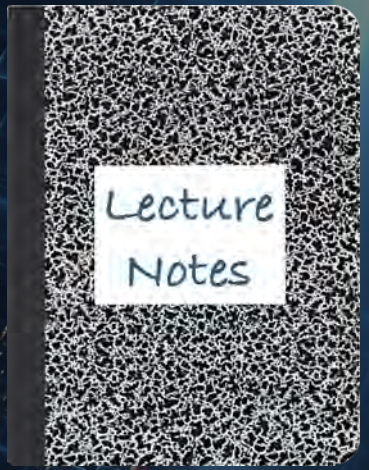
CS 417 – DISTRIBUTED SYSTEMS

Week 9: Distributed Databases
Part 2: Bigtable, Cassandra



Paul Krzyzanowski

The Scale Problem & ACID Costs

- Workloads grew past the scale of one machine
- Scaling traditional relational databases systems worked for a while
  - But not for petabytes of data and high throughput

- Maintaining ACID semantics becomes costly

Atomicity	**Consistency**	**Isolation**	**Durability**
2PC overhead	Enforcing integrity	Distributed locking	Write latency

Relax the model or deal with performance problems

Wide-Column Databases: A New Model

Relational

- Fixed schema shared by all rows
- Strong support for joins and constraints
- Storage engines organized around tables, pages, and indexes

Wide-Column

- Sparse rows in an ordered key space
- Different rows may contain different columns
- Often organized into column families

Google Bigtable (& Apache HBase)

Bigtable

- Highly available distributed storage

- Built with semi-structured data in mind
  - URLs: content, metadata, links, anchors, page rank
  - User data: preferences, account info, recent queries
  - Geography: roads, satellite images,
    points of interest, annotations

- Large scale
  - Petabytes of data
    … across 450,000+ of servers (NYT estimaet 2006)
  - Billions of URLs with many versions per page
  - Hundreds of millions of users
  - Thousands of queries per second
  - 100TB+ satellite image data

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 many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have 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 multidimensional 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.

(row:string, column:string, time:int64) → string

To appear in OSDI 2006

Defining Bigtable

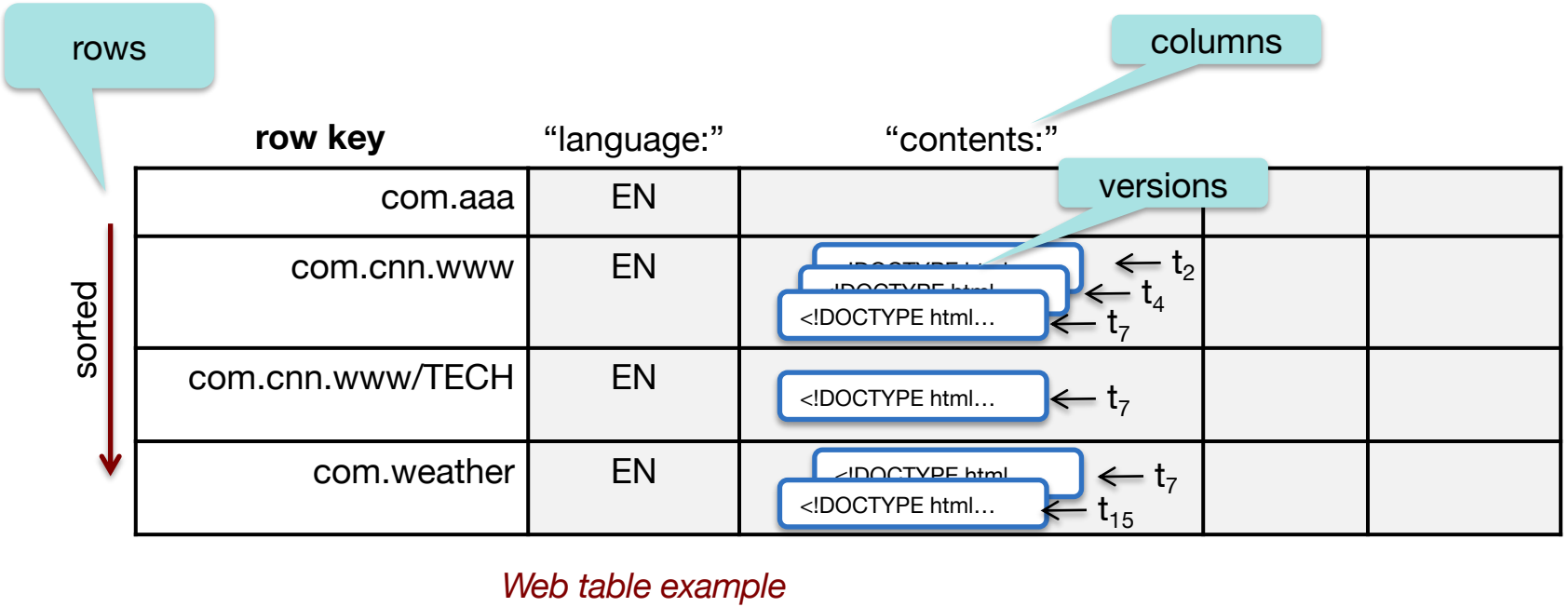
"a sparse, distributed, persistent multidimensional sorted map indexed by row key, column key, and timestamp"

