CC5212-1

Procesamiento Masivo de Datos

Otoño 2021

Lecture 9

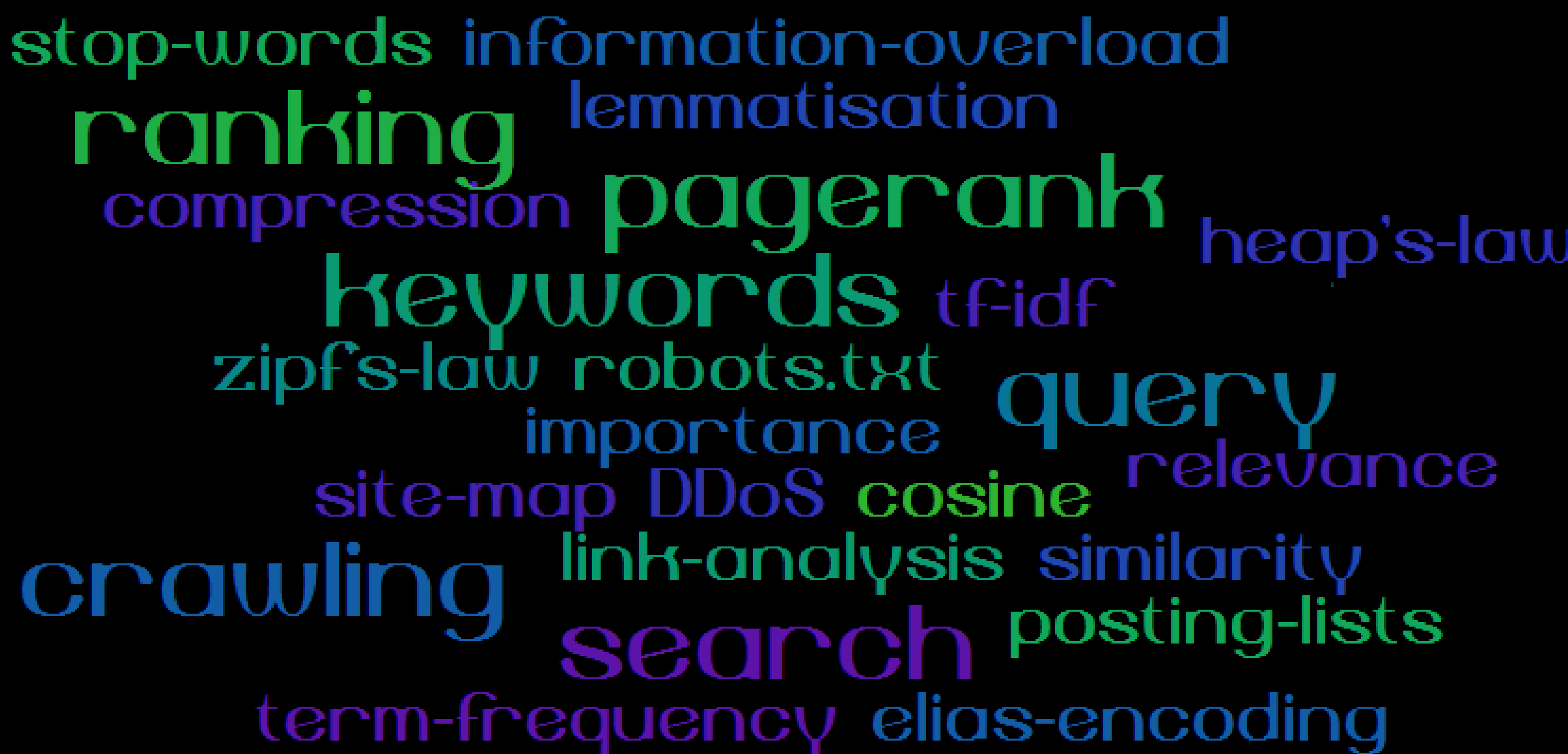
NoSQL: Overview

Aidan Hogan

aidhog@gmail.com

Hadoop/MapReduce/Pig/Spark:
Processing Un/Structured Information

Information Retrieval:
Storing Unstructured Information



Storing Structured Information??

?

Big Data:

Storing Structured Information

Relational Databases


Spongecell
Hula
kiko
Trumba
eskobo
mayomi
Pageflakes
vimeo
skobee
shadows
gravee
YouTube
Zimbra
LookSmart FURL Your Personal Web File
smugmug
newsgator
Blogniscient
TinyFinger
shutterfly
PodDater
Feedster
favoor
Planzo.com
ZAZZLE
Tailrank
TagWorld
nuvvo
dogear
yakalike
grouper
ODDPOST
qoop
iNods
Lulu
rbloc.com
blish
flagr
FireAnt
simplyhired
Veoh
theadcloud
gather
Agatra
browsr
oyogi
cafepress
Renkoo
standpoint
pixagogo
meebo
lastfm
YEDDA
Jotspot
Frappr!
jeteye
dabble db
Writeboard
SHOUTWIRE
iKarma
AirSet
tech.memeorandum
CalendarHub
Suprglu
pando
zigtag
Findory
backfence
clipmarks
wayfaring
gOFFICE
AllPeers
Orb
Rallypoint
Zoozio
blogbeat
Ziggs
zooto
vSocial
Boltfolio
wink
riya
Wordcast
Opinity
reddit
measure map
gumshoo
bluepulse
imeem
STREAMLOAD
Ta-da Lists
FeedSky
jellyBam.INC
nativetext
CONGOO
PODZINGER
RSS MAD
FeedTier
phanfare
WIKIPEDIA
Fruitcast
PubSub
dPolls
flickr
Ning
Ookles
Strongspace
zoominfo
CASTPOST
yubnub
Bloop
ProjectSpaces
FeedBurner
Bloglines
purevolume.com
FOTOLOG
ourmedia
gabbr.com
Gcast
blinkx
openomy
riffs
Yub.com
ajchat
Spot Runner
myspace
NewsAlloy
Allmydata.com
jambo
ROLLYO
ClipShack
chatsum
PANDORA
looklater
PROTOPAGE
squishr
WebJay
PLAZES
Noodly
wondir
diigo
box
Jots
Xdrive
vizu
digg
del.icio.us
omnidrive
AlmondRocks
Tagyu
30 gigs.com
writely
Simpy
Gtalkr
TRUVEO
egoSurf
mozy
Quimble
pegasus
remember the milk
SQUIDOO
picturecloud
newsvine
Clipfire
Basecamp
Netvibes
Goowy
MyVideoKaraoke
Lexxe alpha
BOOKSTER
PXN8
measure map
facebook
43 Things
Technorati
Platial
Backpack
Gmail
yelp
Smarkets
inform
magnolia

Relational Databases: One Size Fits All?

"One Size Fits All": An Idea Whose Time Has Come and Gone

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, M.I.T., and StreamBase Systems, Inc.
stonebraker@csail.mit.edu

Uğur Çetintemel
Department of Computer Science Brown University, and StreamBase Systems, Inc.
ugur@cs.brown.edu

Abstract

The last 25 years of commercial DBMS development can be summed up in a single phrase: "One size fits all". This phrase refers to the fact that the traditional DBMS architecture (originally designed and optimized for business data processing) has been used to support many data-centric applications with widely varying characteristics and requirements.

In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines, some of which may be unified by a common front-end parser. We use examples from the stream-processing market and the data-warehouse market to bolster our claims. We also briefly discuss other markets for which the traditional architecture is a poor fit and argue for a critical rethinking of the current factoring of systems services into products.

of multiple code lines causes various practical problems, including:

- *a cost problem*, because maintenance costs increase at least linearly with the number of code lines;
- *a compatibility problem*, because all applications have to run against every code line;
- *a sales problem*, because salespeople get confused about which product to try to sell to a customer; and
- *a marketing problem*, because multiple code lines need to be positioned correctly in the marketplace.

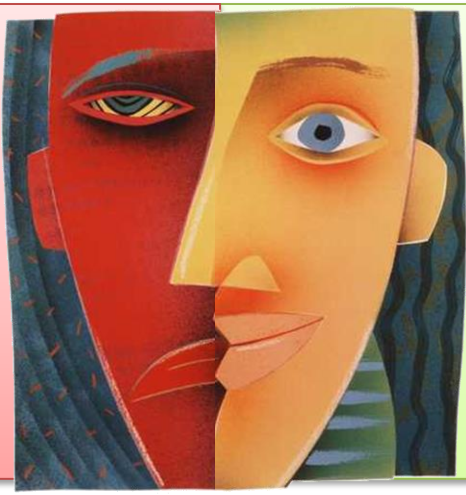
To avoid these problems, all the major DBMS vendors have followed the adage "put all wood behind one arrowhead". In this paper we argue that this strategy has failed already, and will fail more dramatically off into the future.

The rest of the paper is structured as follows. In Section 2, we briefly indicate why the single code-line strategy has failed already by citing some of the key characteristics of the data warehouse market. In Section

SQL

Difficult to optimise

Difficult to distribute



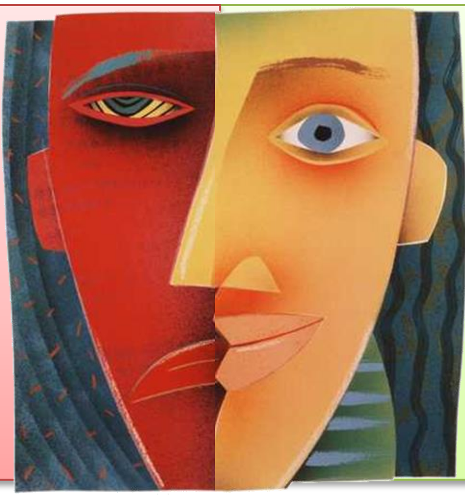
Declarative language

Expressive

ACID

Costly to implement

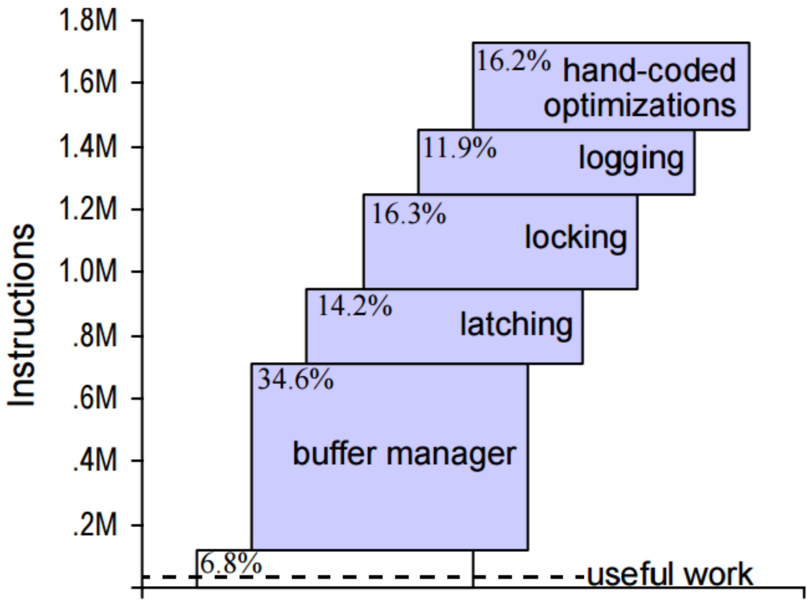
Difficult to distribute



Guarantees correct behaviour

Support transactions

Transactional overhead: the cost of ACID



- 640 transactions per second for system with full transactional support (ACID)
- 12,700 transactions per second for system without logs, transactions or lock scheduling

OLTP Through the Looking Glass, and What We Found There

Stavros Harizopoulos, HP Labs, Palo Alto, CA, stavros@hp.com
Daniel J. Abadi, Yale University, New Haven, CT, dna@cs.yale.edu
Samuel Madden, Michael Stonebraker, Massachusetts Institute of Technology, Cambridge, MA, {madden, stonebraker}@csail.mit.edu

ABSTRACT
Online Transaction Processing (OLTP) databases include a suite of features — disk-resident B-trees and heap files, locking-based concurrency control, support for multi-threading — that were optimized for computer technology of the late 1970's. Advances in modern processors, memories, and networks mean that today's computers are vastly different from those of 30 years ago, such that many OLTP databases will now fit in main memory, and most OLTP transactions can be processed in milliseconds or less. Yet database architecture has changed little.

1. INTRODUCTION
Modern general purpose online transaction processing (OLTP) database systems include a standard suite of features: a collection of on-disk data structures for table storage, including heap files and B-trees, support for multiple concurrent queries via locking-based concurrency control, log-based recovery, and an efficient buffer manager. These features were developed to support transaction processing in the 1970's and 1980's, when an OLTP database was many times larger than the main memory, and when the computers that ran these databases cost hundreds of thousands to

Alternatives to Relational Databases for Big Data?

NoSQL

Anybody know anything about NoSQL?

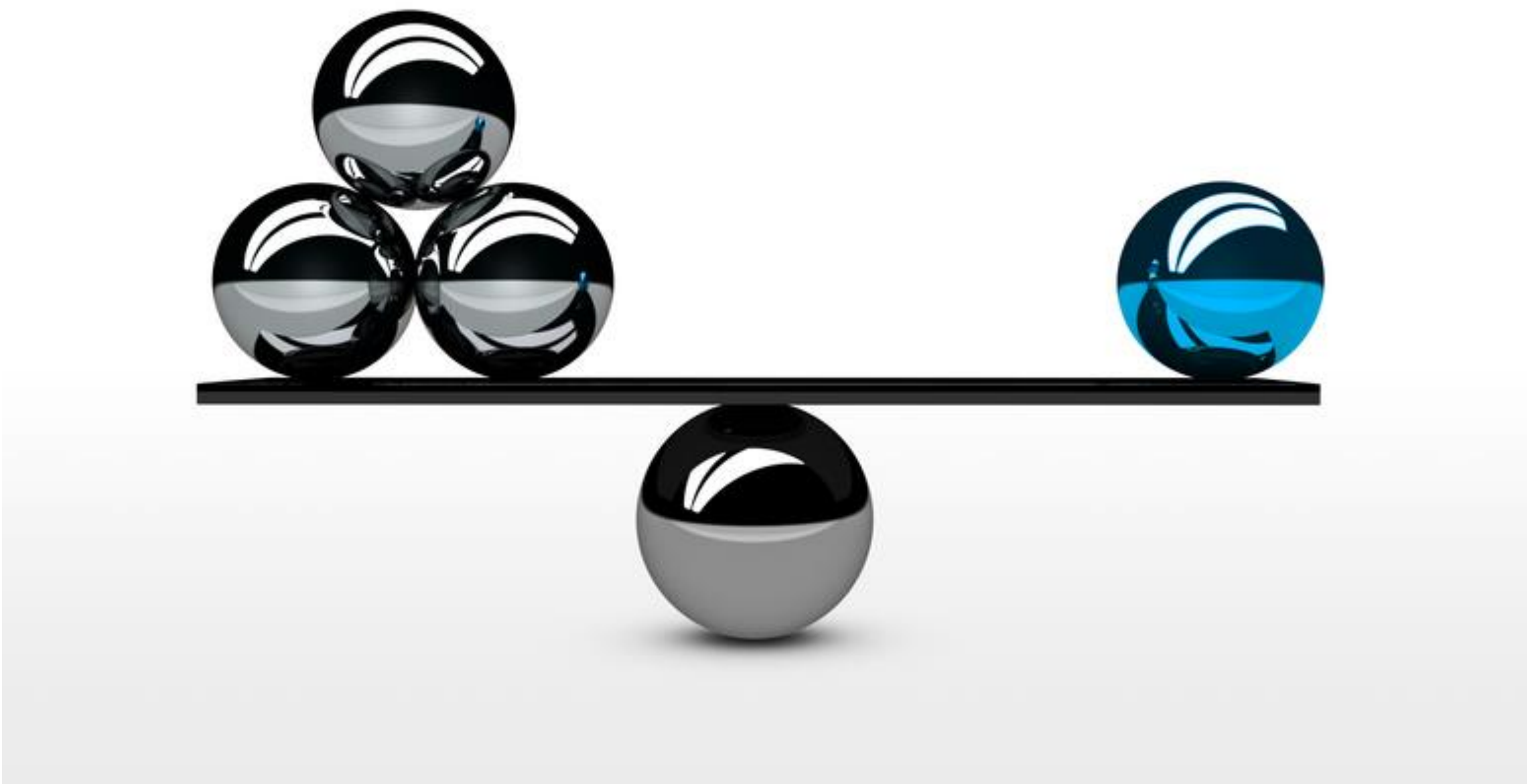
Not Only SQL

371 systems in ranking, May 2021

Rank			DBMS	Database Model	Score		
May 2021	Apr 2021	May 2020			May 2021	Apr 2021	May 2020
1.	1.	1.	Oracle	Relational, Multi-model	1269.94	-4.98	-75.50
2.	2.	2.	MySQL	Relational, Multi-model	1236.38	+15.69	-46.26
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	992.66	-15.30	-85.64
4.	4.	4.	PostgreSQL	Relational, Multi-model	559.25	+5.73	+44.45
5.	5.	5.	MongoDB	Document, Multi-model	481.01	+11.04	+42.02
6.	6.	6.	IBM Db2	Relational, Multi-model	166.66	+8.88	+4.02
7.	7.	↑ 8.	Redis	Key-value, Multi-model	162.17	+6.28	+18.69
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	155.35	+3.18	+6.23
9.	9.	9.	SQLite	Relational	126.69	+1.64	+3.66
10.	10.	10.	Microsoft Access	Relational	115.40	-1.33	-4.50
11.	11.	11.	Cassandra	Wide column	110.93	-3.92	-8.22
12.	12.	12.	MariaDB	Relational, Multi-model	96.69	+0.32	+6.61
13.	13.	13.	Splunk	Search engine	92.11	+3.62	+4.36
14.	14.	14.	Hive	Relational	76.19	-2.31	-5.35
15.	15.	↑ 23.	Microsoft Azure SQL Database	Relational, Multi-model	70.46	-1.39	+27.70
16.	16.	16.	Amazon DynamoDB	Multi-model	70.07	-0.66	+5.35
17.	17.	↓ 15.	Teradata	Relational, Multi-model	69.98	-0.57	-3.91
18.	18.	↑ 20.	SAP HANA	Relational, Multi-model	52.75	-0.69	+2.22
19.	↑ 20.	↑ 21.	Neo4j	Graph	52.23	+1.19	+2.47
20.	↑ 21.	↓ 18.	Solr	Search engine, Multi-model	51.19	+0.59	-1.39
21.	↓ 19.	↓ 17.	SAP Adaptive Server	Relational, Multi-model	49.96	-1.70	-4.03
22.	22.	↓ 19.	FileMaker	Relational	46.73	+0.32	-4.23
23.	23.	↓ 22.	HBase	Wide column	43.24	-0.92	-6.48
24.	24.	↑ 26.	Google BigQuery	Relational	37.63	+2.05	+10.04
25.	25.	↓ 24.	Microsoft Azure Cosmos DB	Multi-model	34.71	+1.19	+4.03

http://db-engines.com/en/ranking

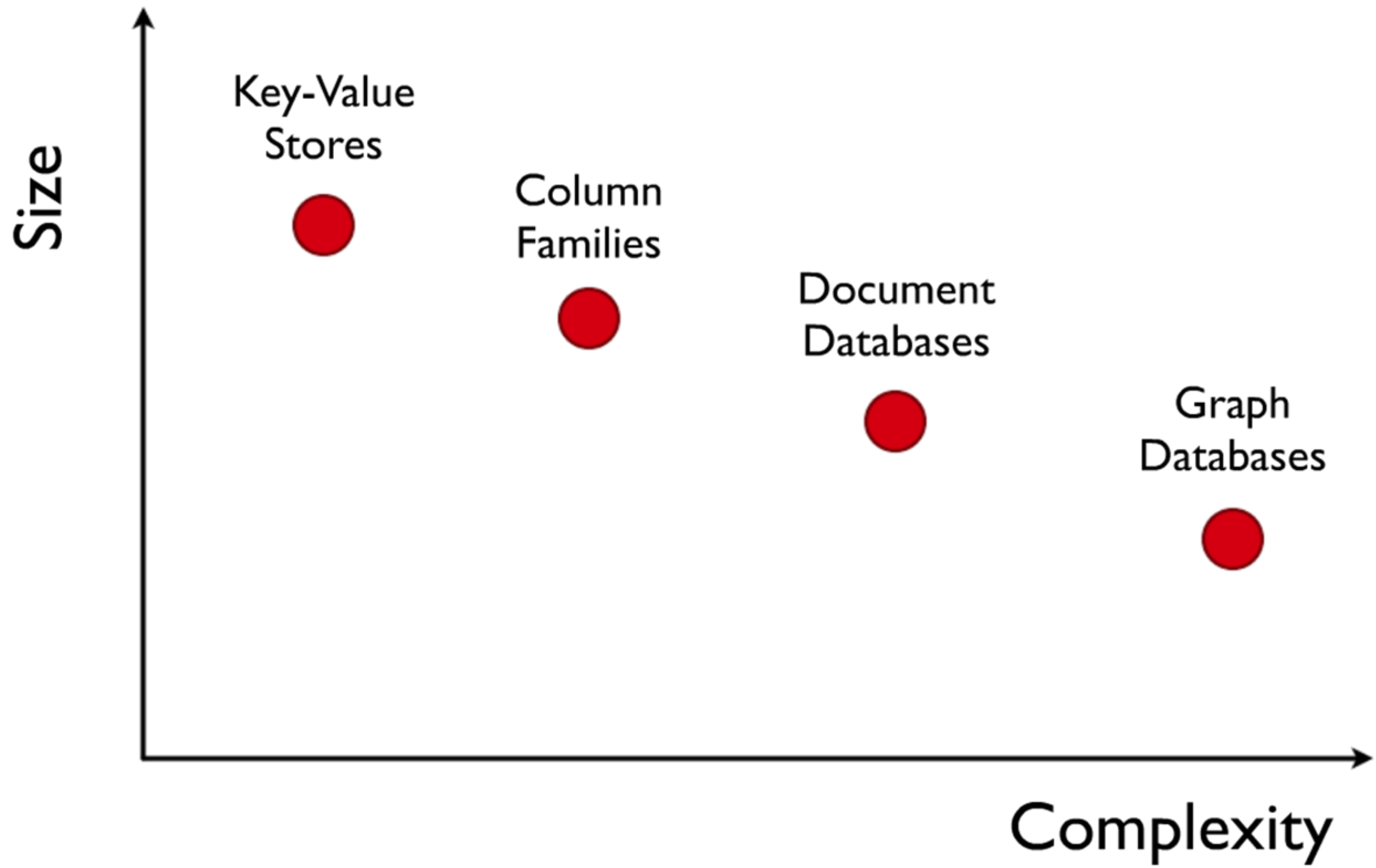
NoSQL: features vs. scale/performance



NoSQL: common characteristics

- Often distributed
- Often simpler languages than SQL
- Different flavours (for different scenarios)

