MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

MapReduce implementationer

MapReduce

Hadoop

Apache open source projekt

Parallelle Programmer

- Traditionelt specielt udviklede programmer for hvert problem man ønsker at løse
  - mange ikke-trivielle detaljer omkring parallel programmer
  - fejl-tolerance (ved 1000'er af maskiner fejler maskiner regelmæssigt)
  - fordeling af data blandt maskiner
  - balancering af arbejdet blandt maskiner
- **MapReduce** interfacet (Google™, 2003)
  - håndter ovenstående automatisk
  - meget begrænsede kommunikation mellem maskiner
  - algoritmen udføres i Map og Reduce faser

MapReduce

- Brugeren skal definere to funktioner

map (k1,v1) → List ((k2,v2))

reduce (k2,List (v2)) → List (v2)

- Eksempel: Antal forekomster af ord i en tekstsamling

map
- (”www.foo.com”, ”der var en gang en...”) → (”der”, ”1”), (”var”, ”1”), (”en”, ”1”), (”gang”, ”1”), (”en”, ”1”)...
- (”www.bar.com”, ”en lang gang...” → (”en”, ”1”), (”lang”, ”1”), (”gang”, ”1”), ...

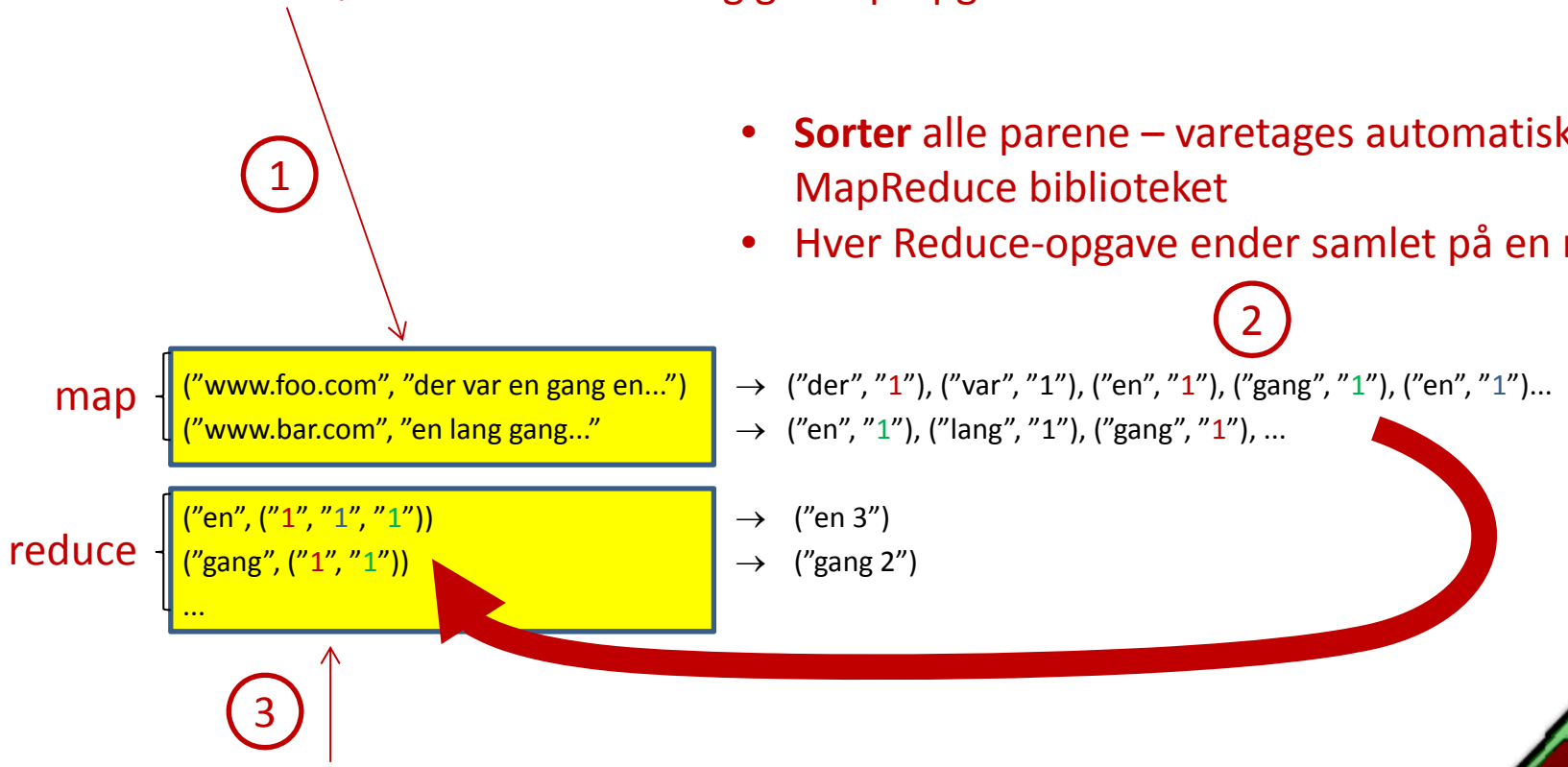
reduce
- (”en”, (”1”, ”1”, ”1”)) → (”en 3”)
- (”gang”, (”1”, ”1”)) → (”gang 2”)
- ...

