Cassandra vs. Redis

NoSQL Frankfurt
2010-09-28

Who?

- Tim Lossen / **@tlossen**
- Ruby developer
- Berlin

wooga – world of gaming

www.wooga.com

wooga
world of gaming

Home Games Jobs About Support

wooga's 1st birthday
Celebrating 10 million monthly users

Monster World
Choose your monster family

Bubble Island
Top puzzle game played by millions

Read the CEO's letter »

wooga on Facebook

Like

wooga German start-up blog deutsche-startups.de visited us and published photos of their tour.
How do you like our office in Berlin?

Hausbesuch bei wooga :: deutsche-startups.de
www.deutsche-startups.de
Deutsche-startups – eine Art Infoservice fur die Web2.0-Branche. Hausbesuch bei wooga

Friday at 1:16am

wooga We added interviews from a few of our employees to find out what they like about working at wooga. Check it out:

Based in Berlin, wooga is the leading European social games developer.

We are hiring!

Currently, we are searching for:

- Senior Creative Producer (m/f)
- Vorstandsassistent (m/w) (m/w)
- Software Engineer – Graduate Position (m/f)

Challenge

Requirements

- backend for facebook game

Requirements

- backend for facebook game

- 1 mio. daily users
- 10 mio. total users
- 100 KB data per user

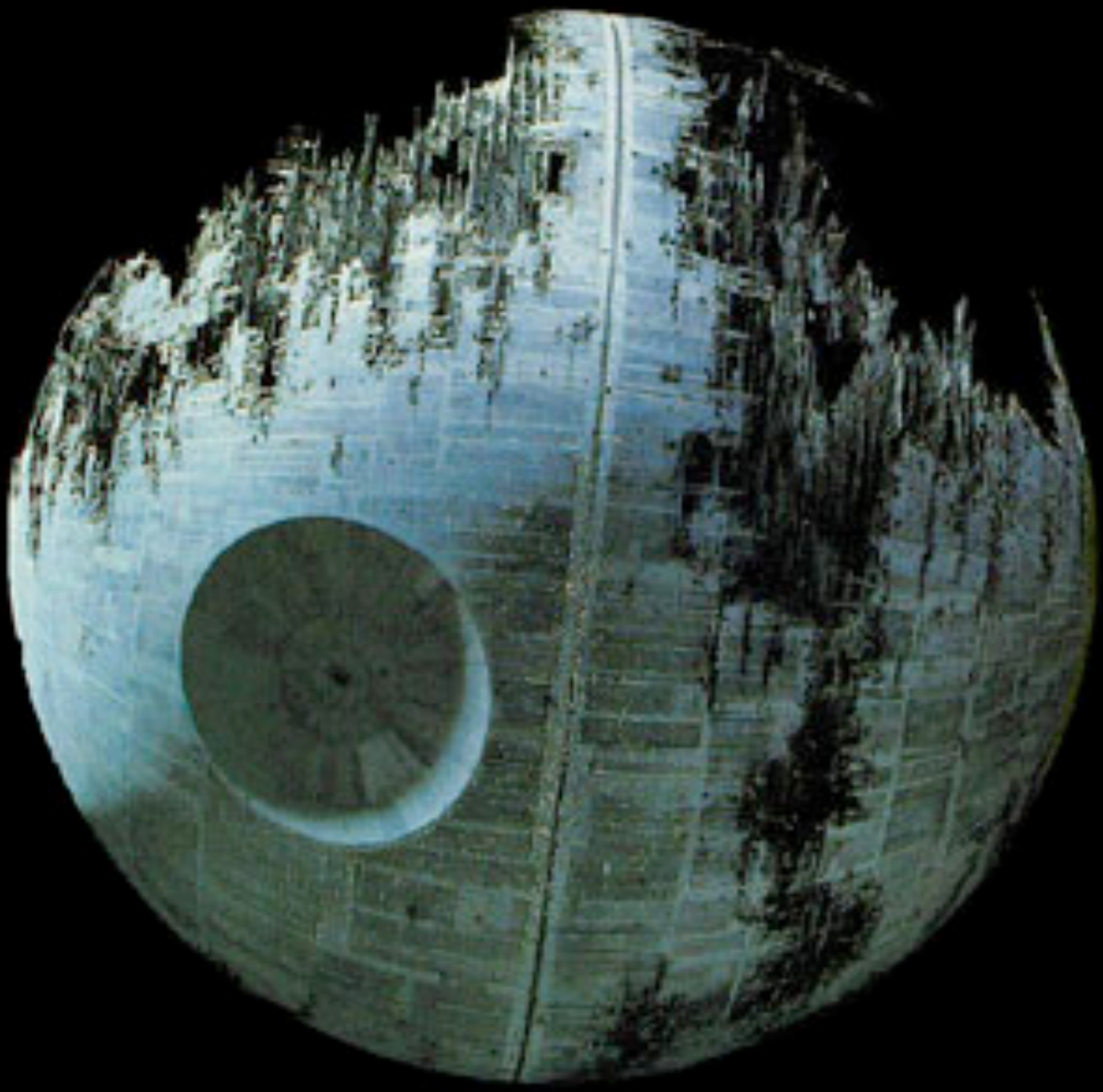
Requirements

- peak traffic:
  - 10.000 concurrent users
  - 200.000 requests per minute

Requirements

- peak traffic:
  - 10.000 concurrent users
  - 200.000 requests per minute
- write-heavy workload

Sneak Preview



Cassandra

Overview

- written in Java
  - 55.000 lines of code

Overview

- written in Java
  - 55.000 lines of code
- Thrift API
  - clients for Java, Python, Ruby

History

- originally developed by facebook
  - in production for “inbox search”

History

- originally developed by facebook
  - in production for “inbox search”
- later open sourced
  - top-level apache project

Features

- high availability
  - no single point of failure

Features

- high availability
  - no single point of failure
- incremental scalability

Features

- high availability
  - no single point of failure
- incremental scalability
- eventual consistency

2006

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. Bigtable has 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 pa-