NoSQL: four main flavours



Limitations of distributed computing: CAP Theorem

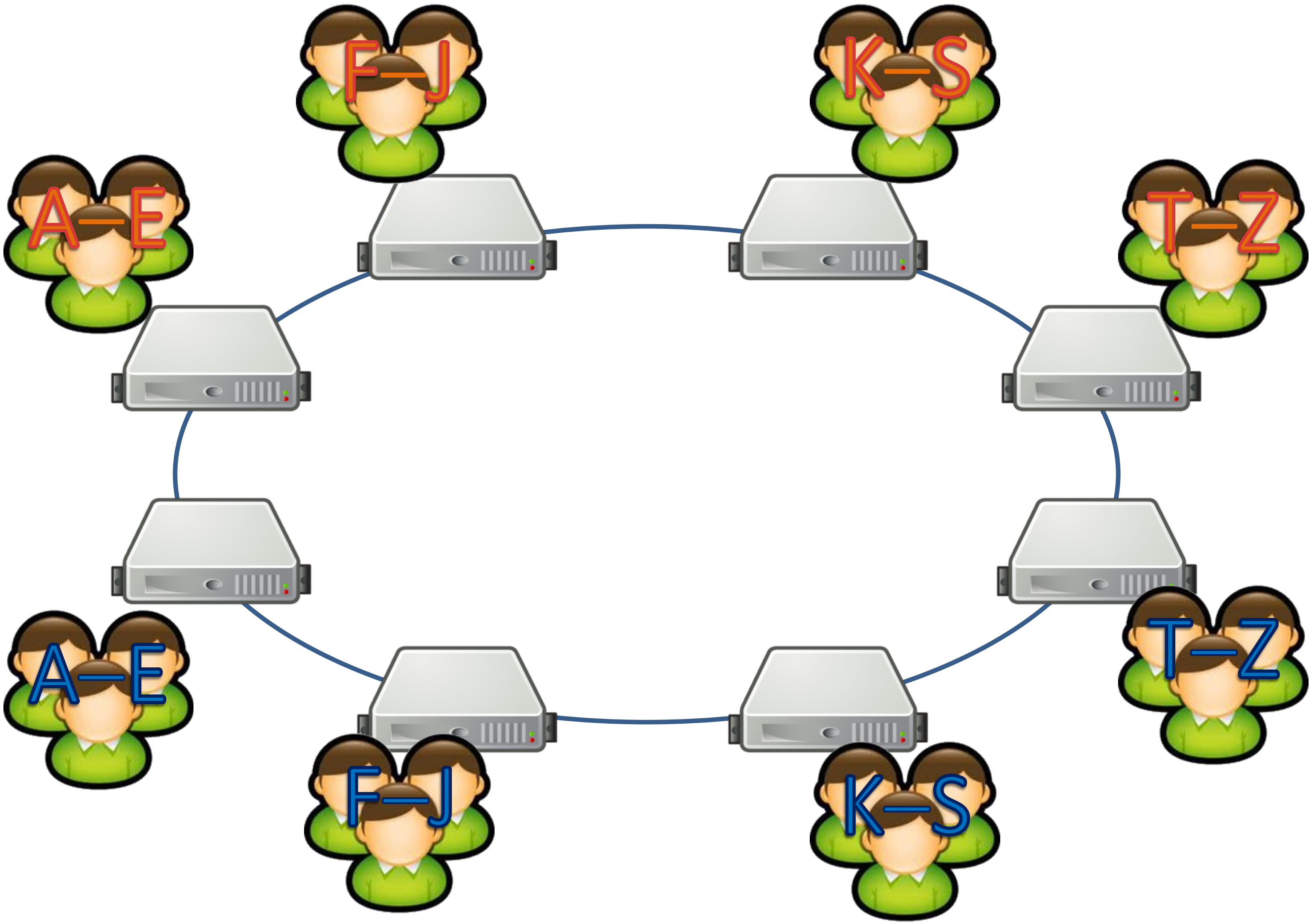
What is CAP?

Three *guarantees* a <u>distributed</u> sys. could make

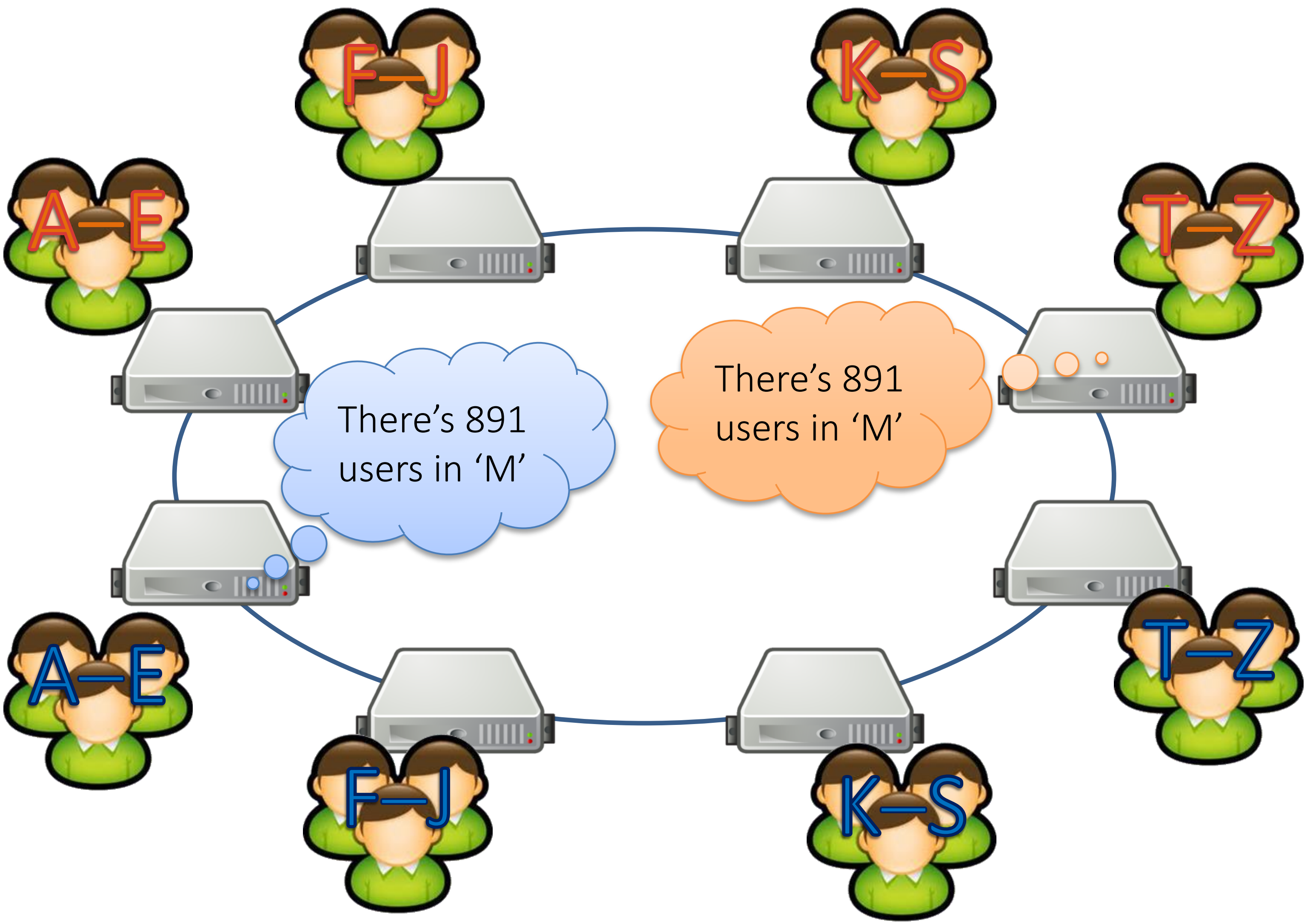
1. **C**onsistency:
   – All nodes have a consistent view of the system
2. **A**vailability:
   – Every read/write is acted upon
3. **P**artition-tolerance:
   – The system works even if messages are lost

CA in CAP not the same as CA in ACID!!

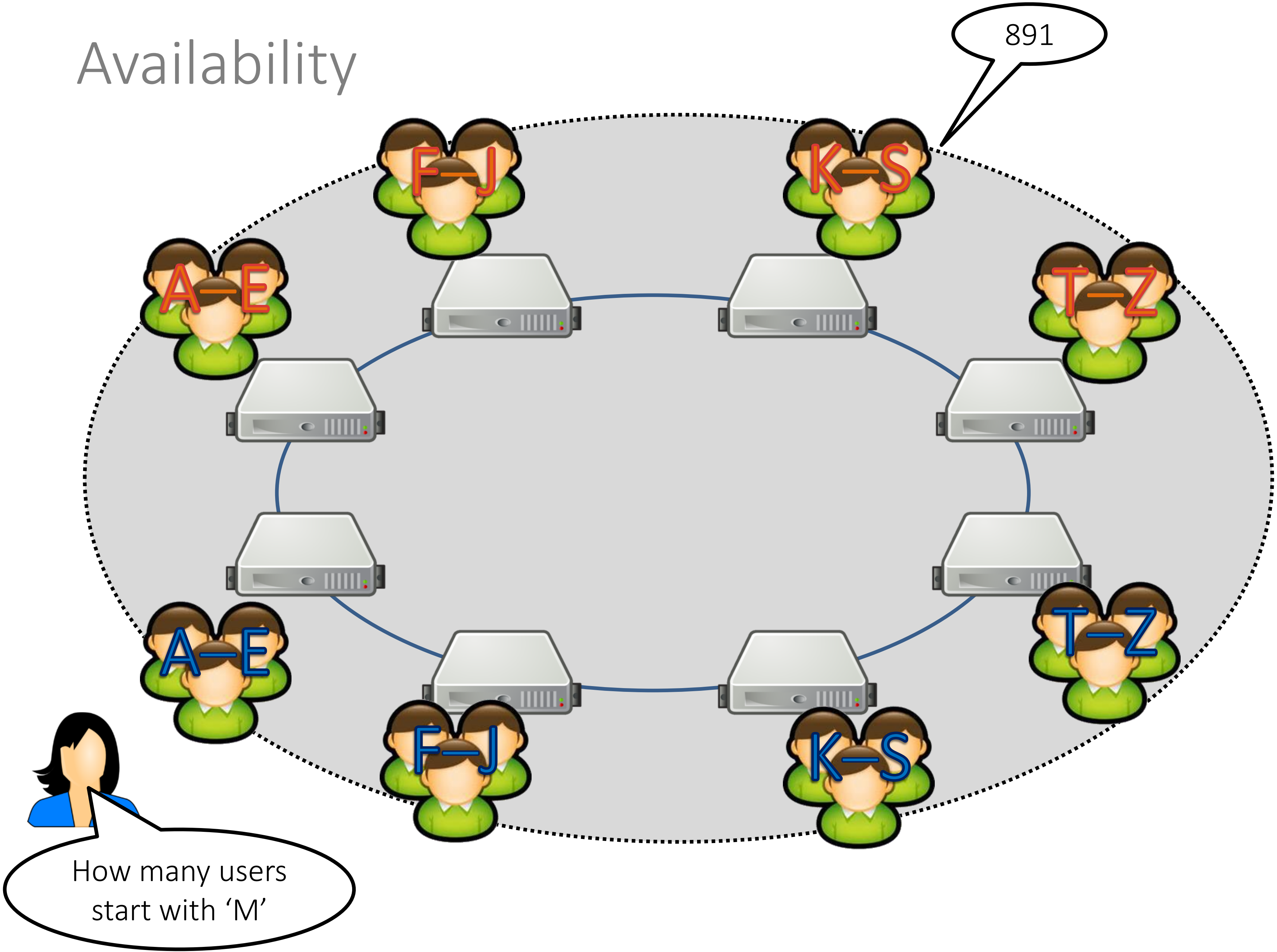
A Distributed System (with Replication)



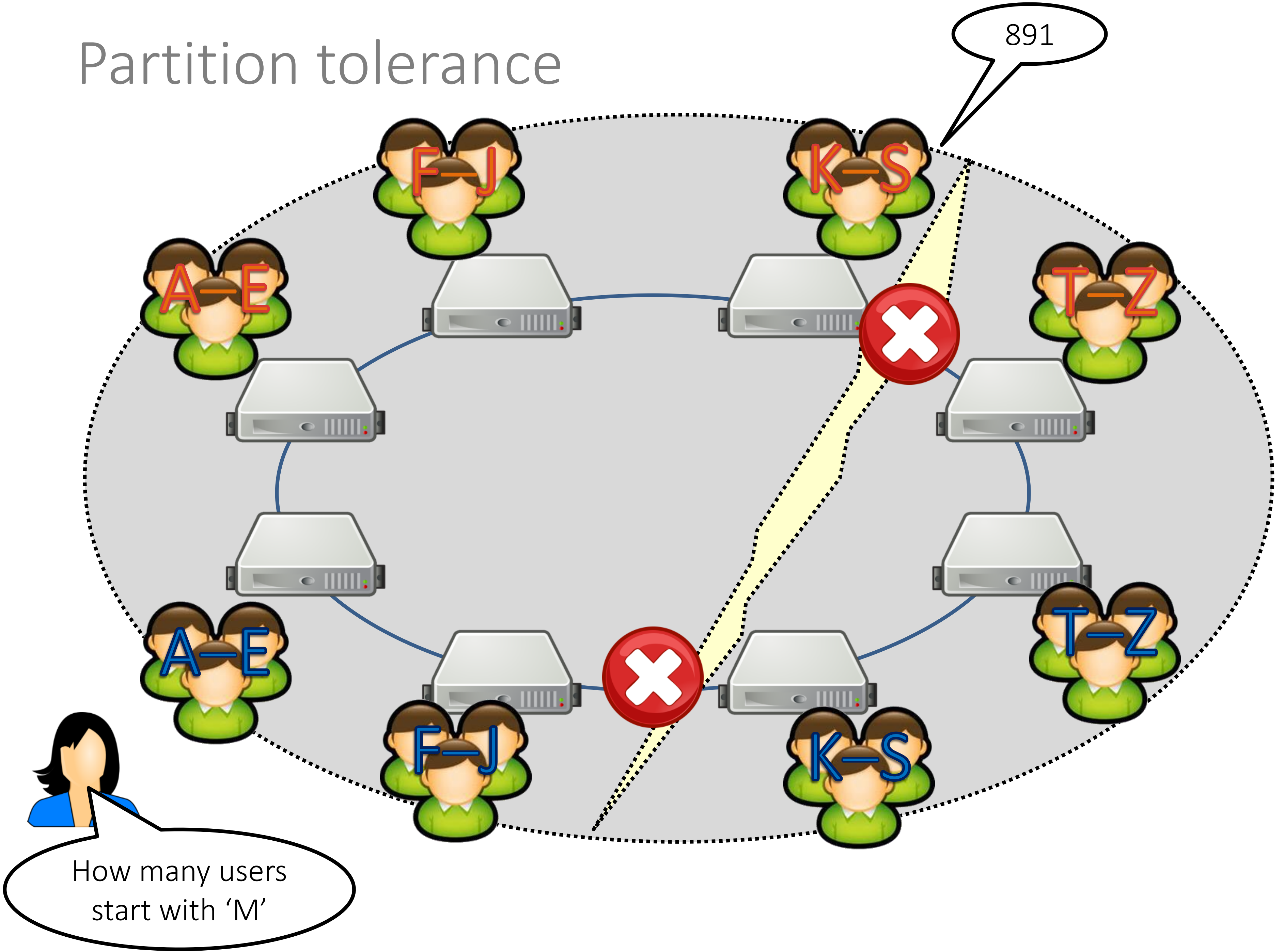
Consistency



Availability



Partition tolerance



The CAP Question

Can a distributed system guarantee

consistency (all nodes have the same up-to-date view),

availability (every read/write is acted upon) and

partition-tolerance (the system works if messages are lost)

at the same time?

What do you think?

The CAP Answer

The CAP Theorem

A distributed system cannot guarantee

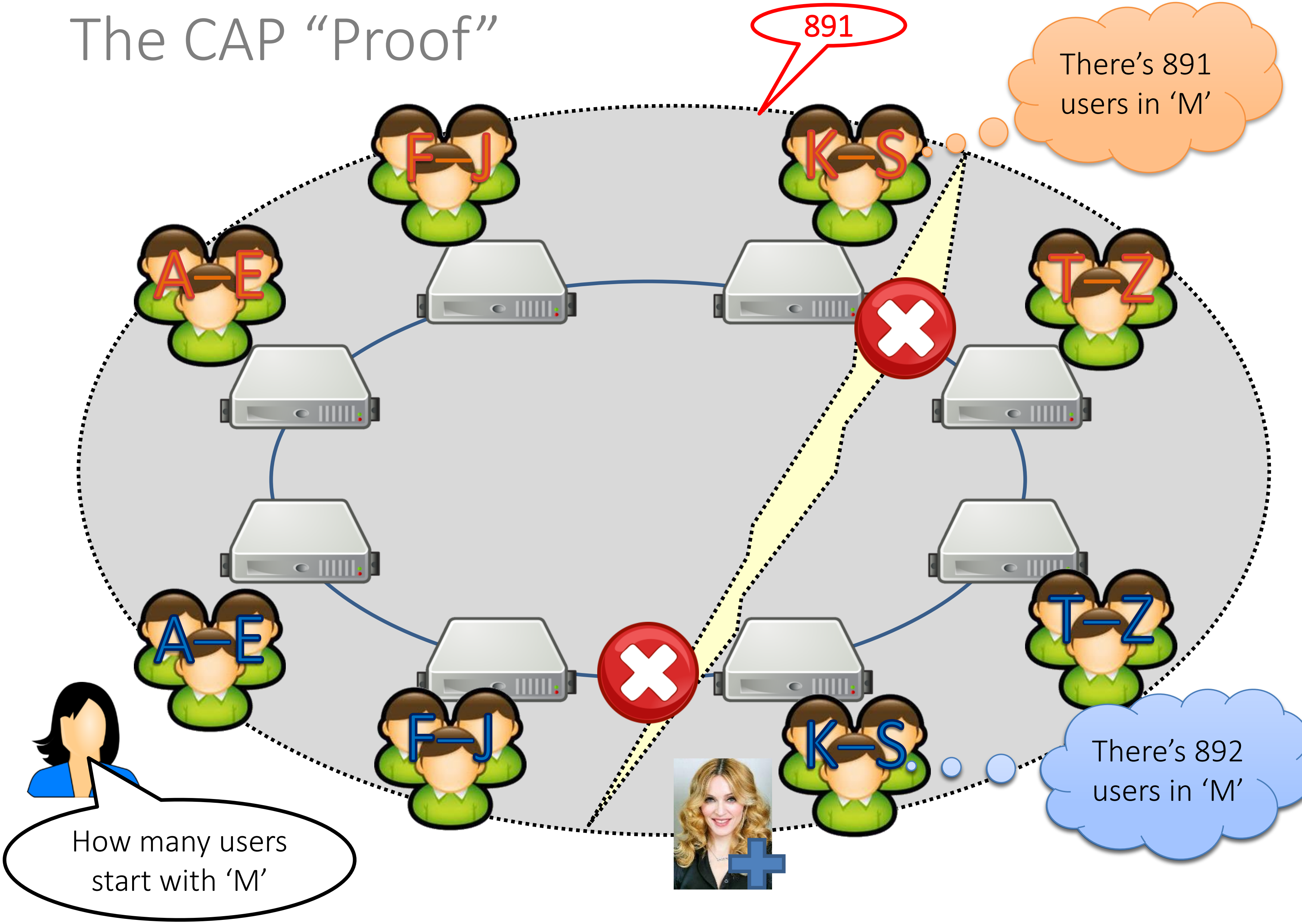
consistency (all nodes have the same up-to-date view),

availability (every read/write is acted upon) and

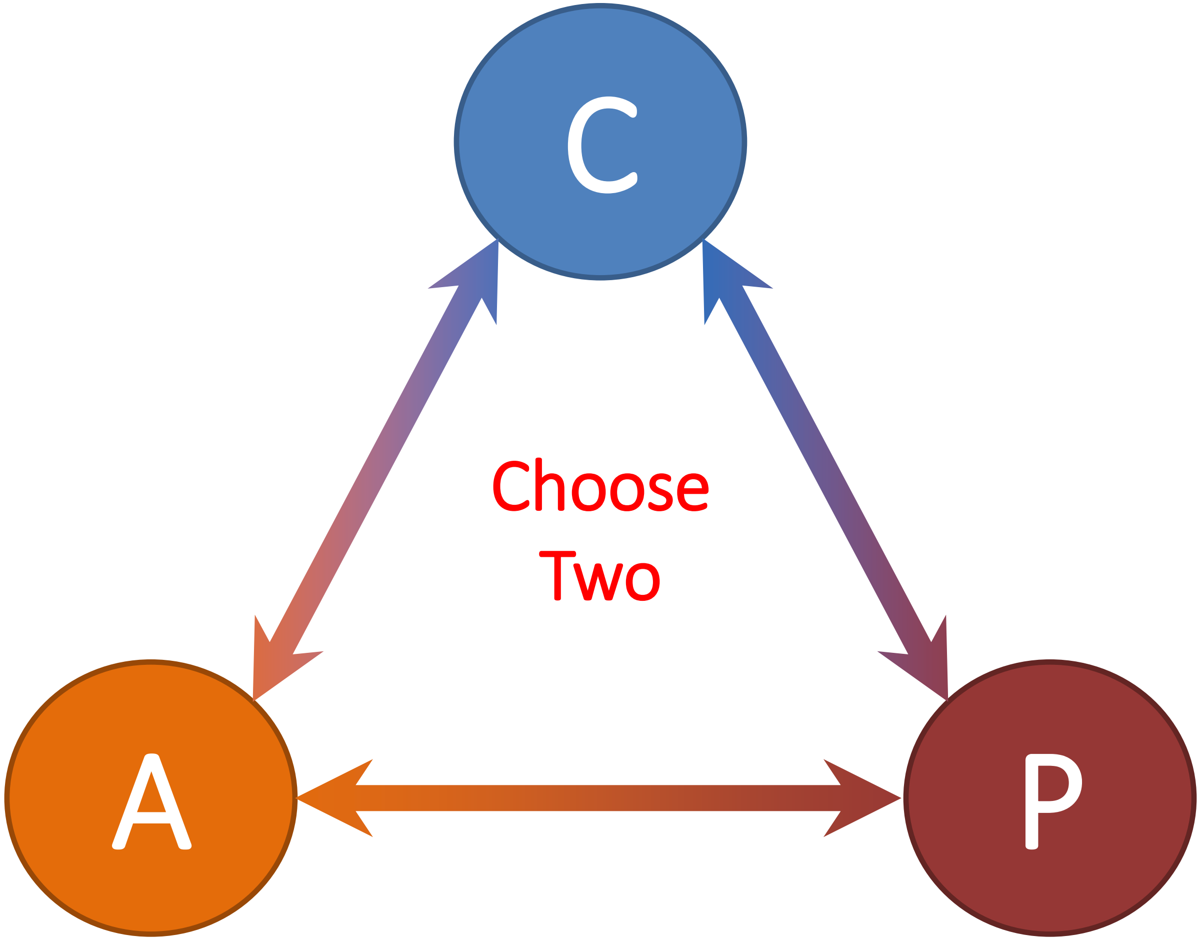
partition-tolerance (the system works if messages are lost)

at the same time!

The CAP “Proof”



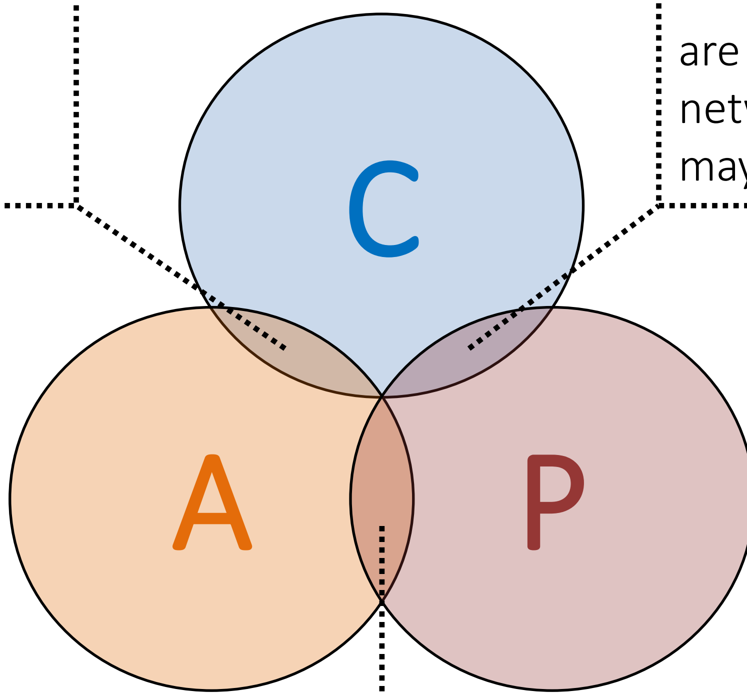
The CAP Triangle



CAP Systems

CA: Guarantees to give a correct response but only while network works fine (*Centralised / Traditional*)

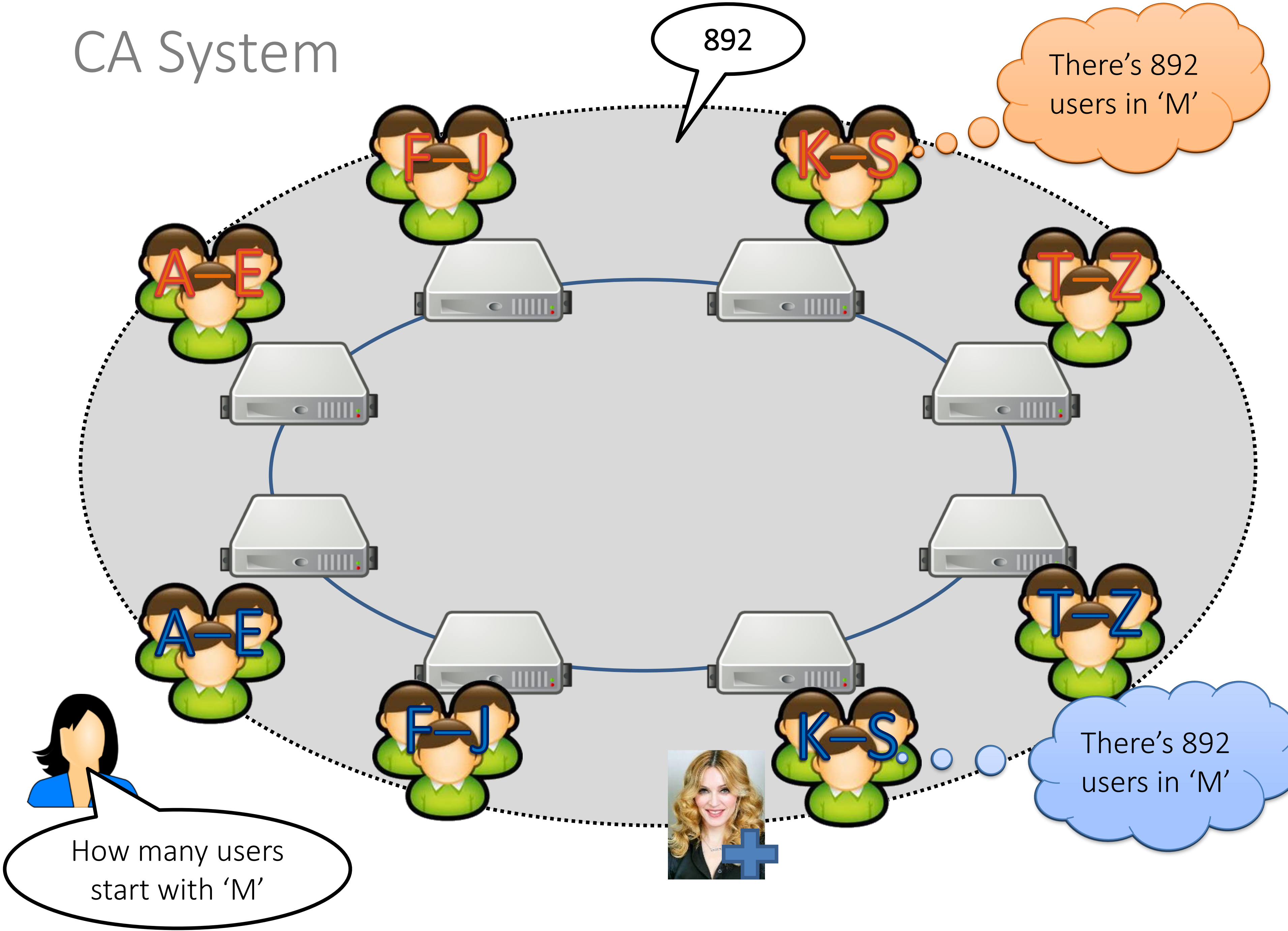
CP: Guarantees responses are correct even if there are network failures, but response may fail (*Weak availability*)



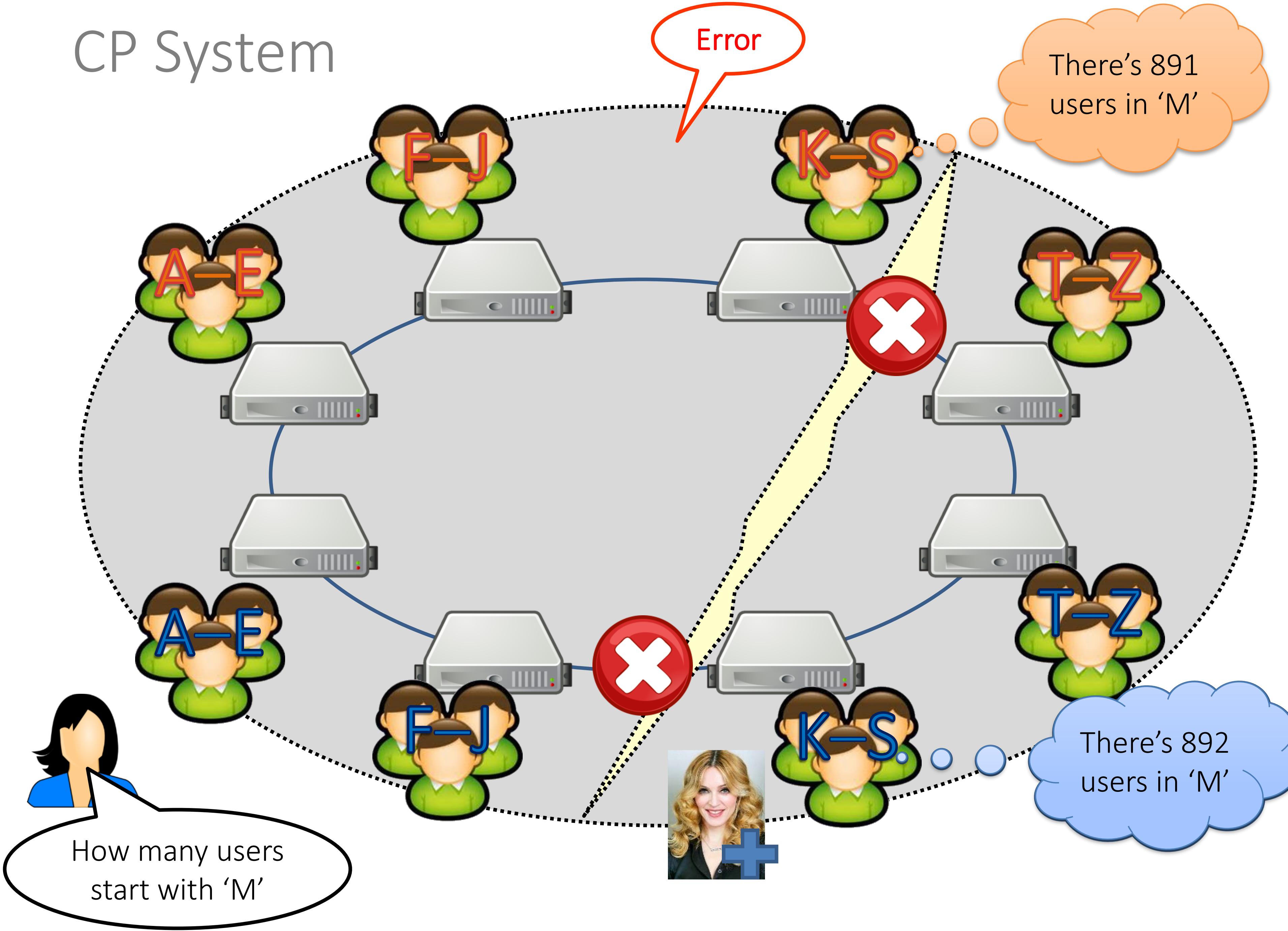
(No intersection)

AP: Always provides a “best-effort” response even in presence of network failures (*Eventual consistency*)

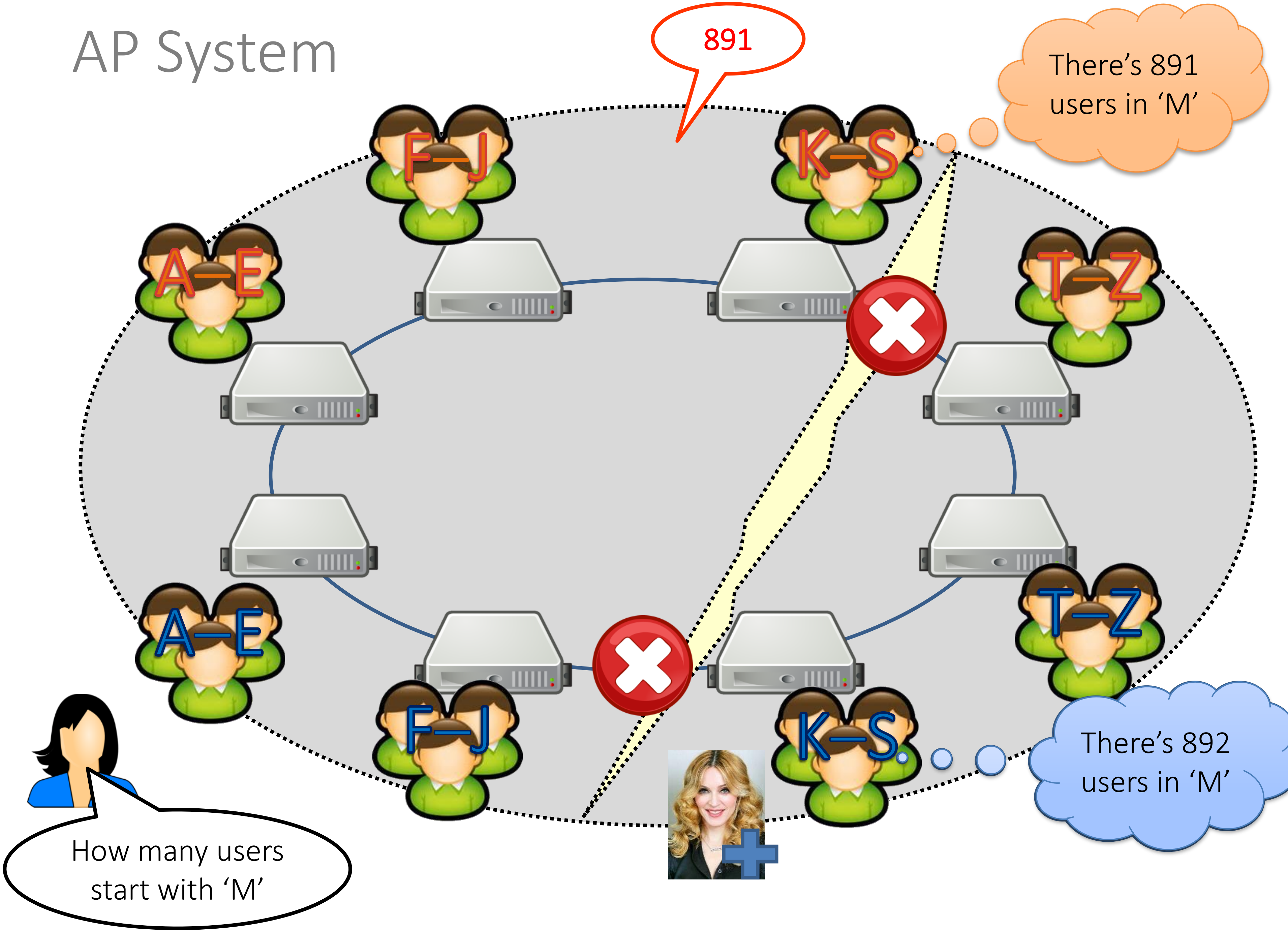
CA System


892
There's 892
users in 'M'
A—E
F—J
K—S
T—Z
A—E
F—J
K—S
T—Z
There's 892
users in 'M'
How many users
start with 'M'

CP System


Error
There's 891 users in 'M'
K–S
F–J
A–E
T–Z
A–E
F–J
K–S
T–Z
There's 892 users in 'M'
How many users start with 'M'

AP System


891
There's 891 users in 'M'
K–S
F–J
A–E
T–Z
A–E
F–J
K–S
T–Z
There's 892 users in 'M'
How many users start with 'M'

BASE (AP)

- Basically Available
  - Almost always “up”
- Soft State
  - Replicated, cached data
- Eventual Consistency
  - Stale data tolerated, for a while

In what way does Twitter act as a BASE (AP) system?

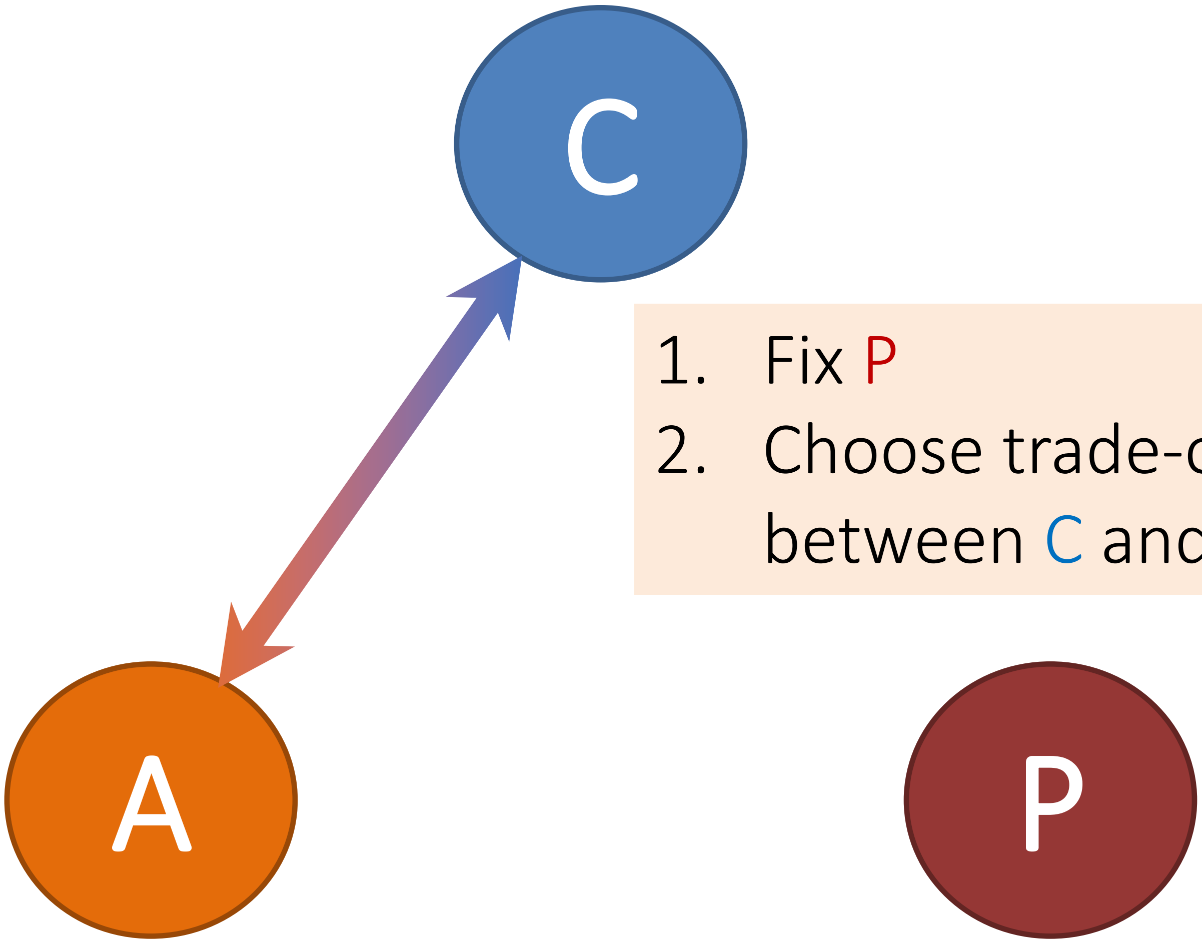
High-fanout creates a "partition"

Rank	Change (monthly)	Account name	Owner	Followers (millions)	Occupation	Country
1	—	@BarackObama	Barack Obama	129.9	44th President of the United States	United States
2	—	@justinbieber	Justin Bieber	114.1	Musician	Canada
3	—	@katyperry[a]	Katy Perry	109.1	Musician	United States
4	—	@rihanna	Rihanna	102.5	Musician and businesswoman	Barbados
5	—	@Cristiano	Cristiano Ronaldo	92.1	Footballer	Portugal
6	—	@taylorswift13	Taylor Swift	88.6	Musician	United States
7	—	@ladygaga	Lady Gaga	83.9	Musician and actress	United States
8	—	@ArianaGrande	Ariana Grande	83.2	Musician and actress	United States
9	—	@TheEllenShow	Ellen DeGeneres	78.5	Comedian and television hostess	United States
10	—	@YouTube	YouTube	73	Online video platform	United States

Users may see retweets of celebrity tweets before the original tweet.

Later when the original tweet arrives the timeline will be reordered and made consistent.

CAP in practical distributed systems



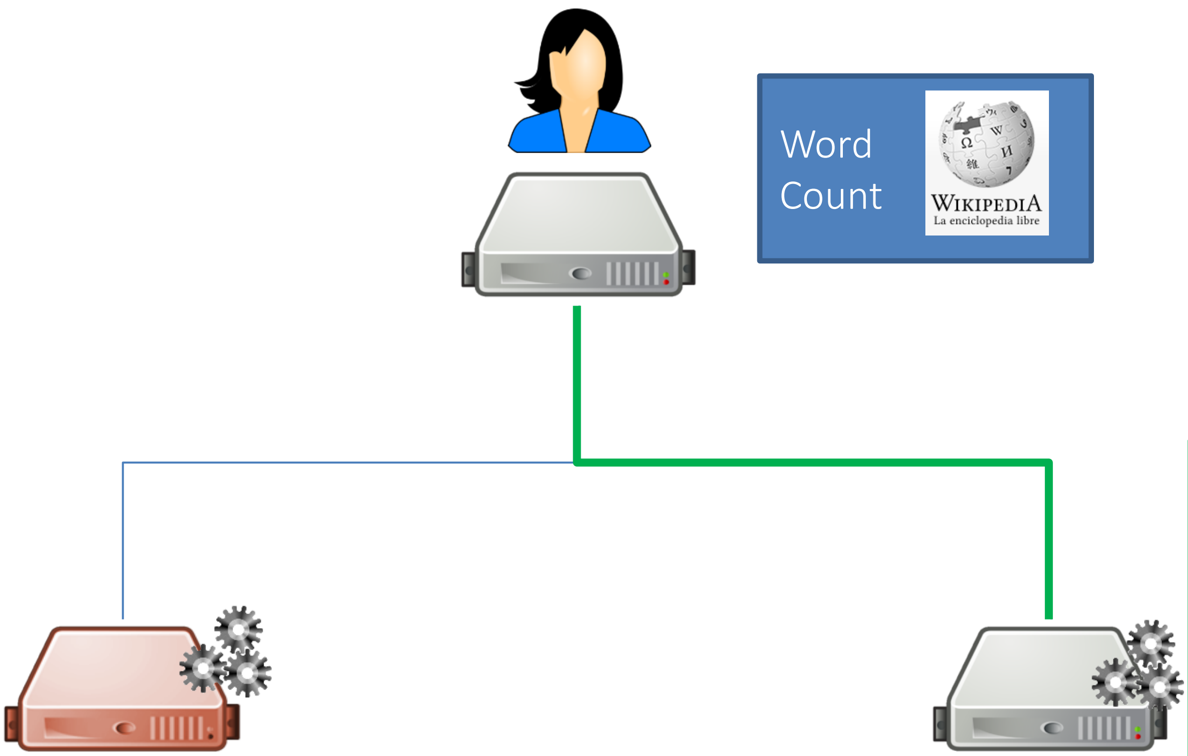
1. Fix P
2. Choose trade-off point between C and A

Partition Tolerance

Faults

Fail–Stop Fault

- A machine fails to respond or times-out
  - often hardware or load
  - need at least $f + 1$ replicated machines
    - f = number of fail-stop failures

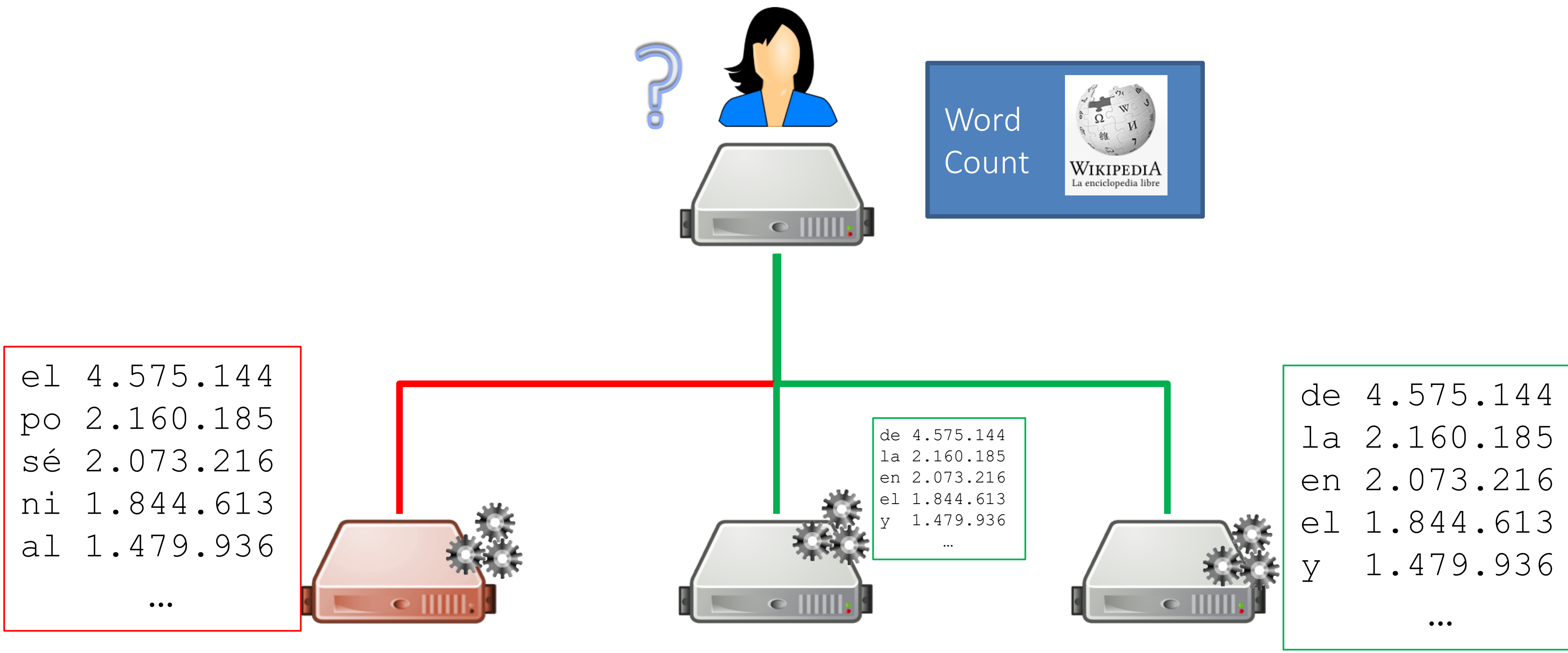


de	4.575.144
la	2.160.185
en	2.073.216
el	1.844.613
y	1.479.936
…	

Byzantine Fault

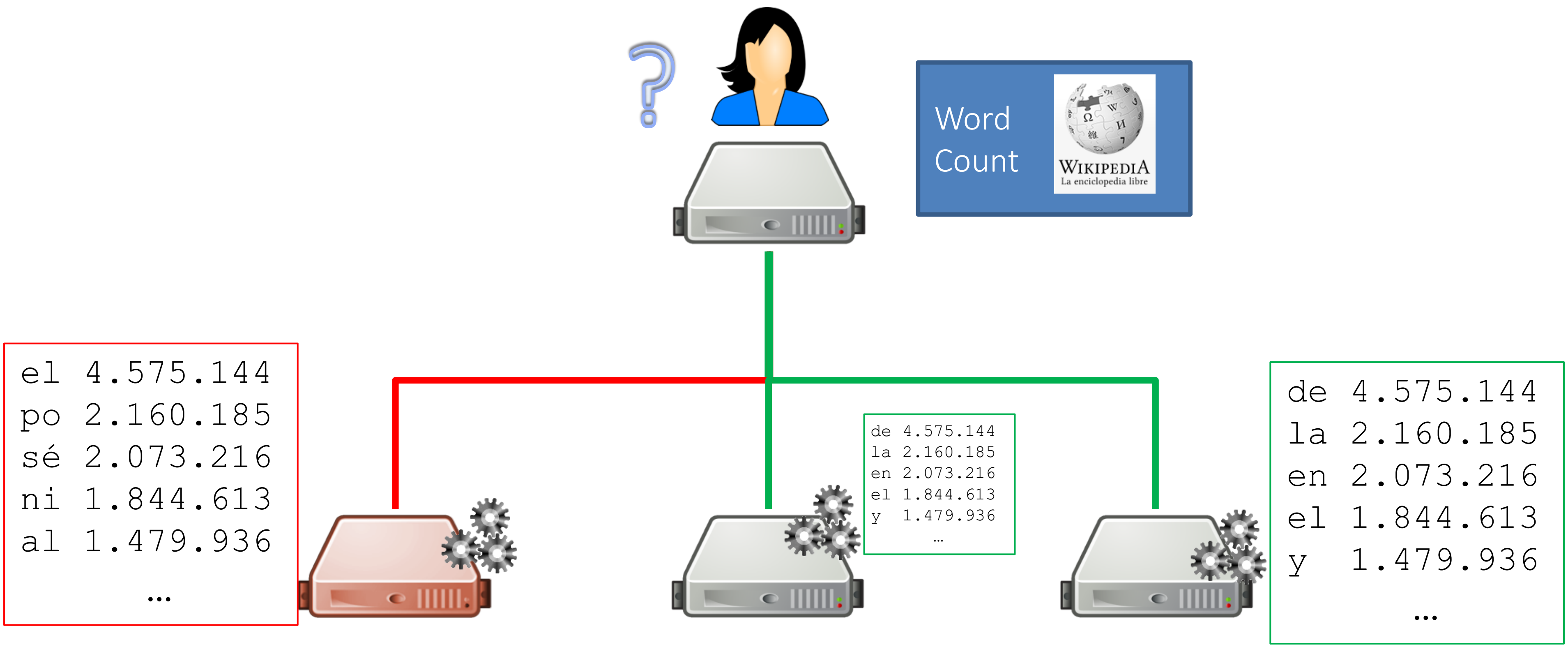
- A machine responds incorrectly/maliciously

How many working machines do we need in the general case to be robust against Byzantine faults?



Byzantine Fault

- A machine responds incorrectly/maliciously
  - Need *at least* **2***f* +1 replicated machines
    - *f* = number of (possibly Byzantine) failures



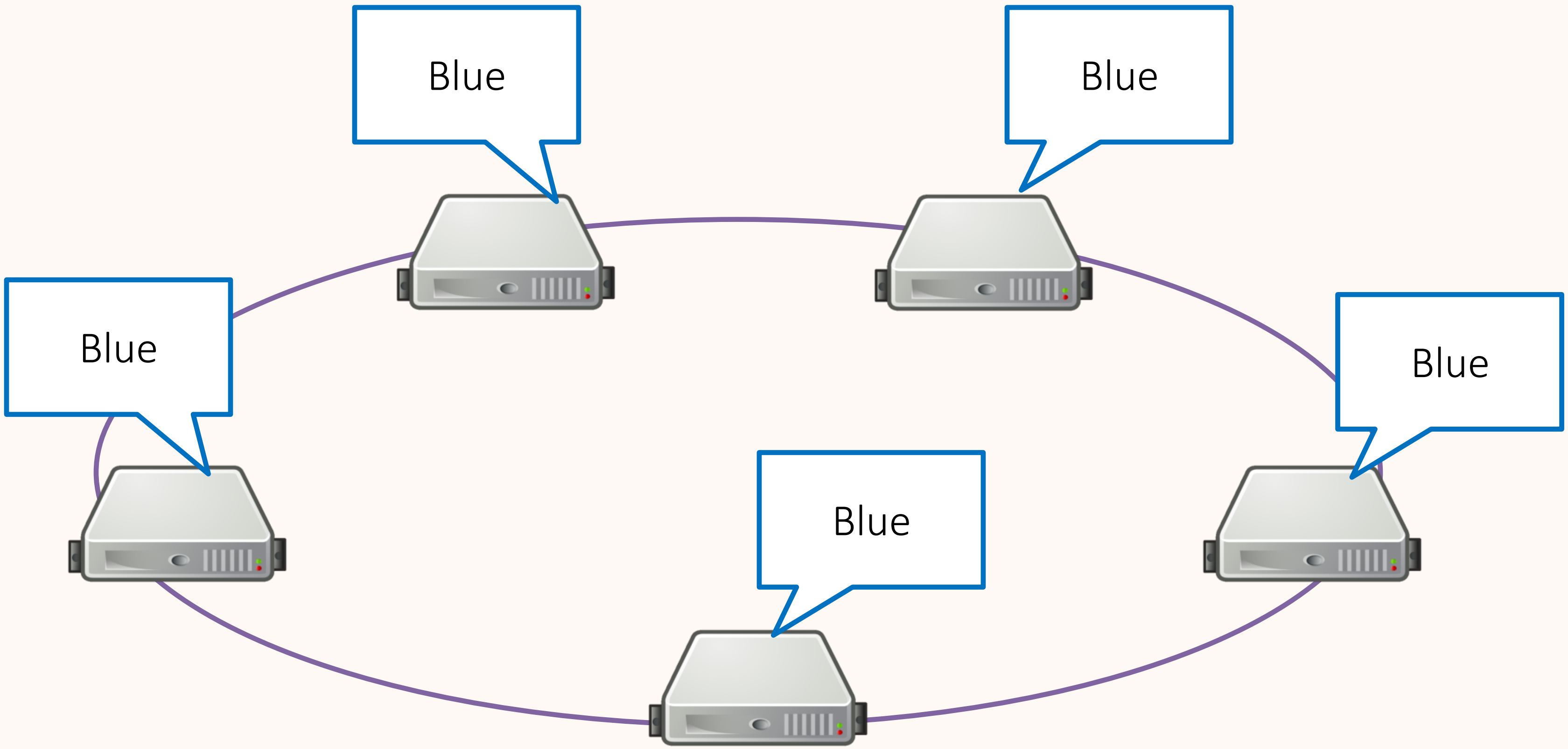
Distributed Consensus

Distributed Consensus

Colour of the dress?

Distributed Consensus

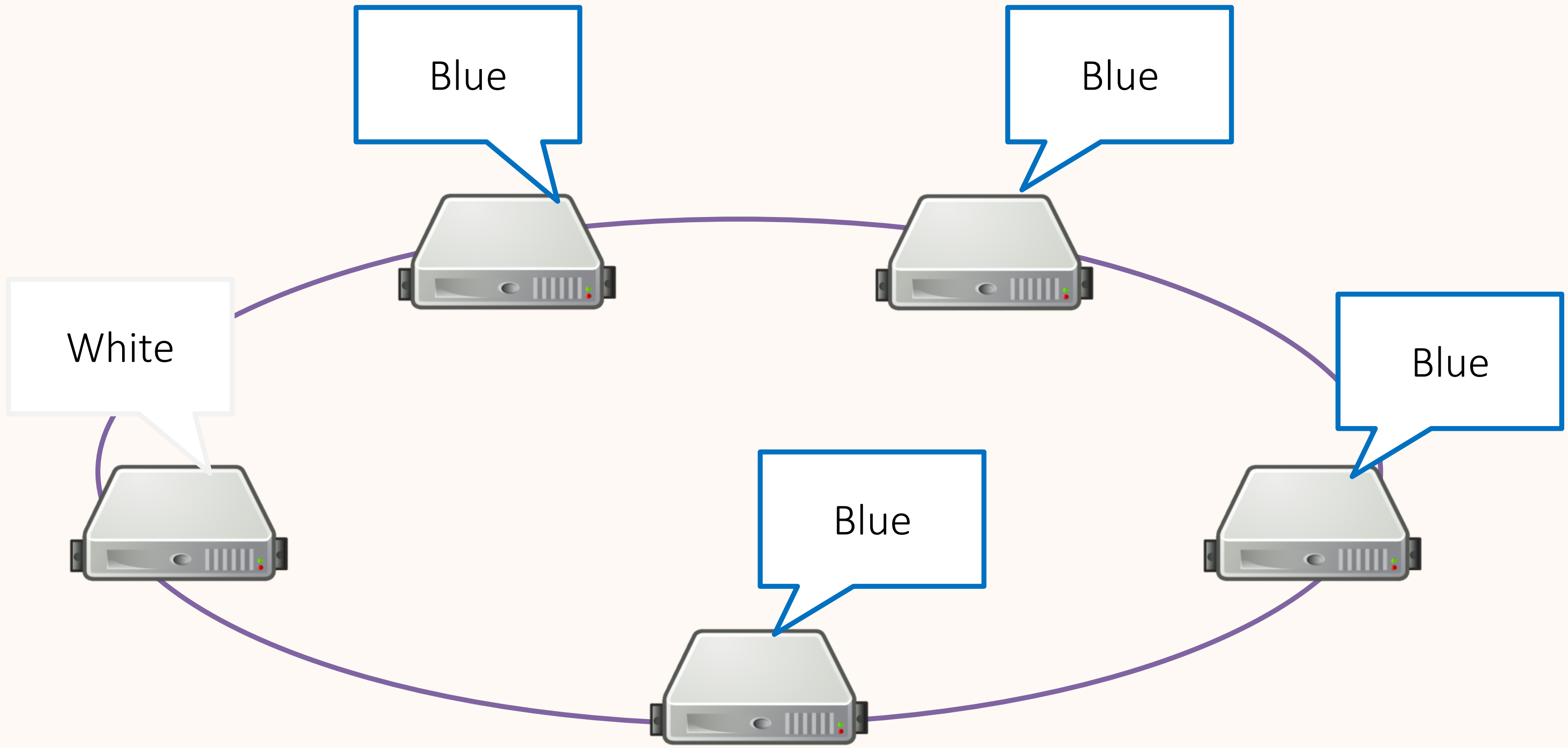
Strong consensus: All nodes need to agree



Consensus.

Distributed Consensus

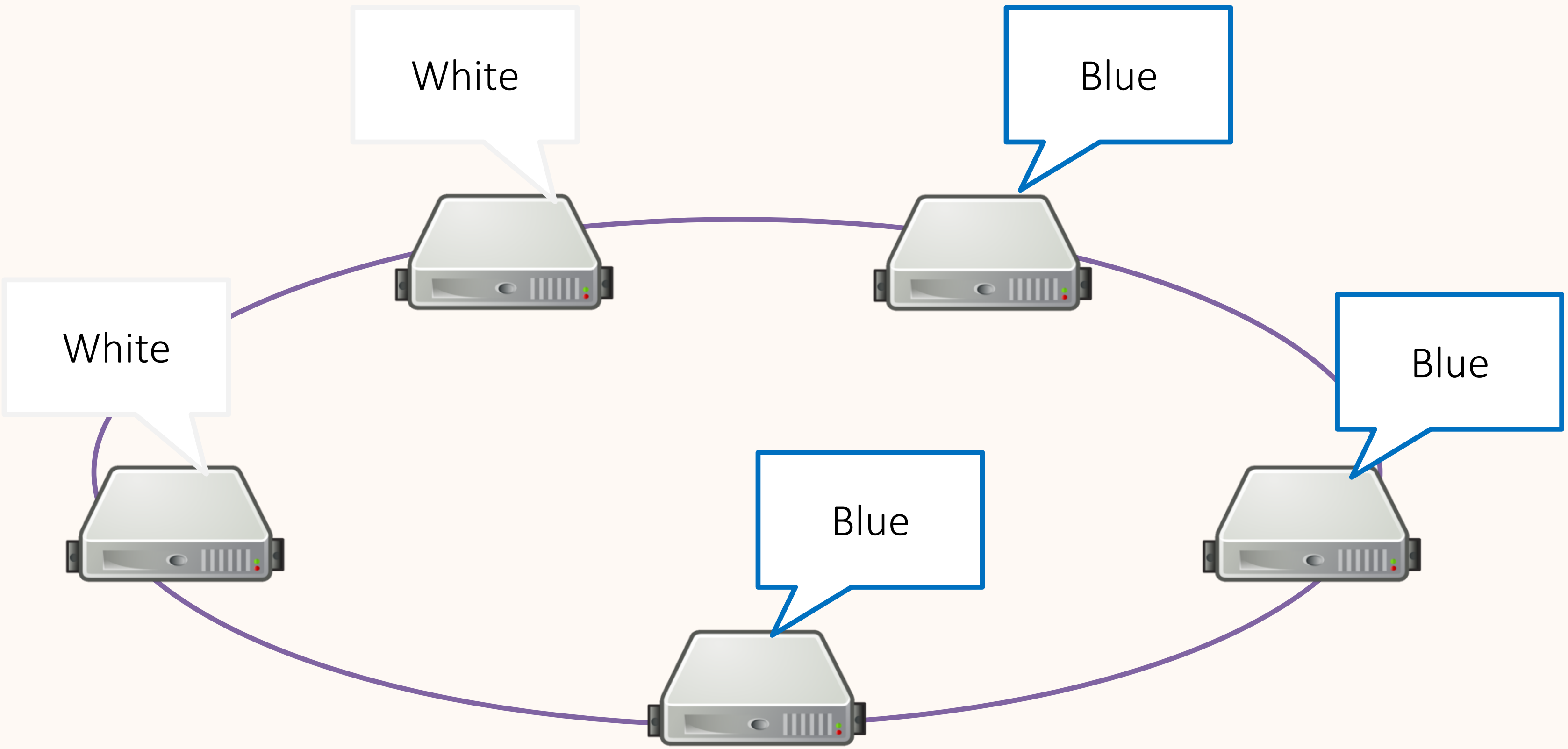
Strong consensus: All nodes need to agree



No consensus.

Distributed Consensus

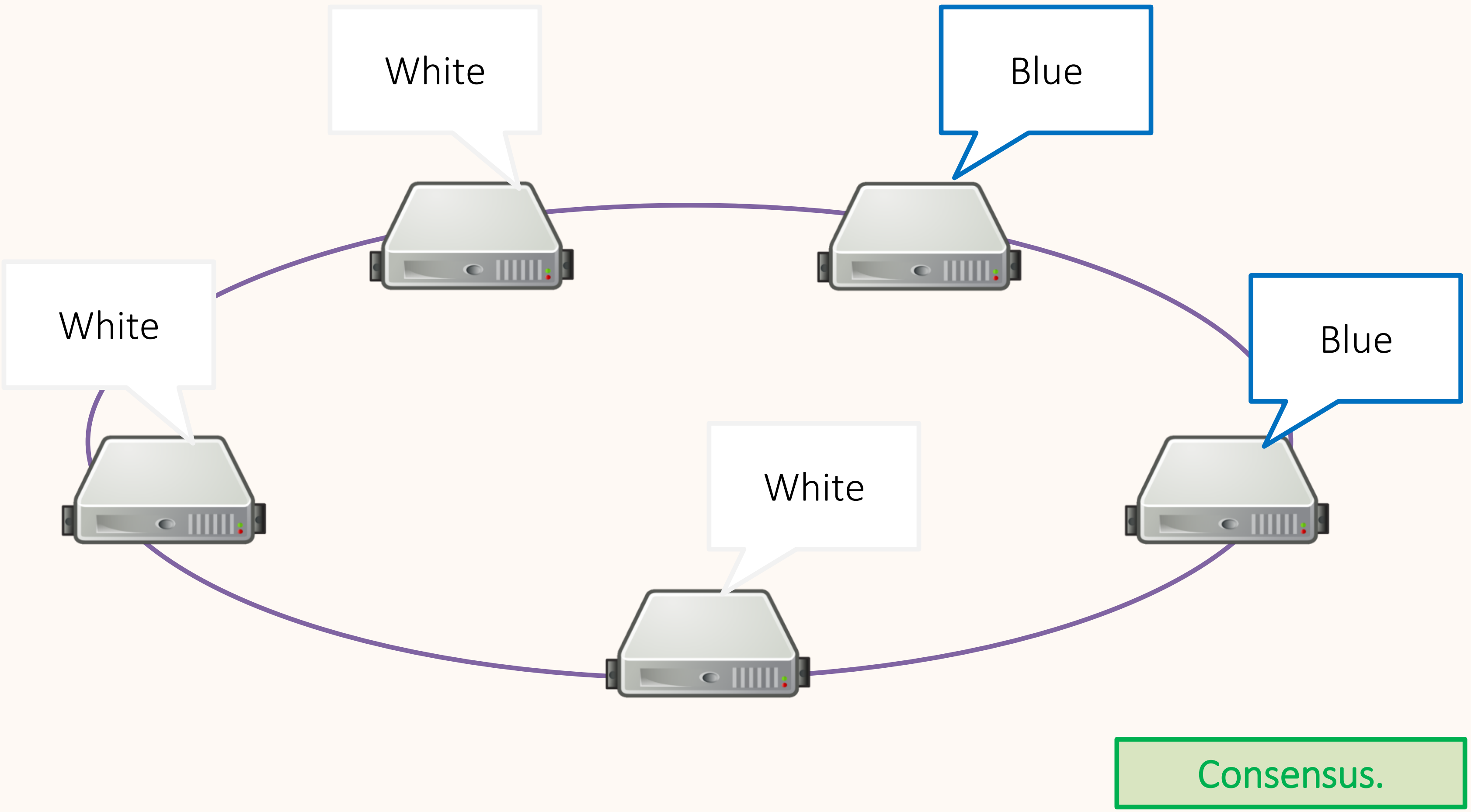
Majority consensus: A majority of nodes need to agree



Consensus.

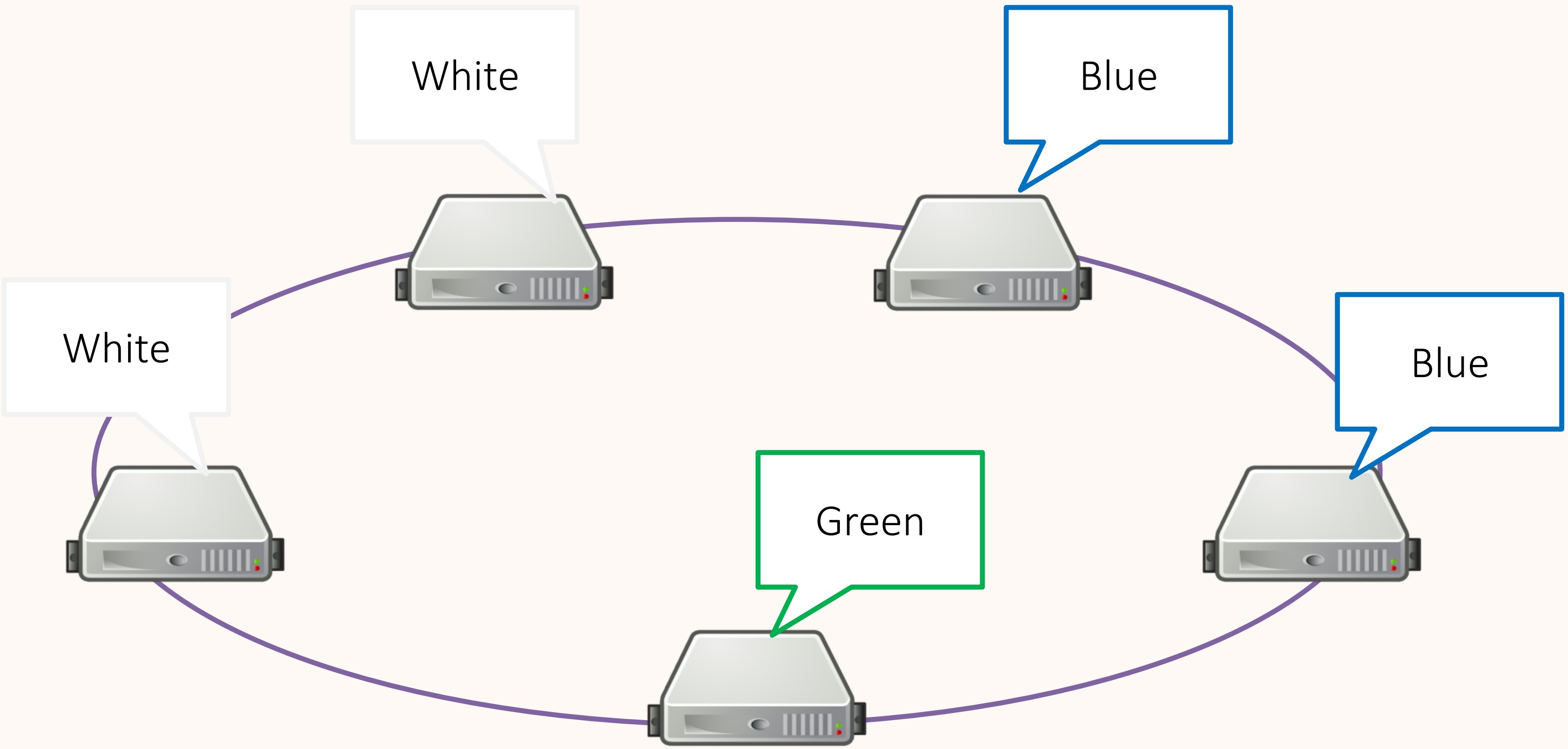
Distributed Consensus

Majority consensus: A majority of nodes need to agree



Distributed Consensus

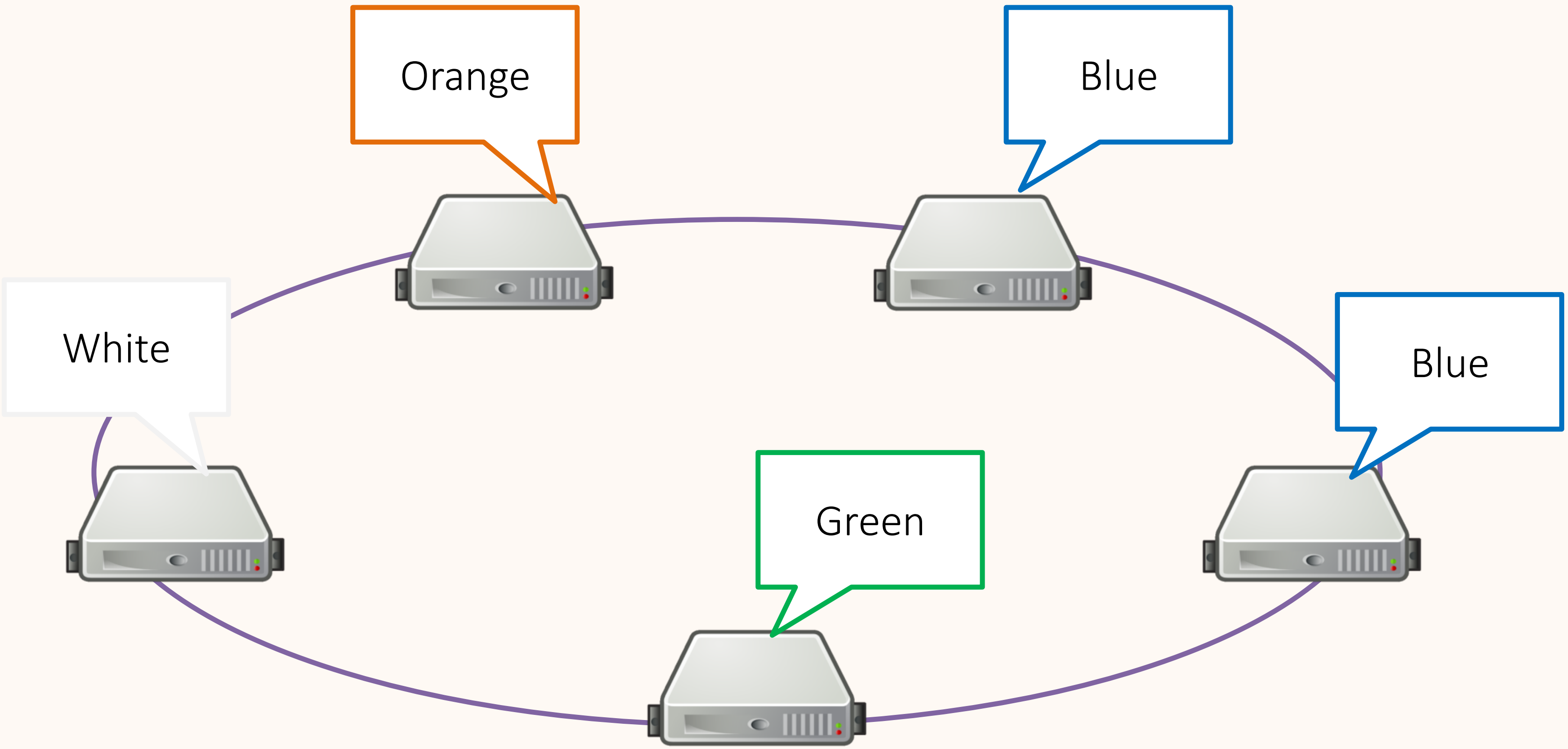
Majority consensus: A majority of nodes need to agree



No consensus.

Distributed Consensus

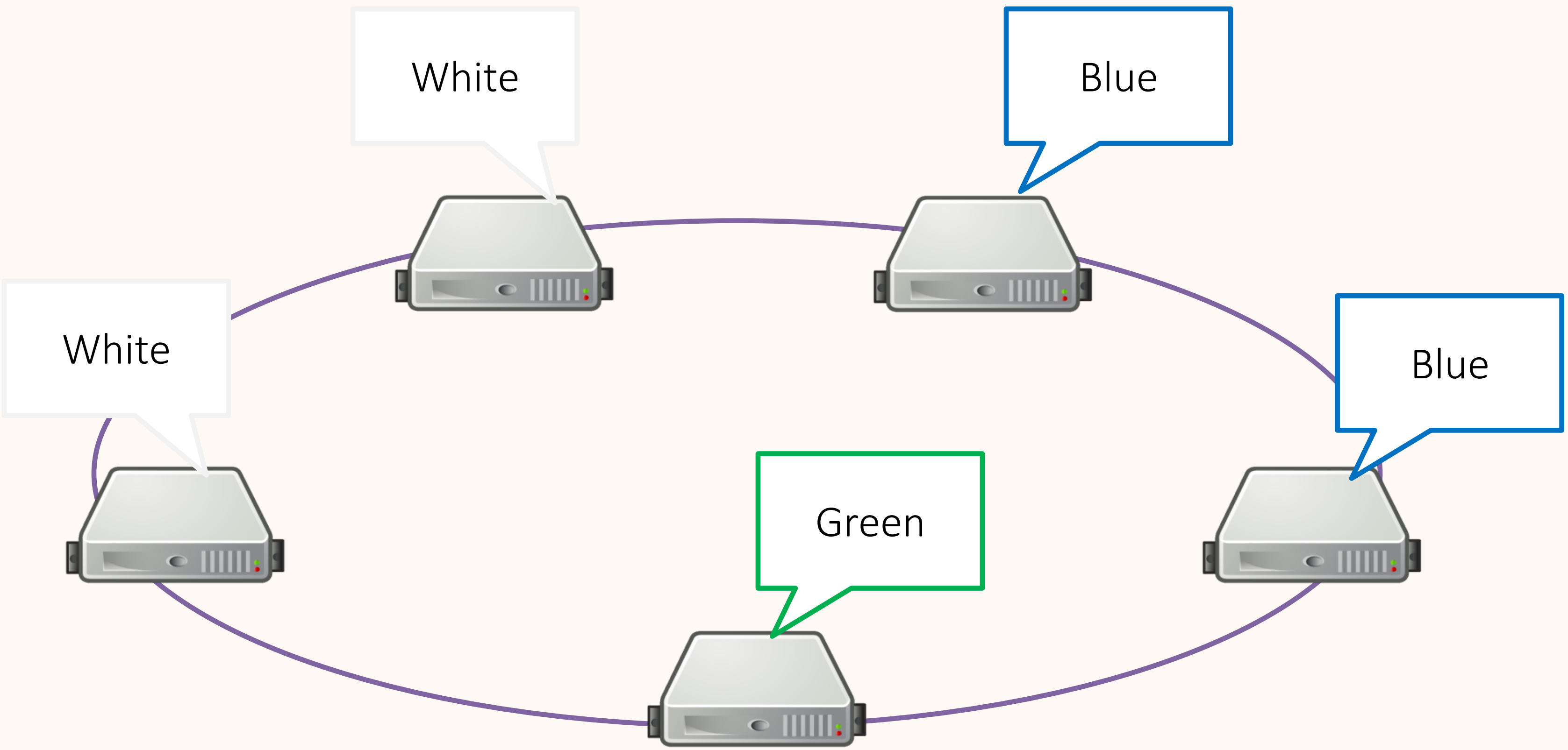
Plurality consensus: A plurality of nodes need to agree



Consensus.