- **Sparse**: Different rows can have different sets of columns
- **Sorted**: Sorted by row keys; row key order is important
- **Versioned**: timestamp-based version history per cell

Data Model: Row keys, Column Families, Qualifiers, Versions

(row, column family, [qualifier], timestamp) → cell contents

- Contents are arbitrary strings (arrays of bytes)



Web table example

Timestamps and Versions

Each cell may contain multiple versions of data

- **Version indexed by a 64-bit timestamp**
  - Real time or assigned by client
- **Per-column-family settings for garbage collection**
  - Keep only latest *n* versions
  - Or keep only versions written since time *t*
- Retrieve most recent version if no version specified
  - If specified, return version where timestamp ≤ requested time

Example: web crawl retains the last 3 versions of each page's HTML

Column Families: Example

Sports Illustrated contains
`<a href="www.cnn.com">CNN</a>`

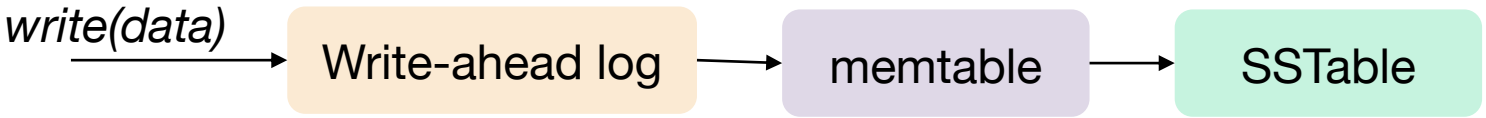
Three column families
- "language:" – language for the web page
- "contents:" – contents of the web page
- "anchor:" – contains text of anchors that reference this page
  - www.cnn.com is referenced by Sports Illustrated (cnnsi.com) and My-Look (mlook.ca)
  - The value of ("com.cnn.www", "anchor:cnnsi.com") is "CNN", the reference text from cnnsi.com.

Column family *anchor*

row key	"language:"	"contents:"	anchor:cnnsi.com	anchor:mylook.ca
com.aaa	EN	<!DOCTYPE html PUBLIC…		
com.cnn.www	EN	<!DOCTYPE HTML PUBLIC…	"CNN"	"CNN.com"
com.cnn.www/TECH	EN	<!DOCTYPE HTML>…		
com.weather	EN	<!DOCTYPE HTML>…		

sorted

Writing Data: Memtable and SSTables



Reads: check memtable and ALL on-disk SSTables

- Both are sorted, so merging is efficient

Periodic compaction: merge SSTables

Tablets: Scaling Bigtable – Splitting

	"language:"	"contents:"		
com.aaa	EN	<!DOCTYPE html PUBLIC…		
com.cnn.www	EN	<!DOCTYPE HTML PUBLIC…		
com.cnn.www/TECH	EN	<!DOCTYPE HTML>…		
com.weather	EN	<!DOCTYPE HTML>…		
com.wikipedia	EN	<!DOCTYPE HTML>…		
com.zcorp	EN	<!DOCTYPE HTML>…		
com.zoom	EN	<!DOCTYPE HTML>…		

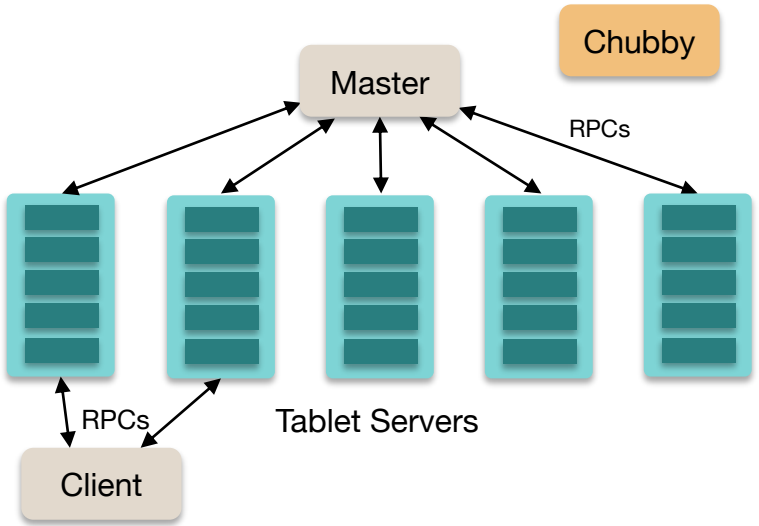
Split

One master, many tablet servers

1. **Many tablet servers – coordinate requests to tablets**
   - Can be added or removed dynamically
   - Each manages a set of tablets (typically 10-1,000 tablets/server)
   - Handles read/write requests to tablets
   - Splits tablets when too large

2. **One master server**
   - Assigns tablets to tablet server
   - Balances tablet server load
   - Garbage collection of unneeded SSTable files
   - Schema changes (table & column family creation)

3. **Client library**
   - Client data does not move through the master
   - Clients communicate directly with tablet servers for reads/writes



Cassandra

Facebook Origins: Motivation

- Initial Problem: Facebook Inbox Search
  - High write throughput, low latency, no single point of failure
  - Support hundreds of millions of users
- Bigtable's master was a bottleneck and single point of failure

Facebook Origins: Motivation

- Initial Problem: Facebook Inbox Search
  - High write throughput, low latency, no single point of failure
  - Support hundreds of millions of users
- Bigtable's master was a bottleneck and single point of failure