2007

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management; D.4.5

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed

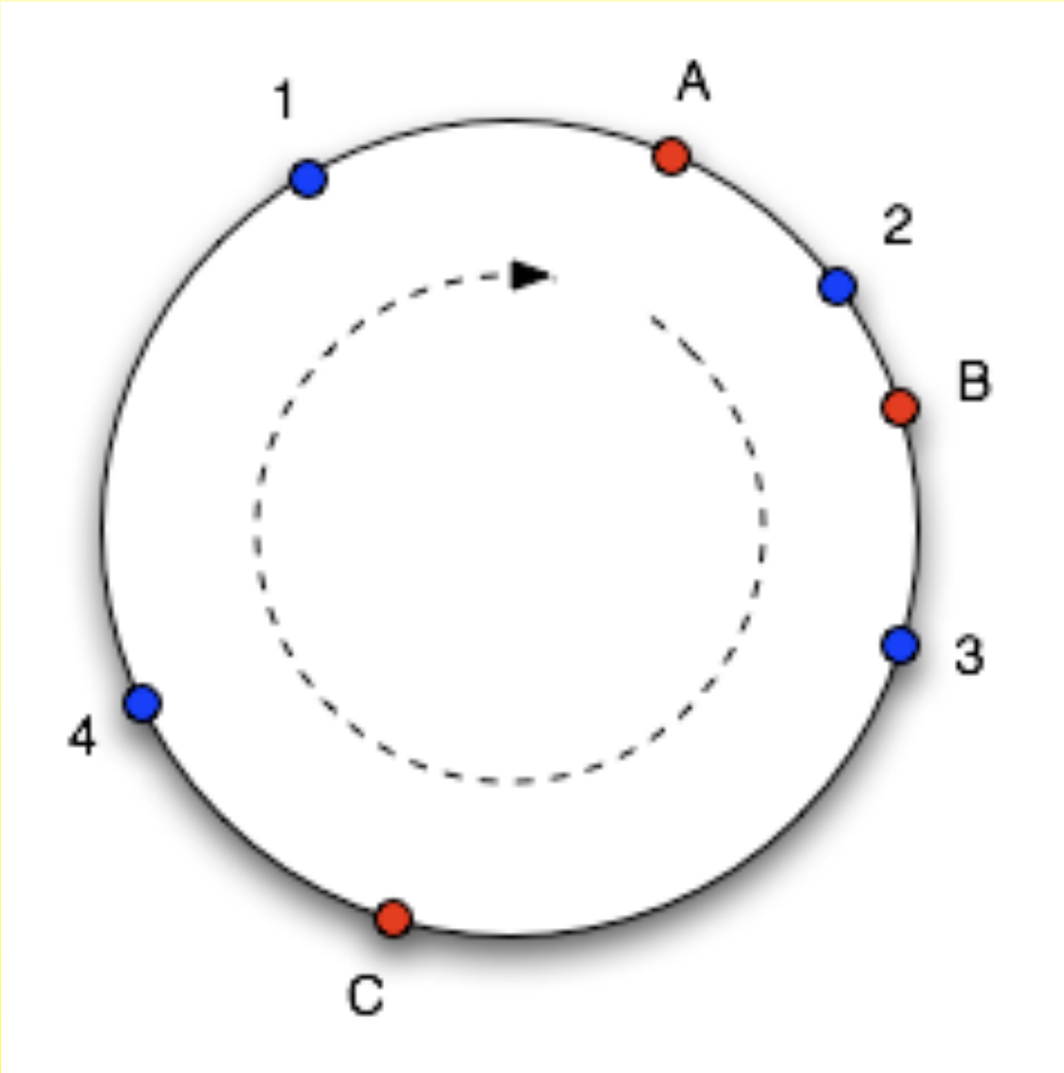
Architecture

- Dynamo-like ring
  - partitioning + replication
  - all nodes are equal

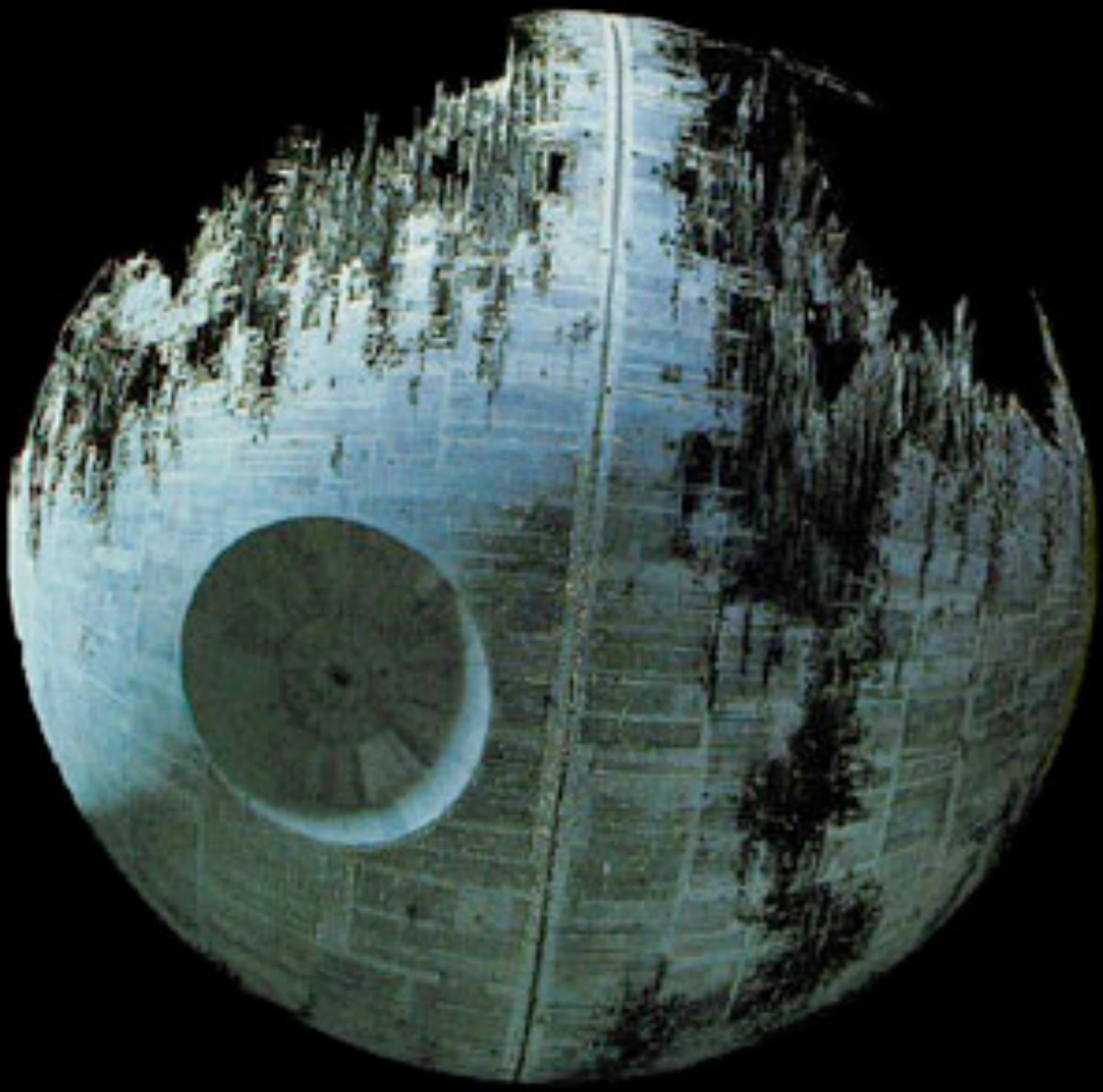
Architecture

- Dynamo-like ring
  - partitioning + replication
  - all nodes are equal
- Bigtable data model
  - column families

Cluster structure



"Cassandra aims to run on top of an infrastructure of hundreds of nodes."



Redis

Overview

- written in C
  - 13.000 lines of code

Overview

- written in C
  - 13.000 lines of code
- socket API
  - redis-cli
  - client libs for all major languages

Features

- high (write) throughput
  - 50 - 100 K ops / second

Features

- high (write) throughput
  - 50 - 100 K ops / second
- interesting data structures
  - lists, hashes, (sorted) sets
  - atomic operations

Features

- high (write) throughput
  - 50 - 100 K ops / second
- interesting data structures
  - lists, hashes, (sorted) sets
  - atomic operations
- full consistency

Architecture

- single instance (**not** clustered)
  - master-slave replication

Architecture

- single instance (**not** clustered)
  - master-slave replication
- in-memory database
  - append-only log on disk
  - virtual memory

"Memory is the new disk,
disk is the new tape."

—Jim Gray

Solution

Key Decisions

- keep operations simple

Key Decisions

- keep operations simple
- use as few machines as possible
  - ideally, only **one**

Architecture

- single Redis master
  - with virtual memory
  - handles all reads / writes

Architecture

- single Redis master
  - with virtual memory
  - handles all reads / writes
- single Redis slave
  - as hot standby (for failover)

Throughput

- redis-benchmark
  - 60 K ops / s = **3.6 mio** ops / m

Throughput

- redis-benchmark
  - 60 K ops / s = **3.6 mio** ops / m
- monitoring (rpm, scout)
  - ca. 10 ops per request

Throughput

- redis-benchmark
  - 60 K ops / s = **3.6 mio** ops / m
- monitoring (rpm, scout)
  - ca. 10 ops per request
- 200 K rpm = **2.0 mio** ops / m ✔

Capacity 1

- 100 KB / user (on disk)
- 10.000 concurrent users (peak)

Capacity 1

- 100 KB / user (on disk)
- 10.000 concurrent users (peak)

- 1 GB memory
  - (plus Redis overhead) ✔

Capacity 2

- Redis keeps all keys in memory
- 10 mio. total users
- 20 GB / 100 mio. integer keys

Capacity 2

- Redis keeps all keys in memory
- 10 mio. total users
- 20 GB / 100 mio. integer keys
- 2 GB memory for keys ✔

Data model

- one Redis hash per user
  - key: facebook id

Data model

- one Redis hash per user
  - key: facebook id
- store data as serialized JSON
  - booleans, strings, numbers, timestamps ...

Advantages

- efficient to swap user data in / out

Advantages

- efficient to swap user data in / out
- turns Redis into “document db”
  - atomic ops on parts

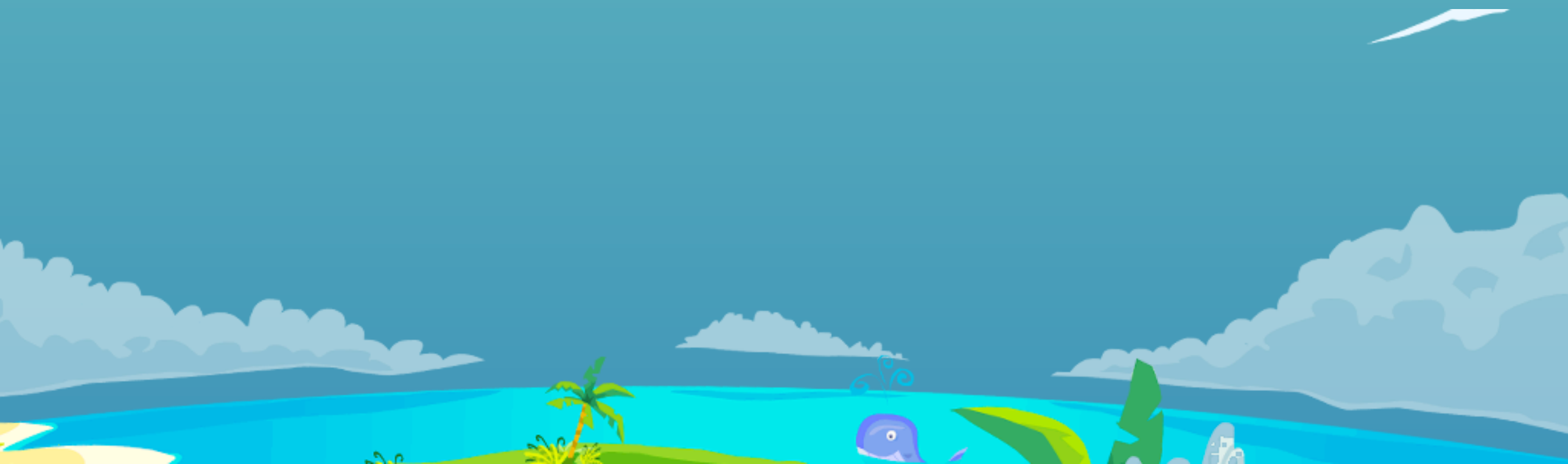
Advantages

- efficient to swap user data in / out
- turns Redis into “document db”
  - atomic ops on parts
- easy to dump / restore user data

Lessons

"You are not facebook."

—me



Advice

- use the right tool for the job

Advice

- use the right tool for the job
- avoid scaling out / sharding, if possible
- do the numbers!

Advice

- use the right tool for the job
- avoid scaling out / sharding, if possible
  - do the numbers!

- **keep it simple**

Q & A

Links

- cassandra.apache.org
- redis.io
- tim.lossen.de