Distributed Consensus

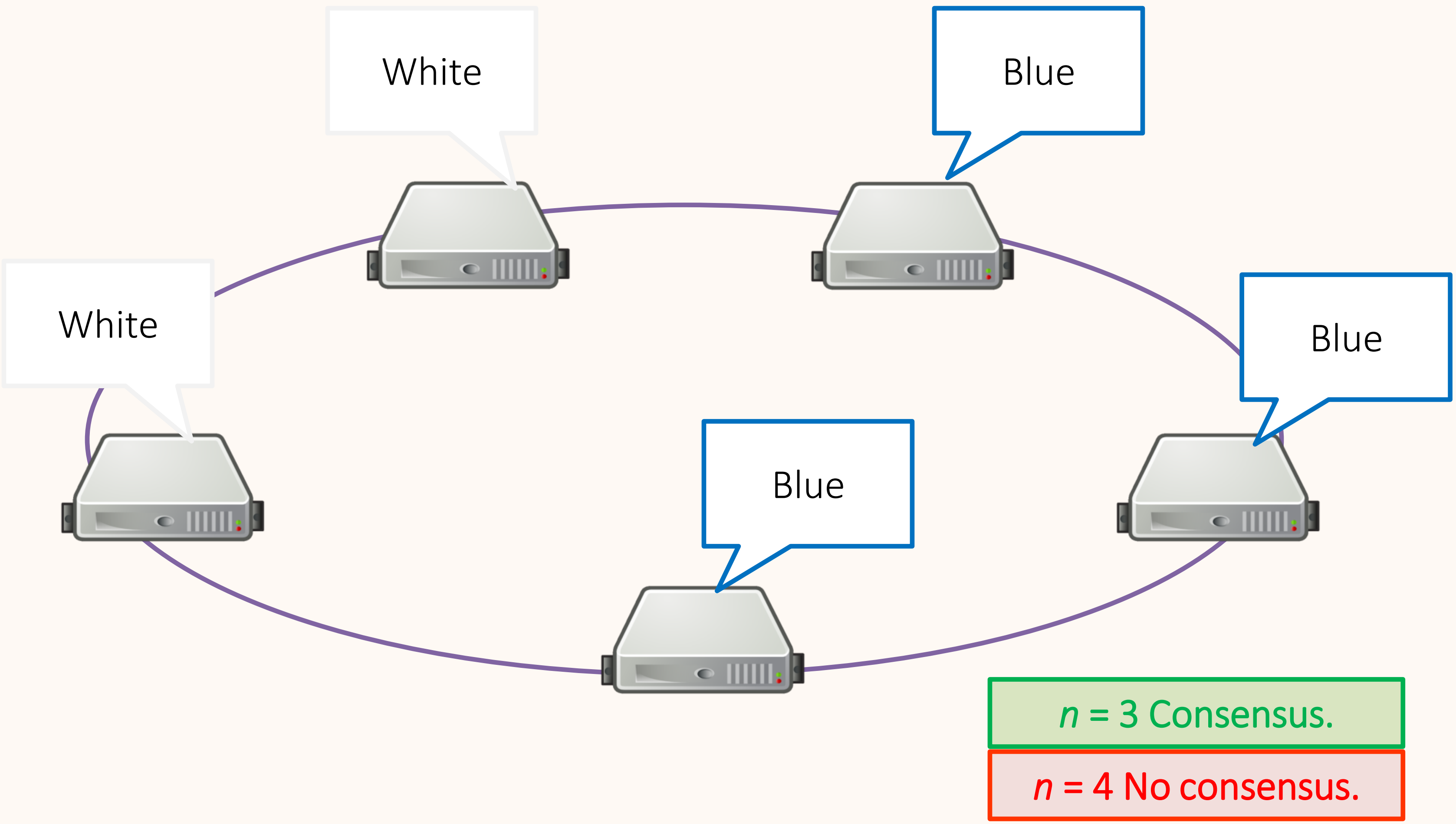
Plurality consensus: A plurality of nodes need to agree



No consensus.

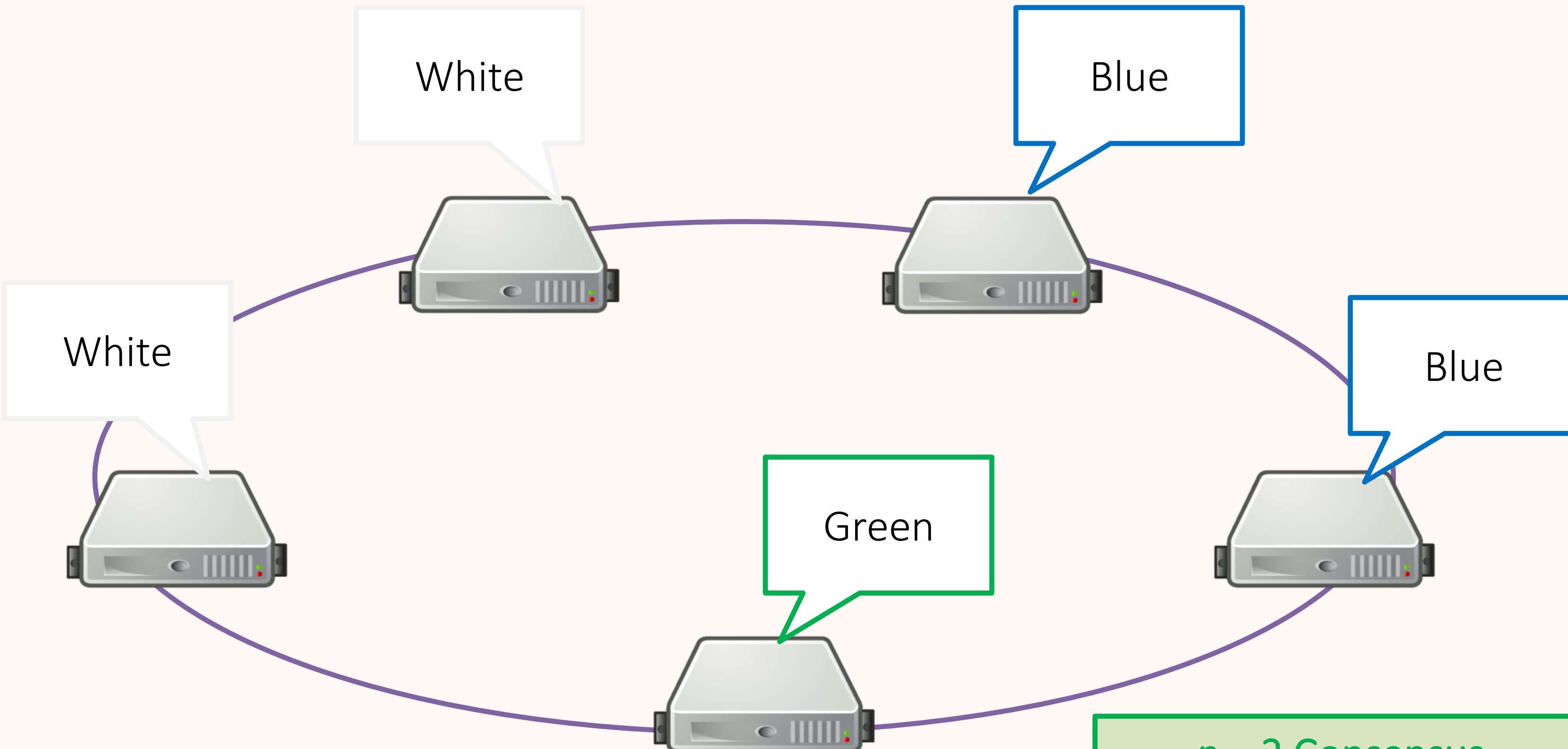
Distributed Consensus

Quorum consensus: n nodes need to agree



Distributed Consensus

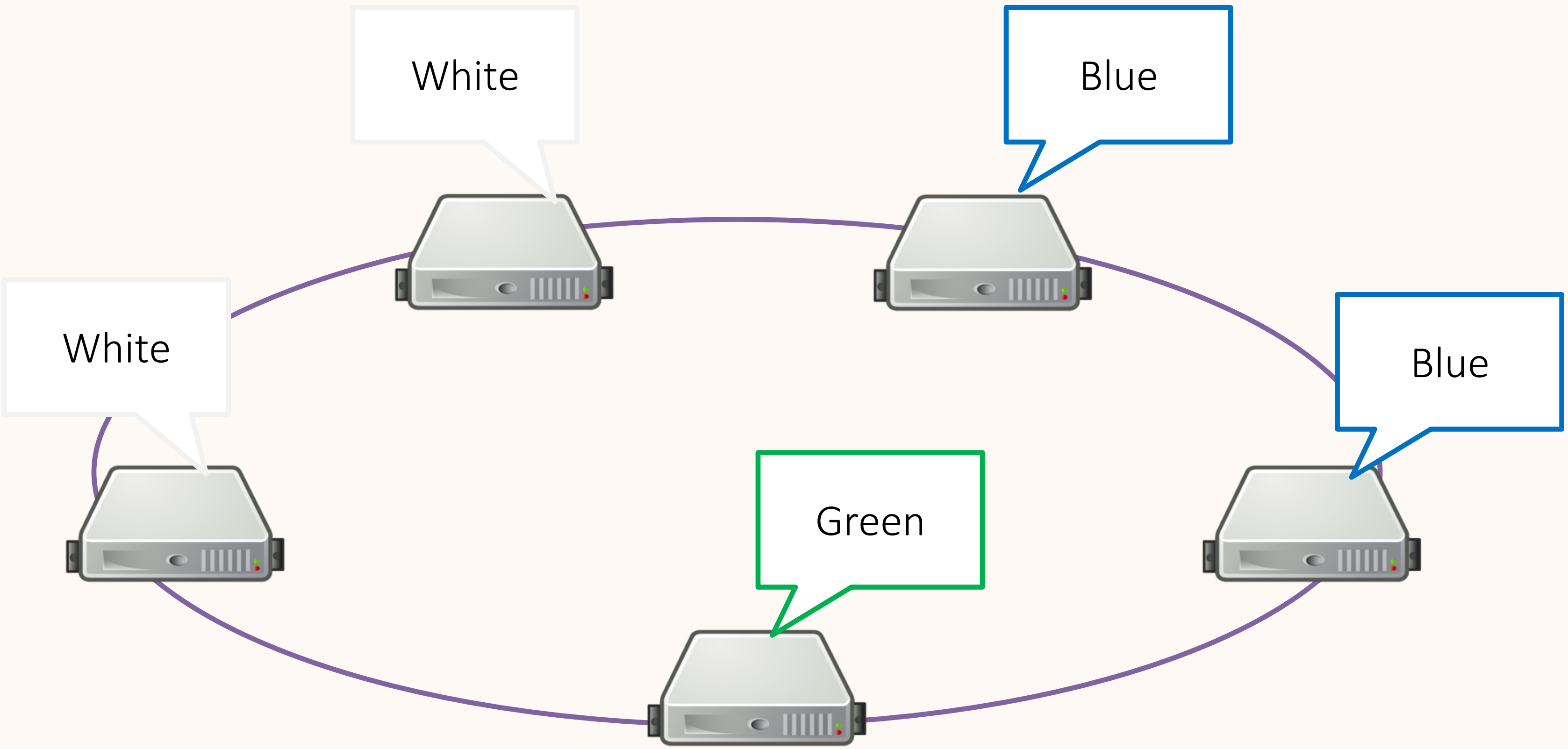
Quorum consensus: n nodes need to agree



n = 2 Consensus.
(First 2 machines asked,
but not unique!)

Distributed Consensus

Quorum consensus: n nodes need to agree

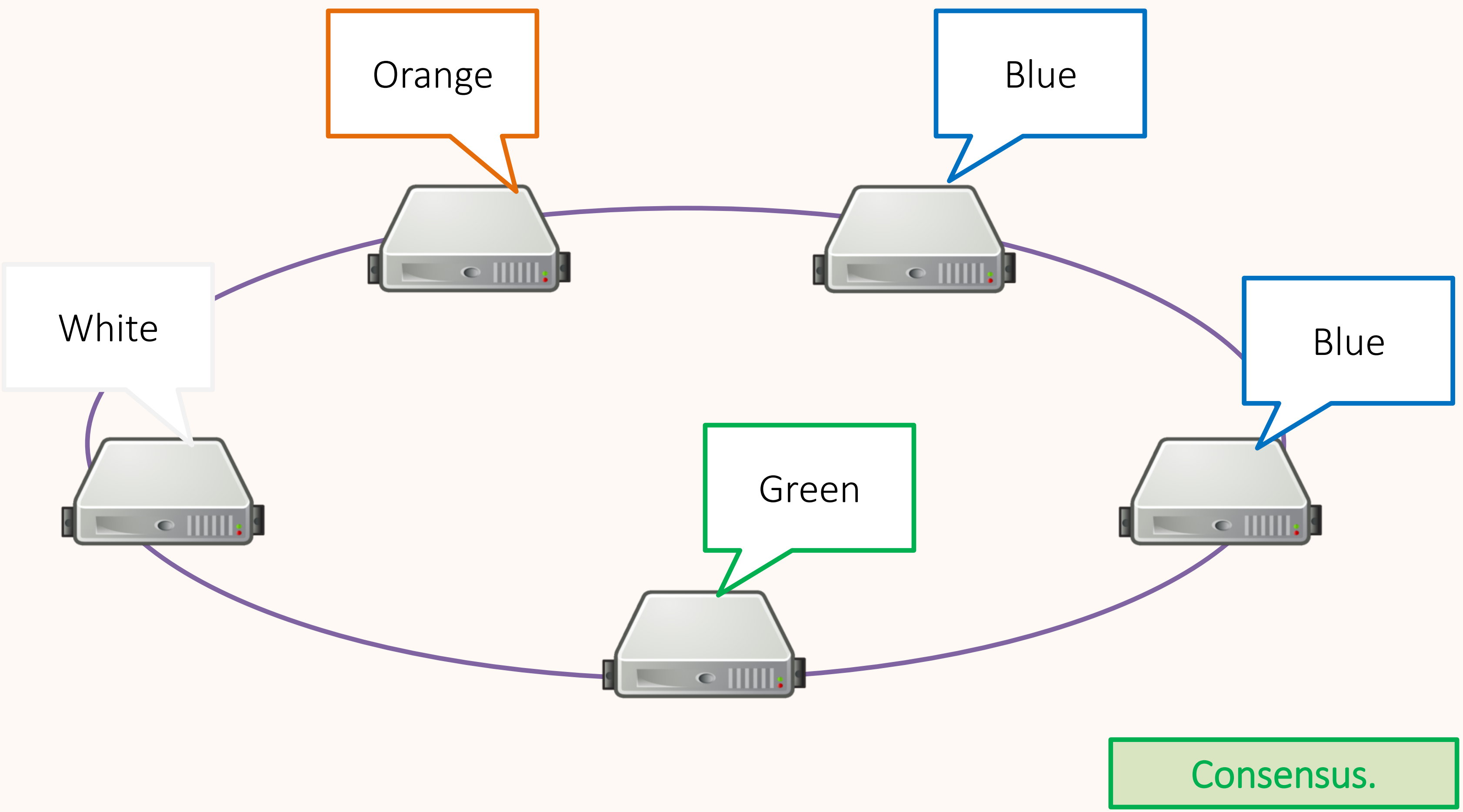


Value of n needed for unique consensus with N nodes?

$n > N/2$

Distributed Consensus

Consensus off: Take first answer



Distributed Consensus

CP vs. AP?

Strong consensus: All nodes need to agree — CP

Majority consensus: A majority of nodes need to agree

Plurality consensus: A plurality of nodes need to agree

Quorom consensus: "Fixed" *n* nodes need to agree

Consensus off: Take first answer — AP

Distributed Consensus

Scale? ?

Strong consensus: All nodes need to agree

More replication

Majority consensus: A majority of nodes need to agree

Plurality consensus: A plurality of nodes need to agree

Quorom consensus: "Fixed" *n* nodes need to agree

Consensus off: Take first answer

Less replication

Distributed Consensus

Strong consensus: All nodes need to agree

Majority consensus: A majority of nodes need to agree

Plurality consensus: A plurality of nodes need to agree

Quorom consensus: "Fixed" *n* nodes need to agree

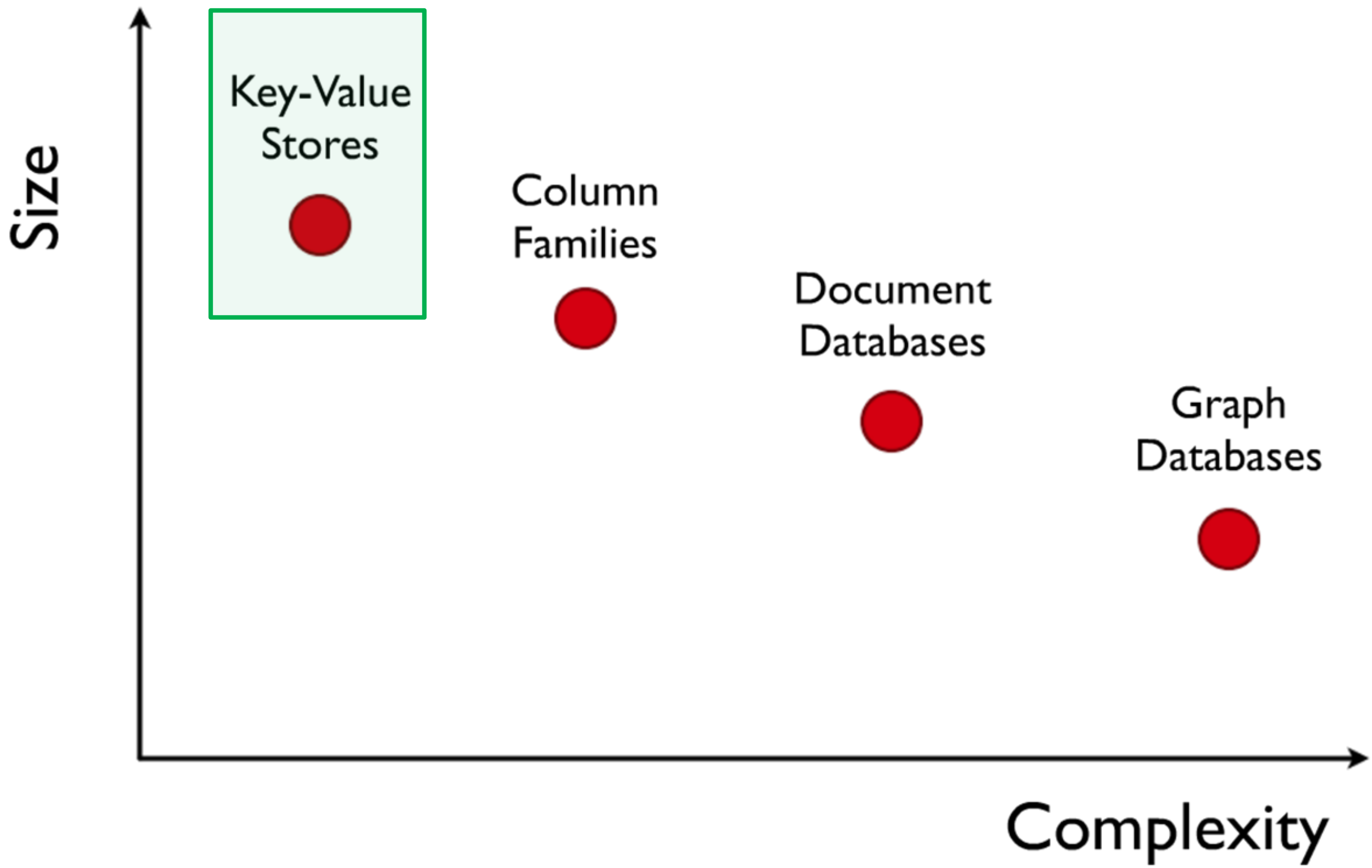
Consensus off: Take first answer

Choice is application dependent:

Many NoSQL stores allow you to choose level of consensus/replication

NoSQL: Key–Value stores

NoSQL: Key–Value Stores



Key–Value Store Model

It's just a Map / Associate Array / Dictionary ☺

- `put(key,value)`
- `get(key)`
- `delete(key)`

Key	Value
Afghanistan	Kabul
Albania	Tirana
Algeria	Algiers
Andorra la Vella	Andorra la Vella
Angola	Luanda
Antigua and Barbuda	St. John's
…	…

But You Can Do a Lot With a Map

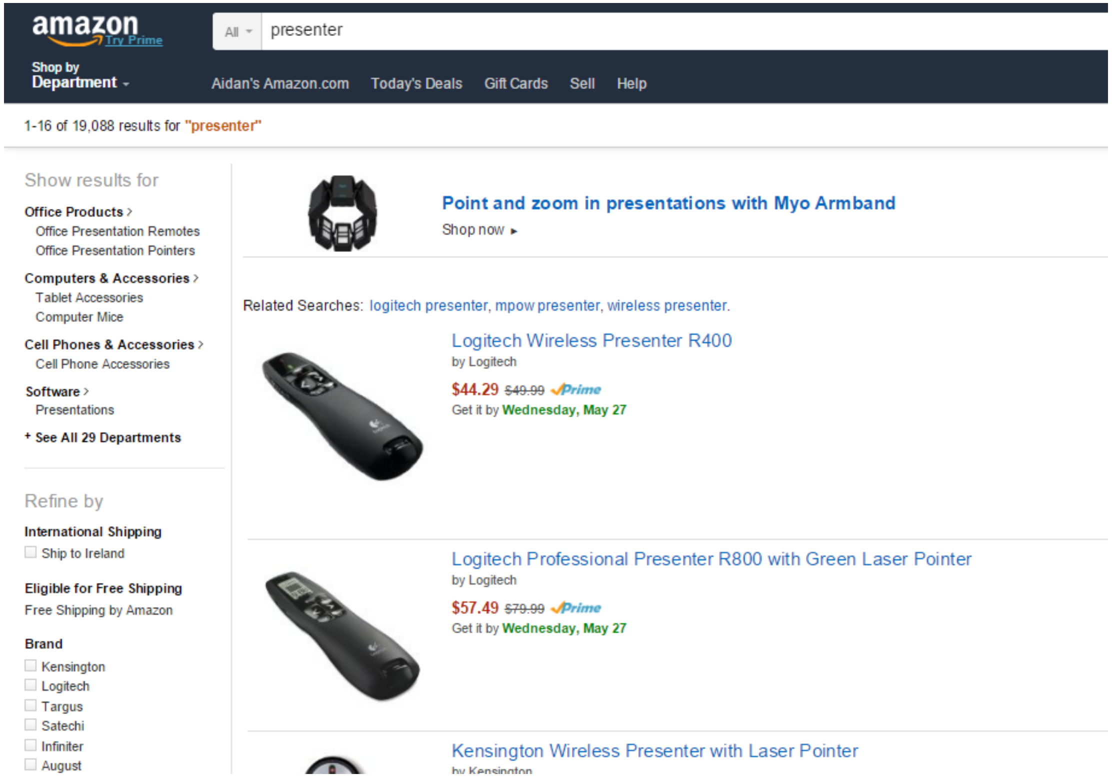
Key	Value
country:Afghanistan	capital@city:Kabul,continent:Asia,pop:31108077#2011
country:Albania	capital@city:Tirana,continent:Europe,pop:3011405#2013
…	…
city:Kabul	country:Afghanistan,pop:3476000#2013
city:Tirana	country:Albania,pop:3011405#2013
…	…
user:10239	basedIn@city:Tirana,post:{103,10430,201}
…	…

… actually you can model any data in a map (but possibly with a lot of redundancy and inefficient lookups if unsorted).

The Case of Amazon

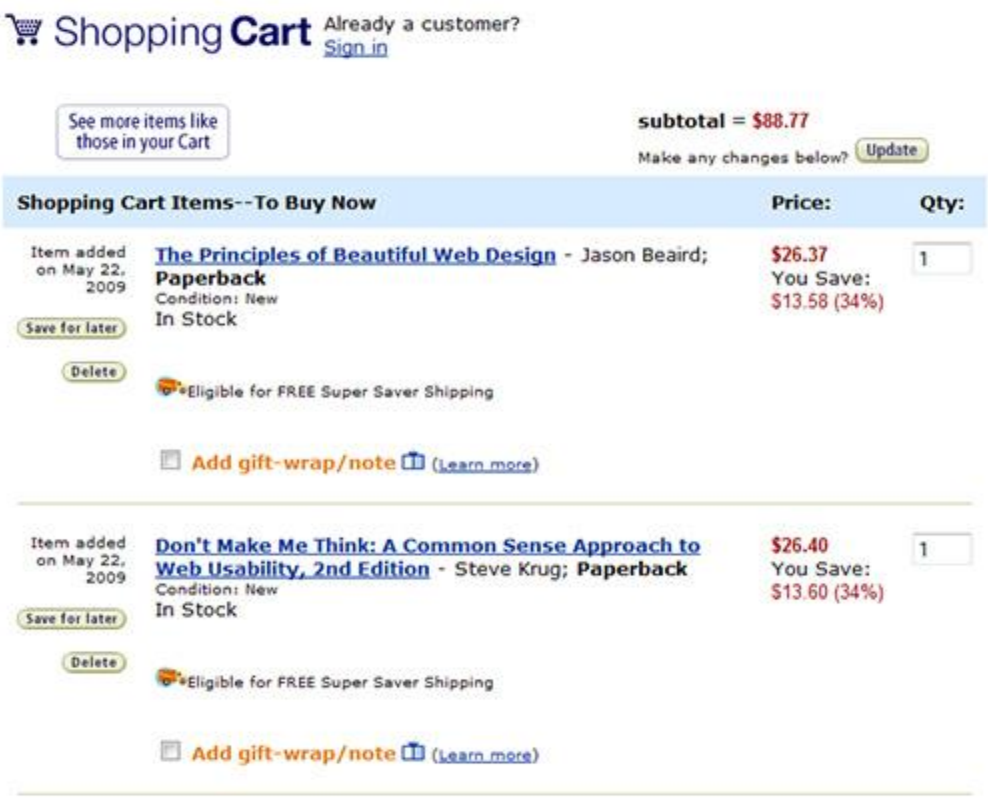
The Amazon Scenario

Products Listings: prices, details, stock



The Amazon Scenario

Customer info: shopping cart, account, etc.

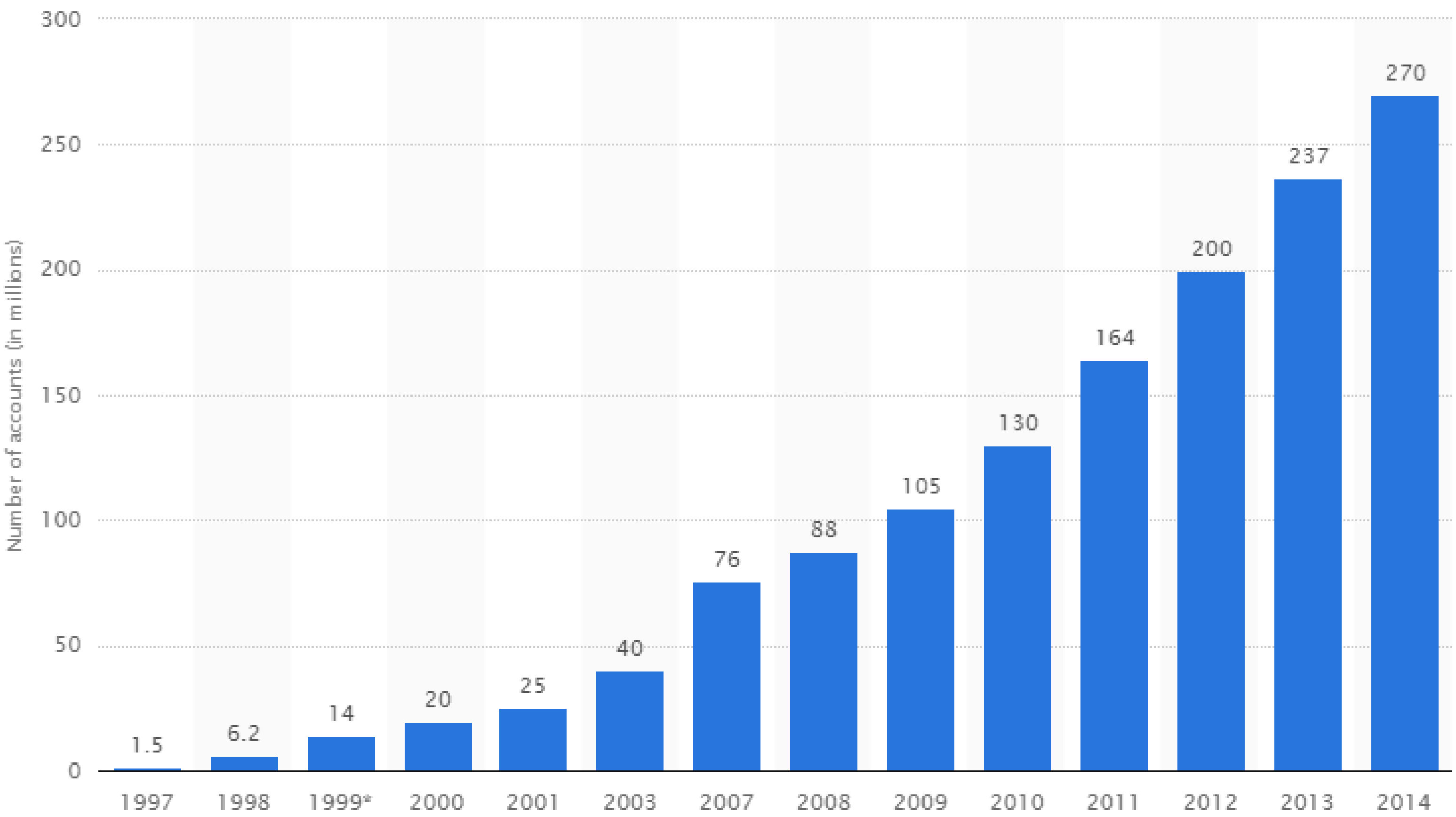


The Amazon Scenario

Recommendations, etc.:

The Amazon Scenario

- Amazon customers:

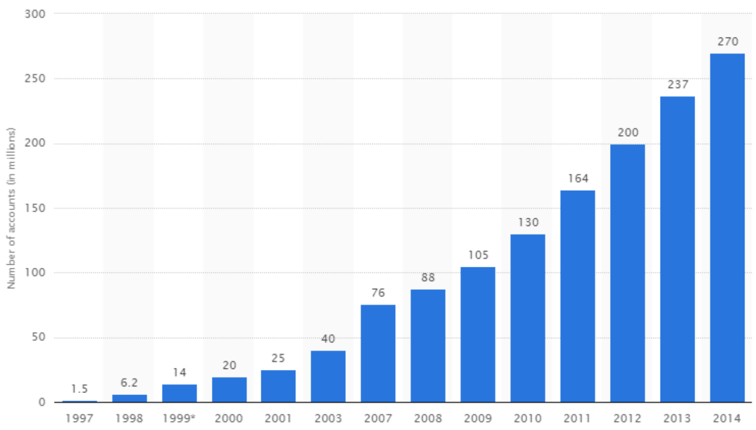


The Amazon Scenario

amazon web services™

The Amazon Scenario

Databases struggling …



But many Amazon services don't need:

- SQL (a simple map often enough)

or even:

- transactions, strong consistency, etc.

Key–Value Store: Amazon Dynamo(DB)

Dynamo: Amazon's Highly Available Key-value Store

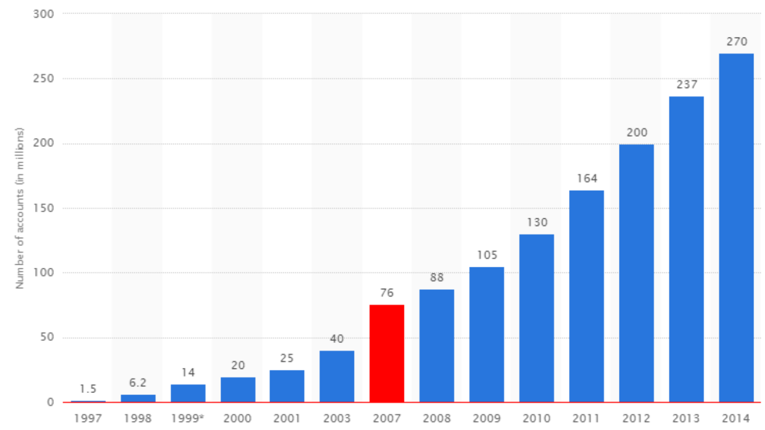
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

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

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



Goals:

- Scalability (able to grow)
- High availability (reliable)
- Performance (fast)

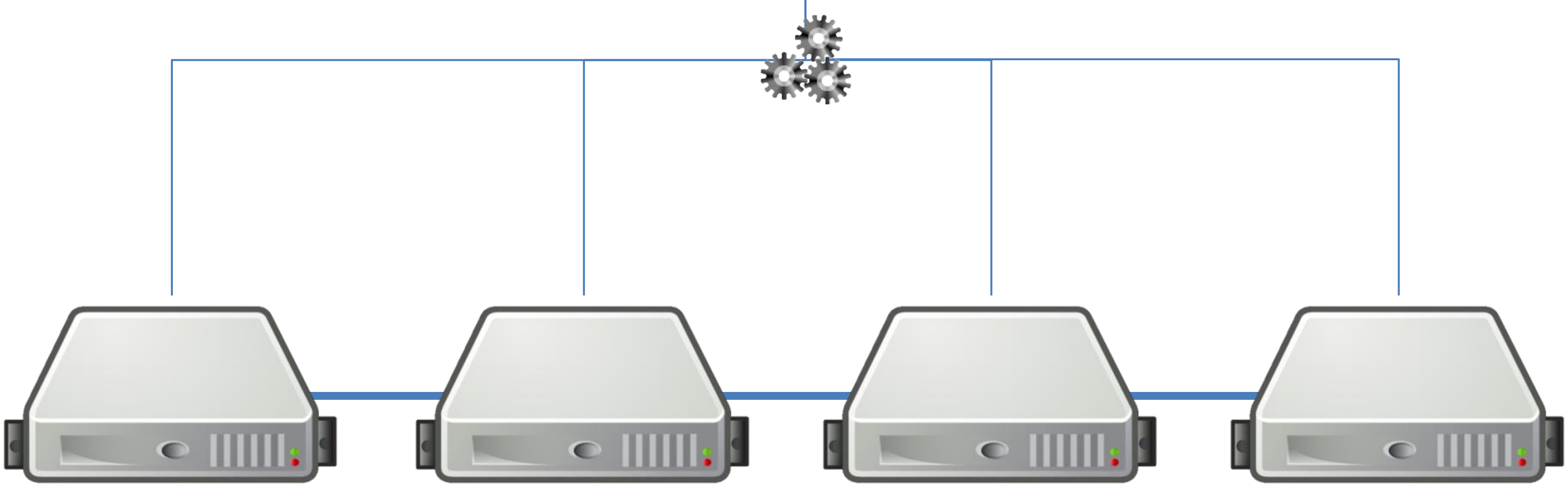
Don't need full SQL, don't need full ACID

Key–Value Store: Distribution

How might we distribute a key–value store over multiple machines?

Key	Value
country:Afghanistan	capital@city:Kabul,continent:Asia,pop:31108077#2011
country:Albania	capital@city:Tirana,continent:Europe,pop:3011405#2013
...	...
city:Kabul	country:Afghanistan,pop:3476000#2013
city:Tirana	country:Albania,pop:3011405#2013
...	...
user:10239	basedIn@city:Tirana,post:{103,10430,201}
...	...

$$\boxed{\mathrm{mod}(\mathrm{hash}(key), m)}$$



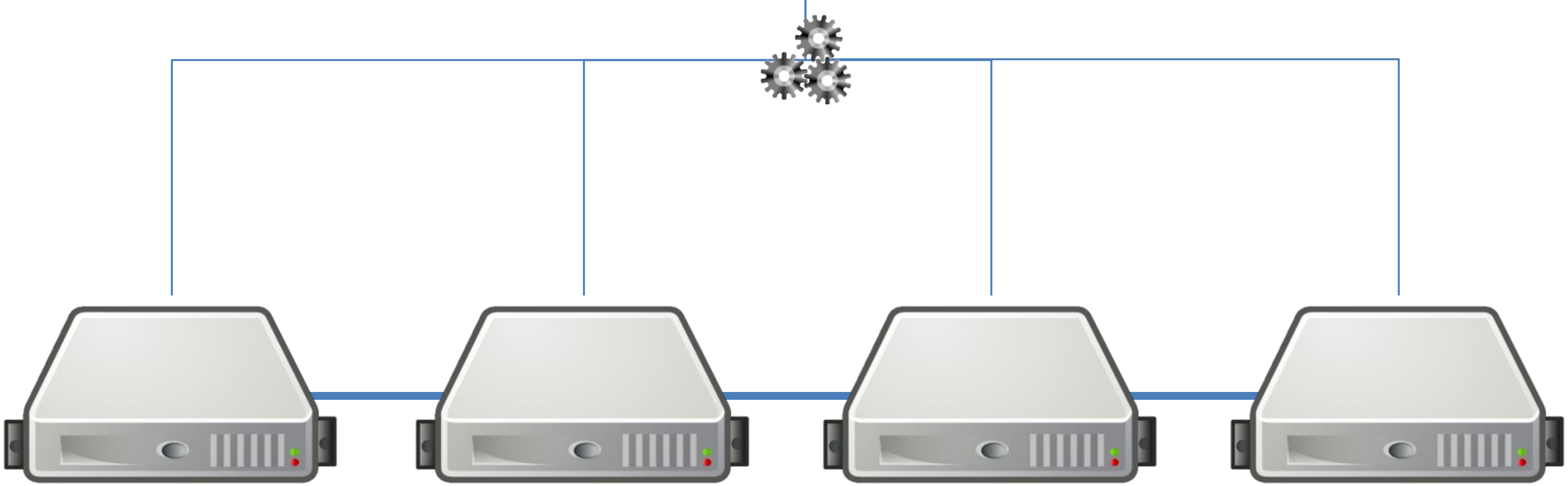
Key–Value Store: Distribution

What happens if a machine leaves or joins afterwards?

How can we avoid rehashing everything?

Key	Value
country:Afghanistan	capital@city:Kabul,continent:Asia,pop:31108077#2011
country:Albania	capital@city:Tirana,continent:Europe,pop:3011405#2013
...	...
city:Kabul	country:Afghanistan,pop:3476000#2013
city:Tirana	country:Albania,pop:3011405#2013
...	...
user:10239	basedIn@city:Tirana,post:{103,10430,201}
...	...