From Dynamo Partitioning, replication, gossip, tunable consistency

+

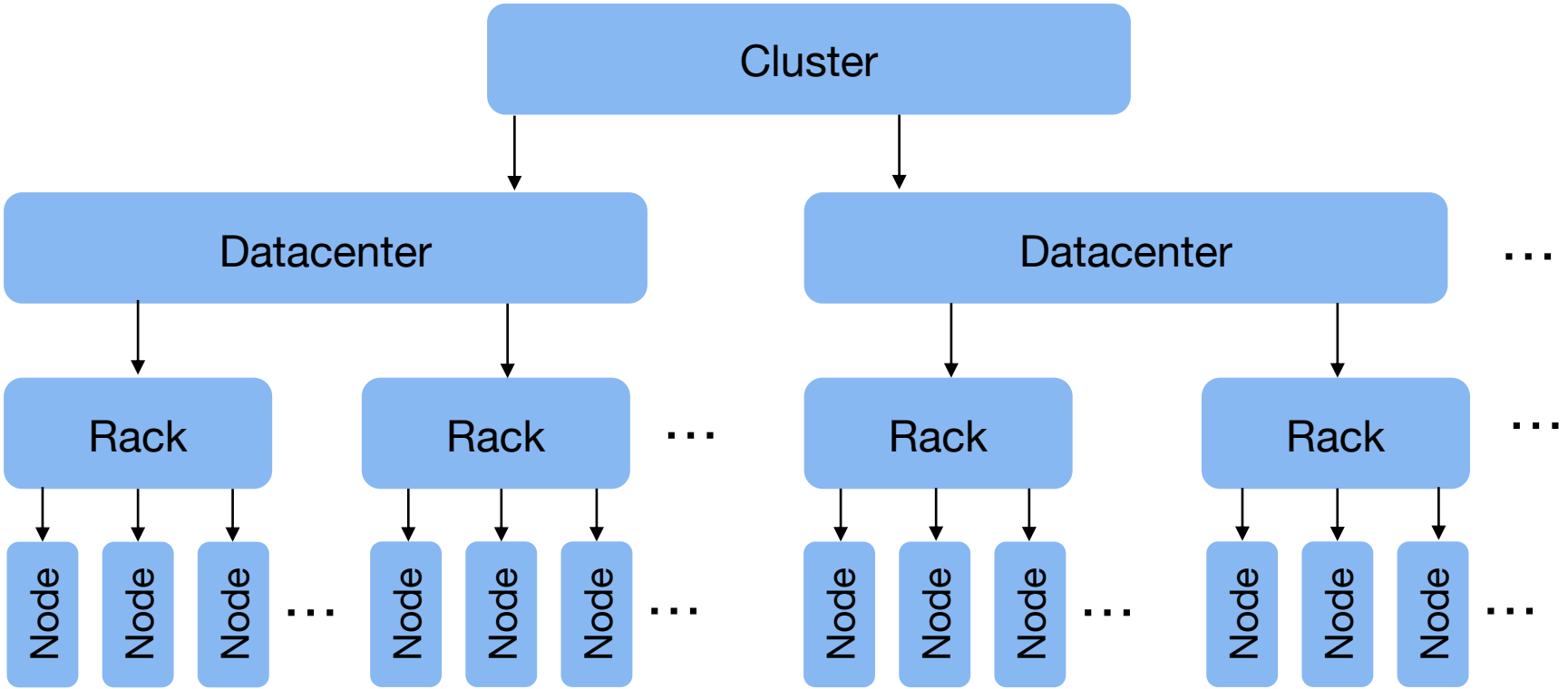
From Bigtable Column families, memtables, SSTables

Masterless, wide-column, distributed, NoSQL database

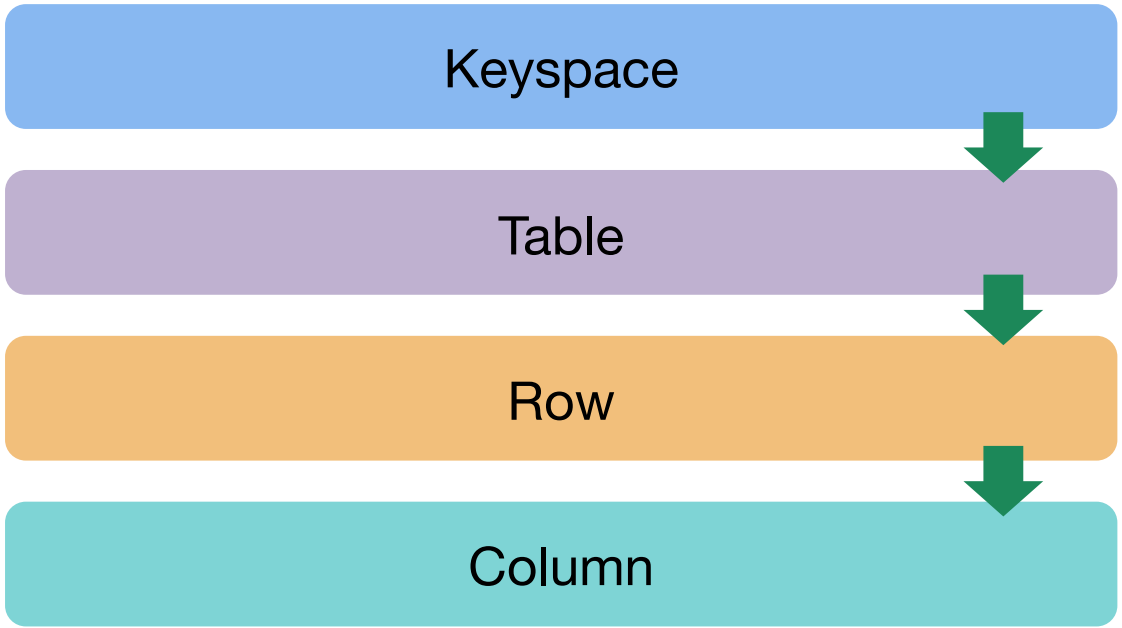
Cassandra Design Goals

- High availability – no central coordinator
- Low latency
- Commodity hardware
- Linear performance increase with additional nodes
- Tunable consistency
- Key-oriented queries
- Flexible data model
- SQL-like query language (CQL - Cassandra Query Language)

Machine hierarchy



Data Hierarchy



Top-level namespace

Collection of sorted rows

Key and one or more columns

{ key, value, timestamp }

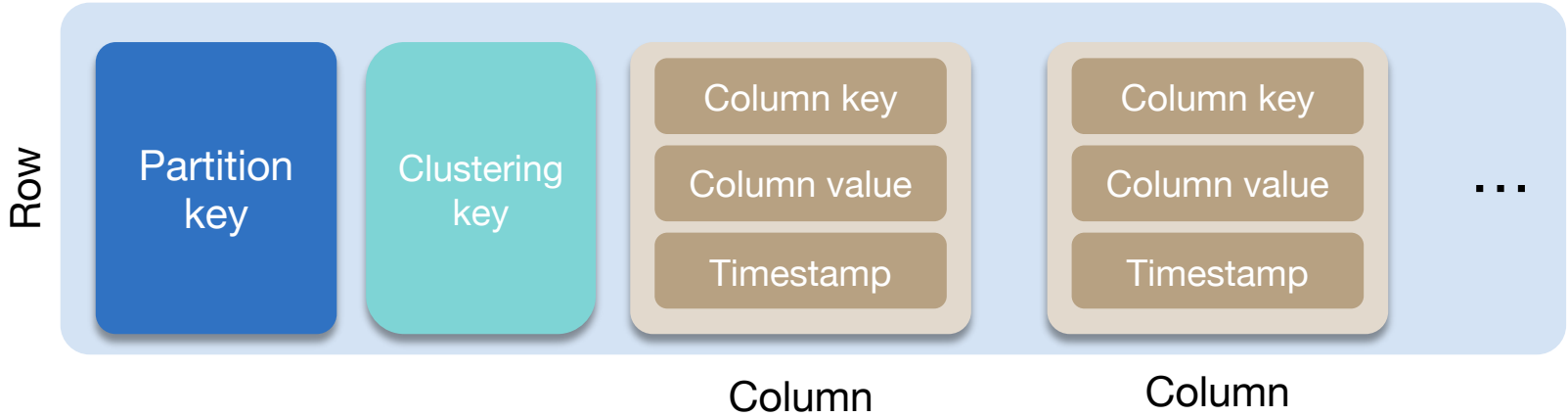
Data Model: Keyspace, Partition, Row, Column

Keyspace: Top-level namespace

Partition: All rows with the same partition key – determines placement

Clustering key: Identifies the sorting sequence for rows

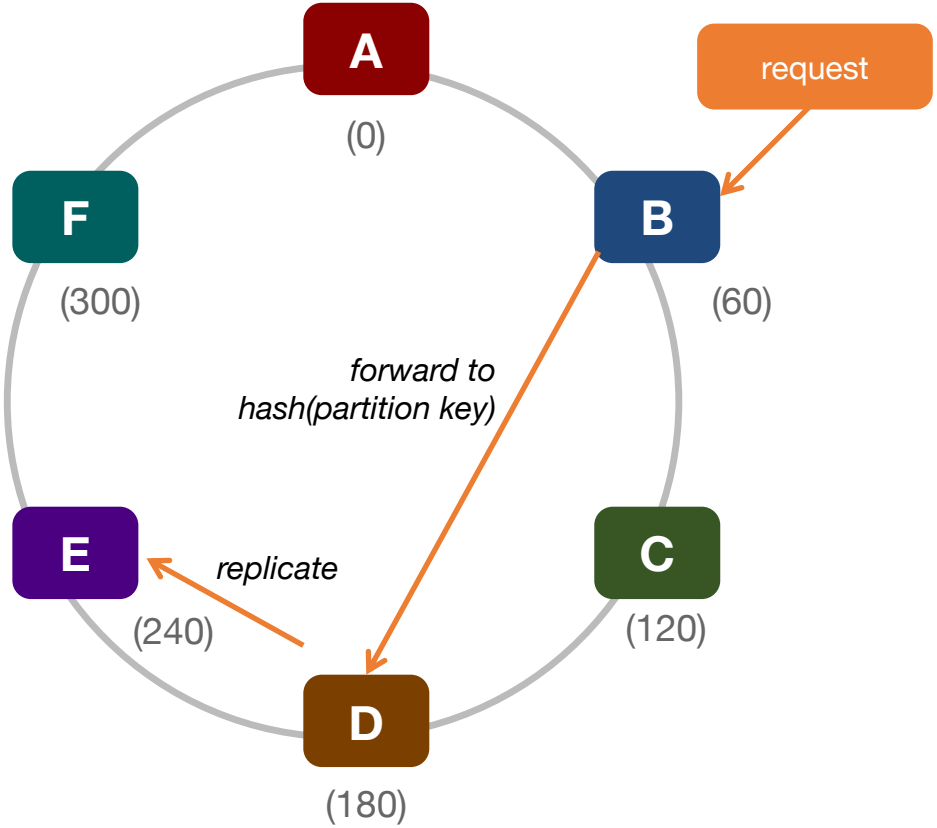
Row: Ordered collection of columns identified by [partition key, row key]



Consistent Hashing in Cassandra

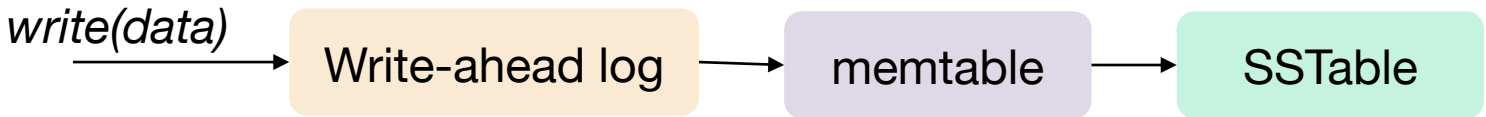
No master node

- Hash(partition key)
- Node owns all values from predecessor to self
- Replication factor
  - Copies to next N nodes
- Adding a node only impacts adjacent nodes



Write & read operations

- Data first written to a commit log – then the *write* may be acknowledged
  - This is a crash recovery mechanism
- Data added to memtable at the node
- Periodically, the memtable is written to an **SSTable** disk structure
- The commit log is purged once the data is written



Reads: combine results from memtable and possibly multiple SSTables

Tunable Consistency

Per-operation consistency level:
how many replicas must respond before the operation succeeds

Fixed count	1, 2, or 3 replicas	low latency; stale reads possible
Quorum-based	• Majority across cluster • Local data center • Quorum in each	balance consistency & availability
All	all replicas	strongest consistency

Who Uses Cassandra?

- **Activision**: Player messaging
- **Apple**: iCloud storage infrastructure
  - Over 75,000 Cassandra nodes, storing more than 10 petabytes
- **Best Buy**: Managing spikes in holiday traffic: 50,000 requests/second
- **Instagram**: one of the world's largest Cassandra deployments
- **CERN**: large time series data to support 20,000 apps on 2,400 computers
- **eBay**: 200TB+ storage: 400M+ writes, 100M+ reads per day
- **Netflix**: Petabytes of data serving data in milliseconds
- **Bloomberg**, **Discord**, **Grubhub**, **Hulu**, **Home Depot**, **Spotify**, **Target**, **Uber**, Yelp

The End