CS 240
Computer Systems

#18: S3 APIs and MapReduce Overview

March 29, 2022 · Wade Fagen-Ulmschneider

Cloud Object Storage

Instead of using file storage on disk, object storage in the cloud provides us access to a file-system-like interface without the need for all programs to be running on the same computer!

Reading a file in Python:

18/local.py

```python
f = open("settings.json", "r")
print(f.read())
```

Initializing an S3 connection:

18/s3.py

```python
import boto3
s3 = boto3.client('s3', [...])
```

Reading Data from S3:

18/s3.py

```python
# Reading data from S3:
obj = s3.get_object(Bucket="cs240", Key="session_data")
f = obj["Body"]
```

- The **f** variable in local.py and s3.py are both _______________!
- **Key Idea:**

18/s3.py

```python
print("== S3 Response ==")
print(obj)
print()

print("== Contents ==")
print(f.read().decode("utf-8"))
print()
```

Writing Data to S3 is one line just like files to disk:

18/s3-put.py

```python
# Add an object as a string:
s3.put_object(Bucket="cs240", Key="session_data",
                  Body=json.dumps({"hello": "world"}))

# Upload a file:
s3.upload_file("cs240.png", Bucket="cs240",
                      Key="profile-picture.png")
```

EC Scavenger Hunt on S3:

MapReduce

- Developed as a research project out of Google.
- OSDI'04: *"MapReduce: Simplified Data Processing on Large Clusters"*
- **Big Idea:** Create a framework for processing data based on functions that can be "automatically parallelized".
    - Allows many nodes to contribute to processing the data without human design/programming.

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

1

https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

MapReduce: Map Functions

- Input:
- Output:

Reduce Function:

- Input:
- Output:

Example #1: Word Count

The	*quick*	*brown*	*fox*	*jumps*	*over*	*the*	*lazy*	*dog*
[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**

Map:

Reduce:

Example #2: Mutual Friends

Through asking about your friends about their friends, you have identified who are friends of whom (→ means *"is friends with"*):

```
A → B, C
B → A, C, D
C → A, B, D
D → B, C
```

You want to identify all **mutual friends** to any set of two people. For example: {A, B} → C, D.

Map:

Reduce: