# HBASE

# Non-relational, scalable database built on HDFS

# Based on Google's BigTable

**Bigtable: A Distributed Storage System for Structured Data**

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
*Google, Inc.*

**Abstract**

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

## 1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

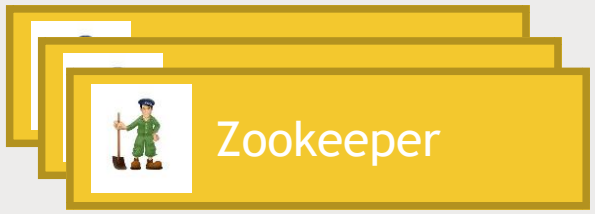
## 2 Data Model

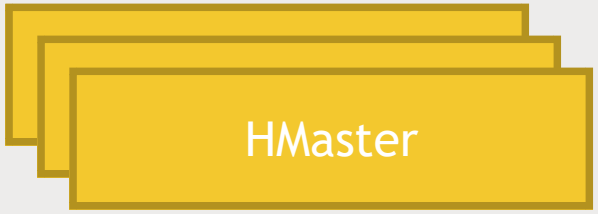
A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes

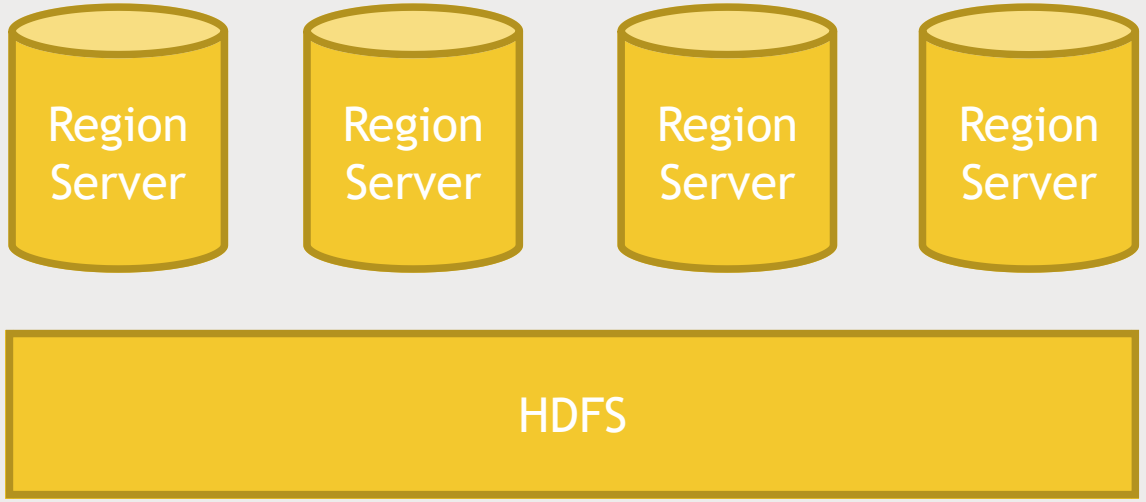
# CRUD

- Create
- Read
- Update
- Delete
- There is no query language, only CRUD API’s!

# HBase architecture




Auto-sharding!

# HBase data model

- Fast access to any given ROW
- A ROW is referenced by a unique KEY
- Each ROW has some small number of COLUMN FAMILIES
- A COLUMN FAMILY may contain arbitrary COLUMNS
- You can have a very large number of COLUMNS in a COLUMN FAMILY
- Each CELL can have many VERSIONS with given timestamps
- Sparse data is A-OK – missing columns in a row consume no storage.

# Example: One row of a web table

Key

com.cnn.www

Contents column family

Contents:

<html><head>
CNN…

Anchor column family

Anchor:cnnsi.com

"CNN"

Anchor:my.look.ca

"CNN.com"

# Some ways to access HBase

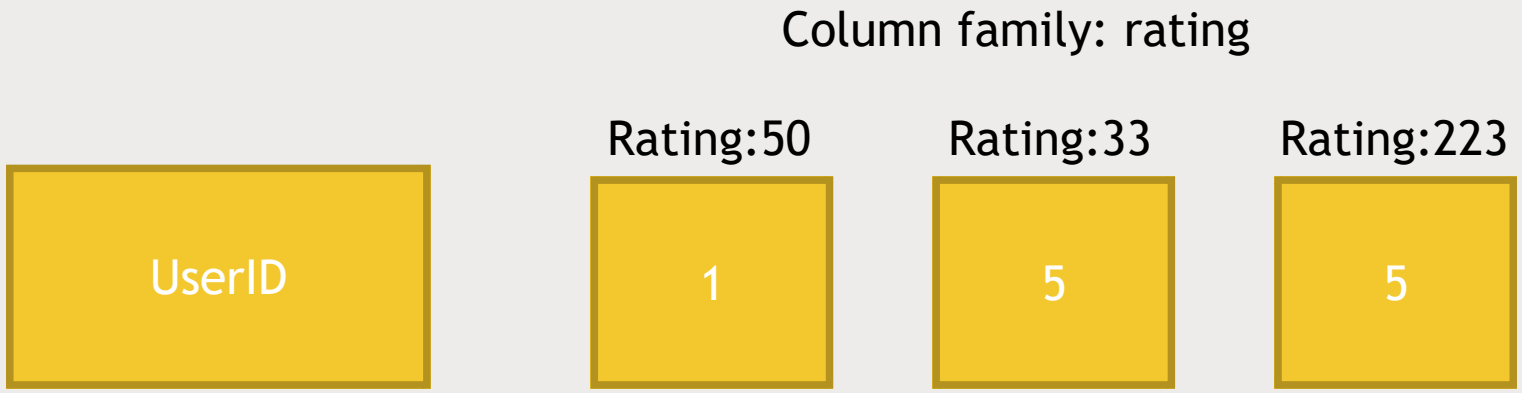
- HBase shell
- Java API
  - *Wrappers for Python, Scala, etc.*
- Spark, Hive, Pig
- REST service
- Thrift service
- Avro service

# LET'S PLAY WITH HBASE

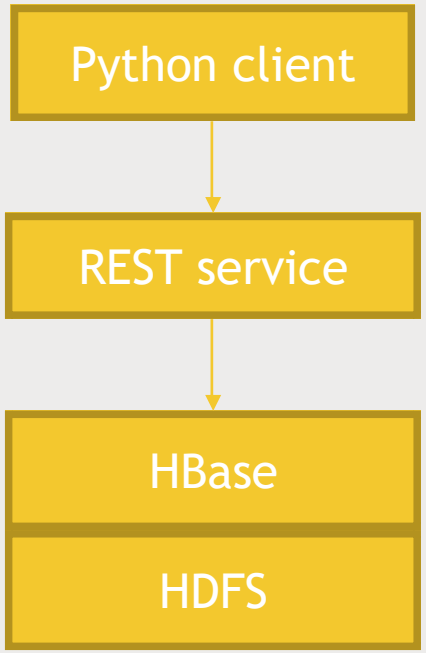
Creating a HBase table with Python via REST

# What are we doing?

- Create a HBase table for movie ratings by user
- Then show we can quickly query it for individual users
- Good example of sparse data



# How are we doing it?



# Let’s do this

# HBASE / PIG INTEGRATION

Populating HBase at scale

# Integrating Pig with HBase

- Must create HBase table ahead of time
- Your relation must have a unique key as its first column, followed by subsequent columns as you want them saved in Hbase
- USING clause allows you to STORE into an HBase table
- Can work at scale – Hbase is transactional on rows

# Let’s do this