- Output fra en map-reduce kan være input til den næste map-reduce

MapReduce

Master = en maskine der
1. planlægger og fordeler opgaverne
2. genstarter døde/hængene opgaver på andre maskiner

- Map-opgaverne ligger spredt ud på maskinerne (i et GFS hvor data typisk er replikleret 3 gange)
- Hver maskine udfører et antal uafhængige map-opgaver



- **Sorter** alle parene – varetages automatisk a MapReduce biblioteket
- Hver Reduce-opgave ender samlet på en maskine

- En reduce-opgave løses sekventielt på én maskine (mulig FLASKEHALS)
- Hver maskine udfører et antal uafhængige reduce-opgaver
- Output er en delliste af det samlede output

En søgemaskines dele

Indsamling af data

- Webcrawling (gennemløb af internet)

Indeksering data

- Parsning af dokumenter
- Leksikon: indeks (ordbog) over alle ord mødt
- Inverteret fil: for alle ord i leksikon, angiv i hvilke dokumenter de findes

Søgning i data

- Find alle dokumenter med søgeordene
- Rank dokumenterne

Inverteret fil

Input: List ((URL, tekst))

Output: List ((Ord, URL'er))

Map

(URL, tekst) $\rightarrow$ (ord_1, (ord_1, URL)),..., (ord_k, (ord_k, URL))

Reduce

(ord, ((ord, URL_1),...,(ord, URL_m))) $\rightarrow$ ((ord, URL_1+⋯+ URL_m))

Indgrad af alle siderne

Input: List ((*i*, *j*))

Output: List ((*i*, indgrad(*i*)))

Map

$(i, j) \rightarrow (\, j, (j, 1)\,)$

Reduce

$(i, \underbrace{((i, 1), (i, 1), \ldots, (i,1))}_{k = \text{indgrad}(i)}) \rightarrow (\, (i, k)\,)$

Sum

Input: $((1, x_1),\ldots,(n, x_n))$

Ouput: $(\ \text{sum}(x_1,\ldots, x_n)\ )$

1

Map

$(i, x_i) \rightarrow (\text{random}(1..R), (i, x_i))$

Reduce

$(r, ((i_1, x_{i_1}),\ldots, (i_k,x_{i_k}))) \rightarrow (\ (r, x_{i_1}+\cdots+ x_{i_k})\ )$

$R \approx$ antal maskiner

2

Map

$(i, x_i) \rightarrow (0, x_i)$

Reduce

$(0, (x_1,\ldots, x_n\ )) \rightarrow (\ x_1+\cdots+ x_n\ )$

Afleveringsopgave: PageRank

Input: List((i, j))

Ouput: List($(i, p_i^{(s)})$)

$$p_i^{(s)} = 0.85 \cdot \sum_{j:j\to i} \frac{p_j^{(s-1)}}{\text{udgrad}(j)} + 0.15 \cdot \frac{1}{n}$$

$$p_1^{(0)} = 1.0 \qquad p_2^{(0)} = \cdots = p_n^{(0)} = 0.0$$