node N)
3:       d ← N.DISTANCE
4:       EMIT(nid n, N)                      ▶ Pass along graph structure
5:       for all nodeid m ∈ N.ADJACENCYLIST do
6:           EMIT(nid m, d + 1)              ▶ Emit distances to reachable nodes

1: class REDUCER
2:   method REDUCE(nid m, [d₁, d₂, …])
3:       dₘᵟᵢᵡ ← ∞
4:       M ← ∅
5:       for all d ∈ counts [d₁, d₂, …] do
6:           if ISNODE(d) then
7:               M ← d                           ▶ Recover graph structure
8:           else if d < dₘᵟᵢᵡ then
9:               dₘᵟᵢᵡ ← d                           ▶ Look for shorter distance
10:      M.DISTANCE ← dₘᵟᵢᵡ                   ▶ Update shortest distance
11:     EMIT(nid m, node M)

Source: Lin & Dyer, “Data-Intensive Text Processing with MapReduce”
Hadoop Subprojects

Source: Tom White, “Hadoop – The Definitive Guide”

**Core:** A set of components and interfaces for distributed file systems and general I/O (serialization, Java RPC, persistent data structures).

**Avro:** A data serialization system for efficient, cross-language RPC, and persistent data storage.

**MapReduce:** A distributed data processing model and execution environment that runs on large clusters of commodity machines.

**HDFS:** A distributed filesystem that runs on large clusters of commodity machines.
Hadoop Subprojects

Source: Tom White, “Hadoop – The Definitive Guide”

**Pig**: A data flow language and execution environment for exploring very large datasets. Pig runs on HDFS and MapReduce clusters.

**HBase**: A distributed, column-oriented database. HBase uses HDFS for its underlying storage, and supports both batch-style computations using MapReduce and point queries (random reads).

**ZooKeeper**: A distributed, highly available coordination service. ZooKeeper provides primitives such as distributed locks that can be used for building distributed applications.
**Hadoop Subprojects**

Source: Tom White, "Hadoop – The Definitive Guide"

**Hive:** A distributed data warehouse. Hive manages data stored in HDFS and provides a query language based on SQL (and which is translated by the runtime engine to MapReduce jobs) for querying the data.

**Chukwa:** A distributed data collection and analysis system. Chukwa runs collectors that store data in HDFS, and it uses MapReduce to produce reports.
The Building Blocks of Hadoop

On a fully configured Hadoop cluster a set of daemons or resident programs run on the different servers in the network.

- NameNode
- DataNode
- Secondary NameNode
- JobTracker
- TaskTracker
**The Building Blocks of Hadoop**

**NameNode:** The bookkeeper of HDFS: keeps track of how files are broken down into file blocks, which nodes store those blocks, and the overall health of the distributed filesystem.

**DataNode:** Each slave machine in the cluster hosts a DataNode daemon to perform the reading and writing of HDFS blocks to actual files on the local filesystem.

Source: Chuck Lam, “Hadoop in Action”
The Building Blocks of Hadoop

**JobTracker:** Determines the execution plan for a job by determining which files to process, assigns nodes to different tasks, and monitors all tasks as they’re running. Should a task fail, the JobTracker will automatically relaunch the task, possibly on a different node.

**TaskTracker:** Manages the execution of individual (map or reduce) tasks on each slave node.
**Secondary NameNode:** It communicates with the NameNode to take periodic snapshots of the HDFS metadata. Does not keep track of any real-time changes to HDFS. Can be configured to work as the NameNode in the event of the failure of the original NameNode.
Hadoop Distributed File System (HDFS) Design

HDFS was designed for

– Very large files
– Streaming data access
– Commodity hardware

But not for

– Low latency access
– Lots of small files
– Multiple writes, arbitrary file modifications
Hadoop MapReduce

Source: Tom White,
“Hadoop – The Definitive Guide”
An Example: Mining Weather Data
Find Maximum Temperature Every Year

Source: Tom White,
“Hadoop – The Definitive Guide”
Maximum Temperature Every Year (Java)

```
public class NewMaxTemperature {

    static class NewMaxTemperatureMapper
        extends Mapper<LongWritable, Text, Text, IntWritable> {
        ...
    }

    static class NewMaxTemperatureReducer
        extends Reducer<Text, IntWritable, Text, IntWritable> {
        ...
    }

    public static void main(String[] args) throws Exception {
        ...
    }
}
```

Source: Tom White, “Hadoop – The Definitive Guide”
Maximum Temperature Every Year (Java)

Source: Tom White, “Hadoop – The Definitive Guide”

```java
static class NewMaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {
    private static final int MISSING = 9999;

    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String line = value.toString();
        String year = line.substring(15, 19);
        int airTemperature;
        if (line.charAt(87) == '+') {
            // parseInt doesn't like leading plus signs
            airTemperature = Integer.parseInt(line.substring(88, 92));
        } else {
            airTemperature = Integer.parseInt(line.substring(87, 92));
        }
        String quality = line.substring(92, 93);
        if (airTemperature != MISSING && quality.matches("[01459]")) {
            context.write(new Text(year), new IntWritable(airTemperature));
        }
    }
}
```
Maximum Temperature Every Year (Java)

Source: Tom White, “Hadoop – The Definitive Guide”

```java
static class NewMaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values,
            Context context)
            throws IOException, InterruptedException {

        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }

        context.write(key, new IntWritable(maxValue));
    }
}
```
Maximum Temperature Every Year (Java)

```
public static void main(String[] args) throws Exception {
    if (args.length != 2) {
        System.err.println("Usage: NewMaxTemperature <input path> <output path>");
        System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(NewMaxTemperature.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setMapperClass(NewMaxTemperatureMapper.class);
    job.setReducerClass(NewMaxTemperatureReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Source: Tom White, “Hadoop – The Definitive Guide”
Maximum Temperature Every Year (Python)

Source: Tom White, “Hadoop – The Definitive Guide”
Maximum Temperature Every Year (C++)

Source: Tom White, “Hadoop – The Definitive Guide”
Maximum Temperature Every Year (C++)

Source: Tom White, “Hadoop – The Definitive Guide”

class MapTemperatureReducer : public HadoopPipes::Reducer {
public:
    MapTemperatureReducer(HadoopPipes::TaskContext& context) {}
    void reduce(HadoopPipes::ReduceContext& context) {
        int maxVal = INT_MIN;
        while (context.nextValue()) {
            maxVal = std::max(maxVal, HadoopUtils::toInt(context.getInputValue()));
        }
        context.emit(context.getKeyValue(), HadoopUtils::toString(maxVal));
    }
};

int main(int argc, char *argv[]) {
    return HadoopPipes::runTask(HadoopPipes::TemplateFactory<MaxTemperatureMapper, MapTemperatureReducer>(),
}