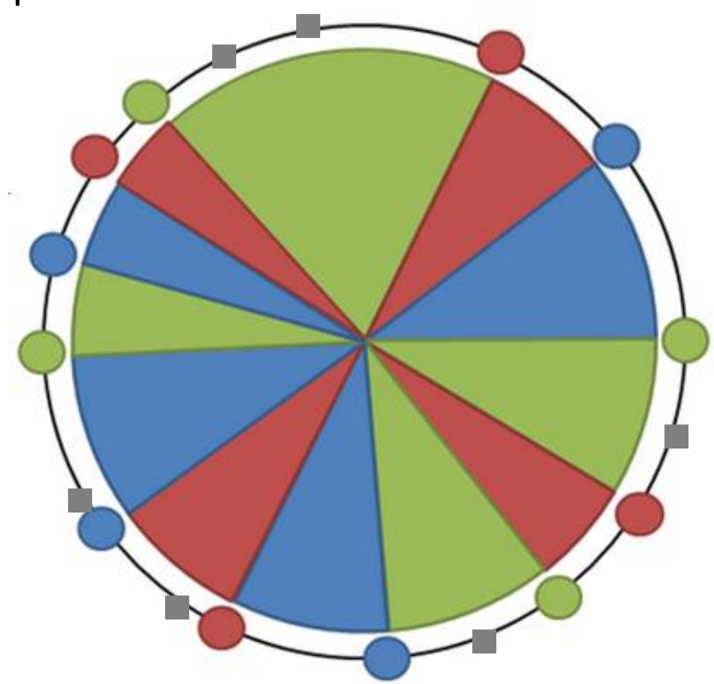
$$\boxed{\mathrm{mod}(\mathrm{hash}(key), m)}$$



Consistent Hashing

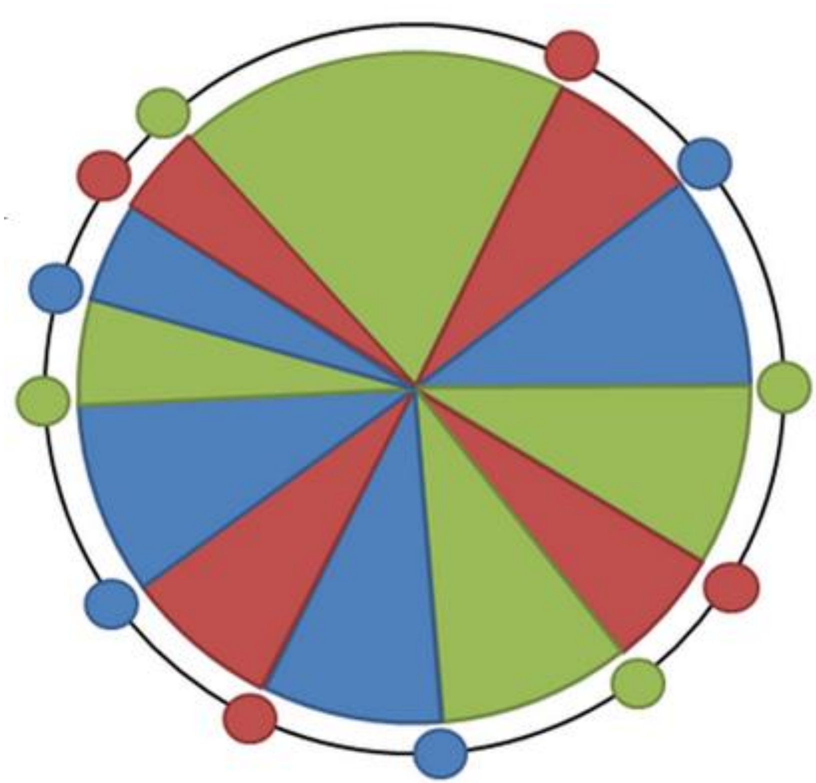
Avoid re-hashing everything

- Hash using a ring
- Each machine picks n pseudo-random points on the ring
- Machine responsible for arc after its point
- Objects mapped to ring
- If a machine leaves, its range moves to previous machine
- If a machine joins, it picks new points



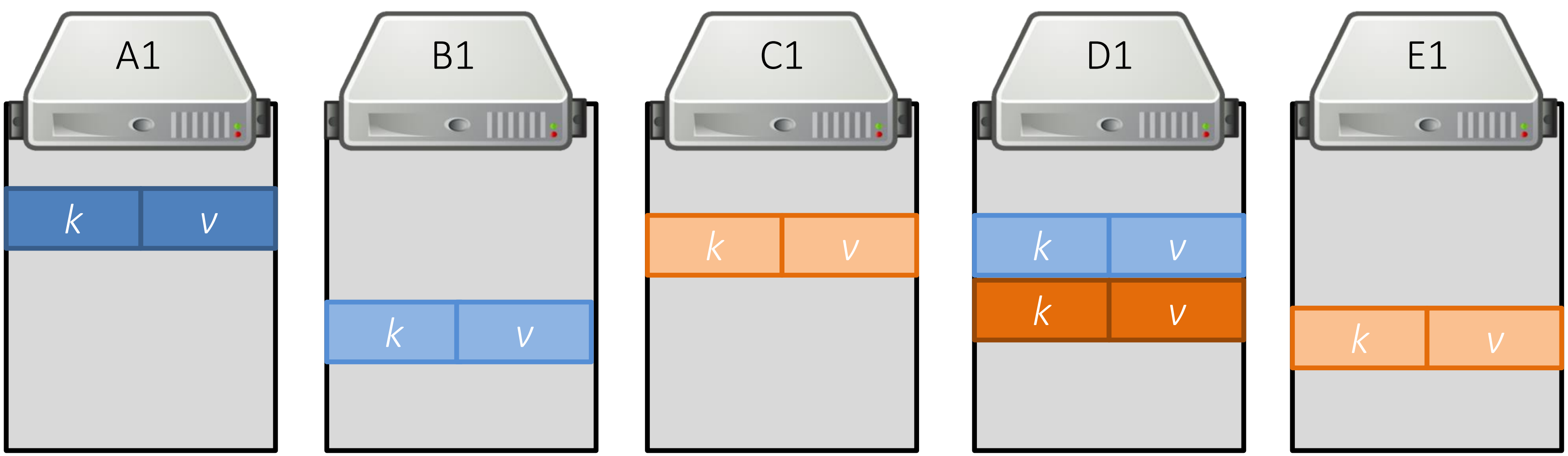
Amazon Dynamo: Hashing

- Consistent Hashing (128-bit MD5)



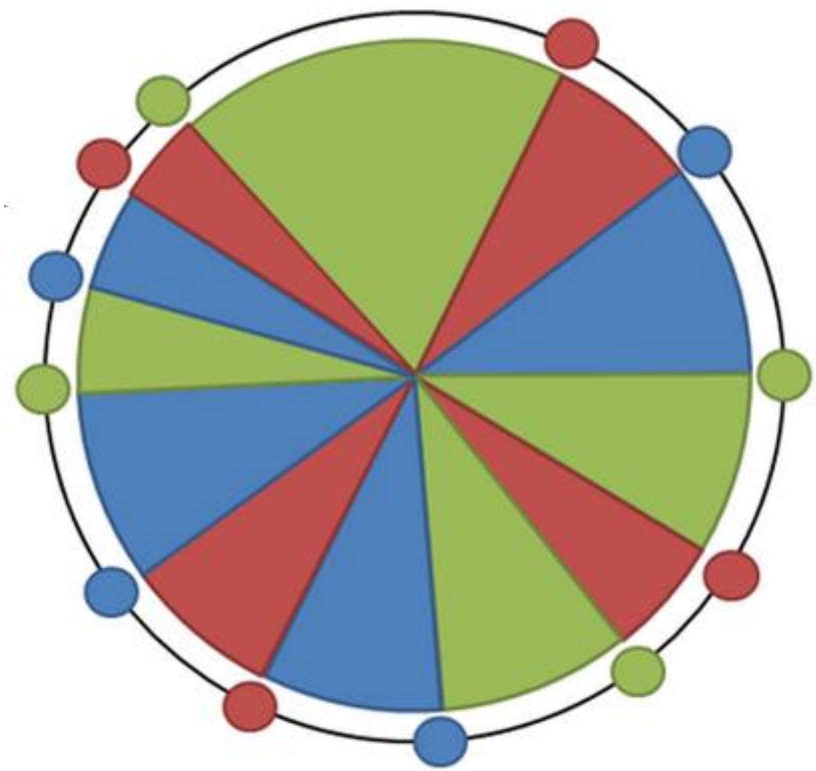
Amazon Dynamo: Replication

- A set replication factor (e.g., 3)
- Commonly primary / secondary replicas
  - Primary replica elected from secondary replicas in the case of failure of primary



Amazon Dynamo: Replication

- Replication factor of n?
  - Easy: pick n next buckets (different machines!)



Amazon Dynamo: Model

- Named table with primary key and a value
- Primary key is hashed / unordered

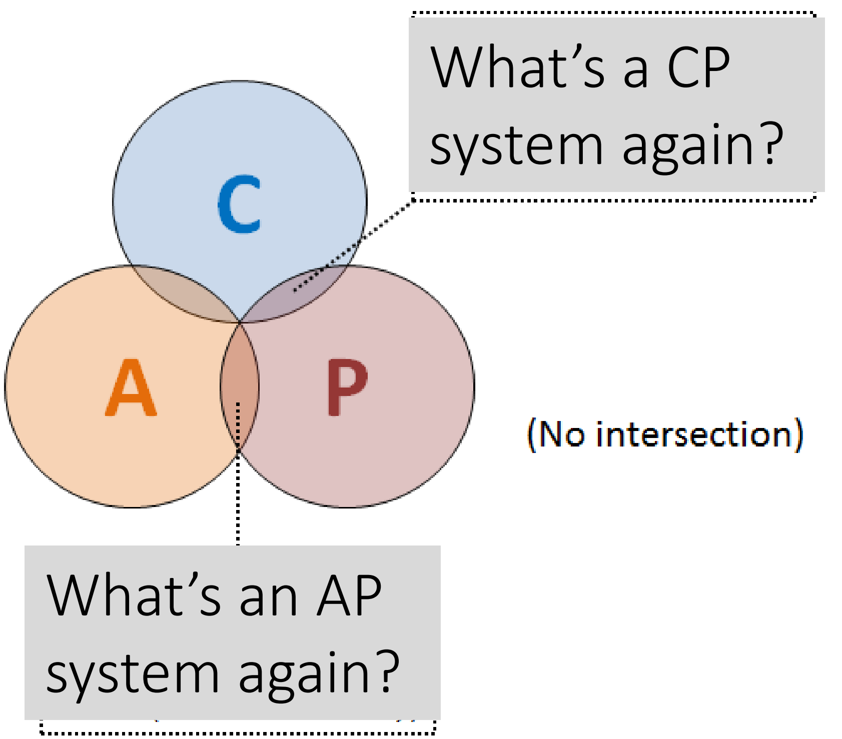
Countries	
Primary Key	**Value**
Afghanistan	capital:Kabul,continent:Asia,pop:31108077#2011
Albania	capital:Tirana,continent:Europe,pop:3011405#2013
…	…

Cities	
Primary Key	**Value**
Kabul	country:Afghanistan,pop:3476000#2013
Tirana	country:Albania,pop:3011405#2013
…	…

Amazon Dynamo: CAP

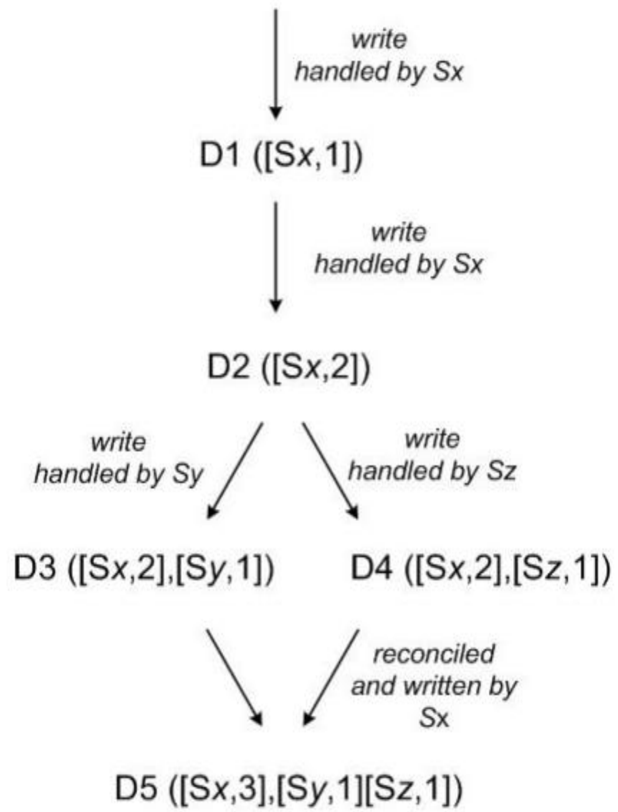
Two options for each table:

- AP: Eventual consistency, High availability
- CP: Strong consistency, Lower availability



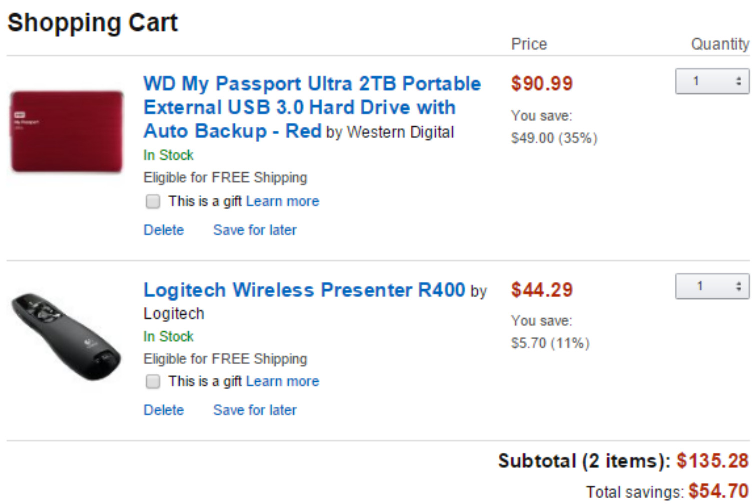
Amazon Dynamo: Consistency

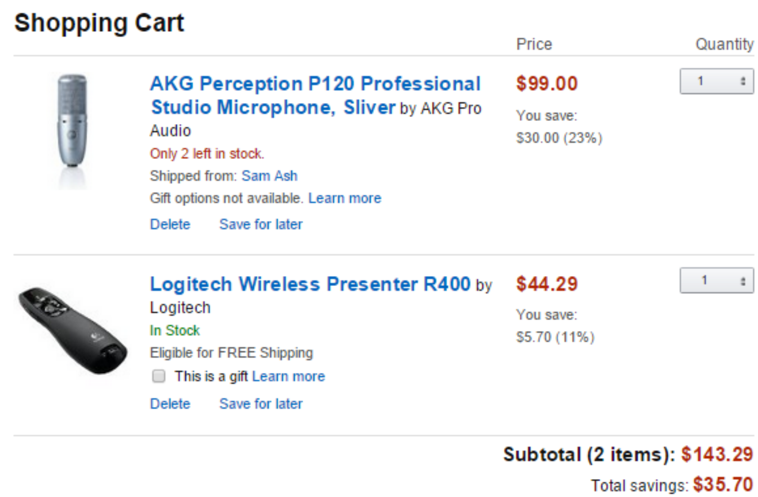
- Vector Clock:
  - A list of pairs indicating a node and operation count
  - Used to track branches of revisions



Amazon Dynamo: Consistency

- Two versions of one shopping cart:





How best to merge multiple conflicting versions of a value (known as reconciliation)?

Application knows best

(… and must support multiple versions being returned)

Amazon Dynamo: Consistency

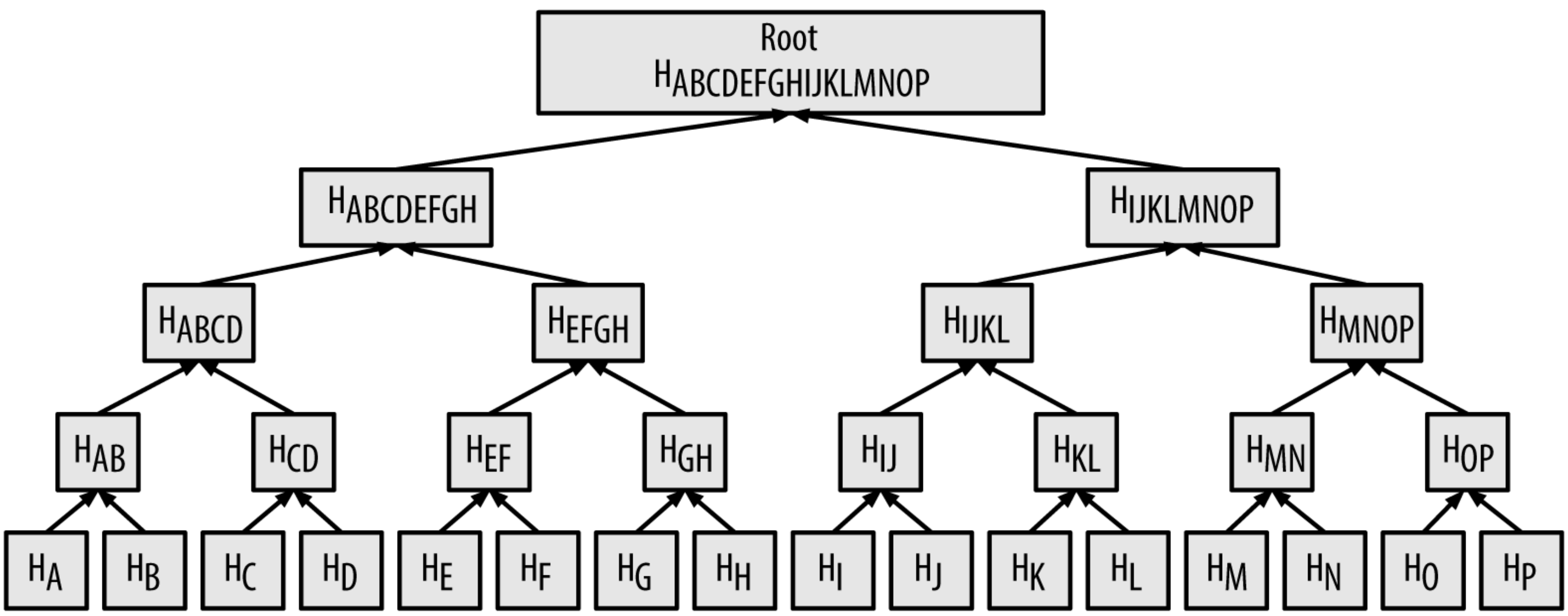
Key	Value
country:Afghanistan	capital@city:Kabul,continent:Asia,pop:31108077#2011
country:Albania	capital@city:Tirana,continent:Europe,pop:3011405#2013
…	…
city:Kabul	country:Afghanistan,pop:3476000#2013
city:Tirana	country:Albania,pop:3011405#2013
…	…
user:10239	basedIn@city:Tirana,post:{103,10430,201,408}
…	…

Key	Value
country:Afghanistan	capital@city:Kabul,continent:Asia,pop:31108077#2011
country:Albania	capital@city:Tirana,continent:Europe,pop:3011405#2013
…	…
city:Kabul	country:Afghanistan,pop:3476000#2013
city:Tirana	country:Albania,pop:3011405#2013
…	…
user:10239	basedIn@city:Tirana,post:{103,10430,201}
…	…

How can we efficiently verify that two copies of a block of data are the same (and find where the differences are)?

Amazon Dynamo: Merkle Trees

- Merkle tree: A hash tree
  - Leaf node compute hashes from data
  - Non-leaf nodes have hashes of their children
  - Find differences between two trees level-by-level



Read More ...

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

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

Other Key–Value stores

Other Key–Value Stores

Other Key–Value Stores

Other Key–Value Stores

cassandra

Adobe

Instagram
Fast beautiful photo sharing

accenture
High performance. Delivered.

Answers.com

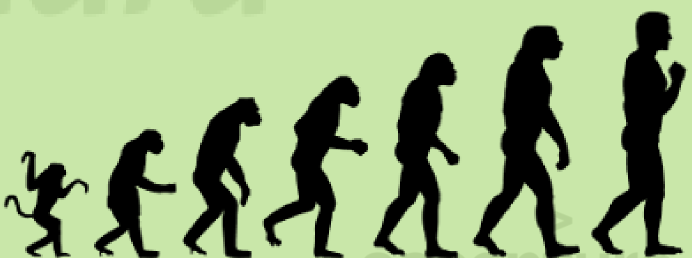
DELL

Disney

CERN

NASA

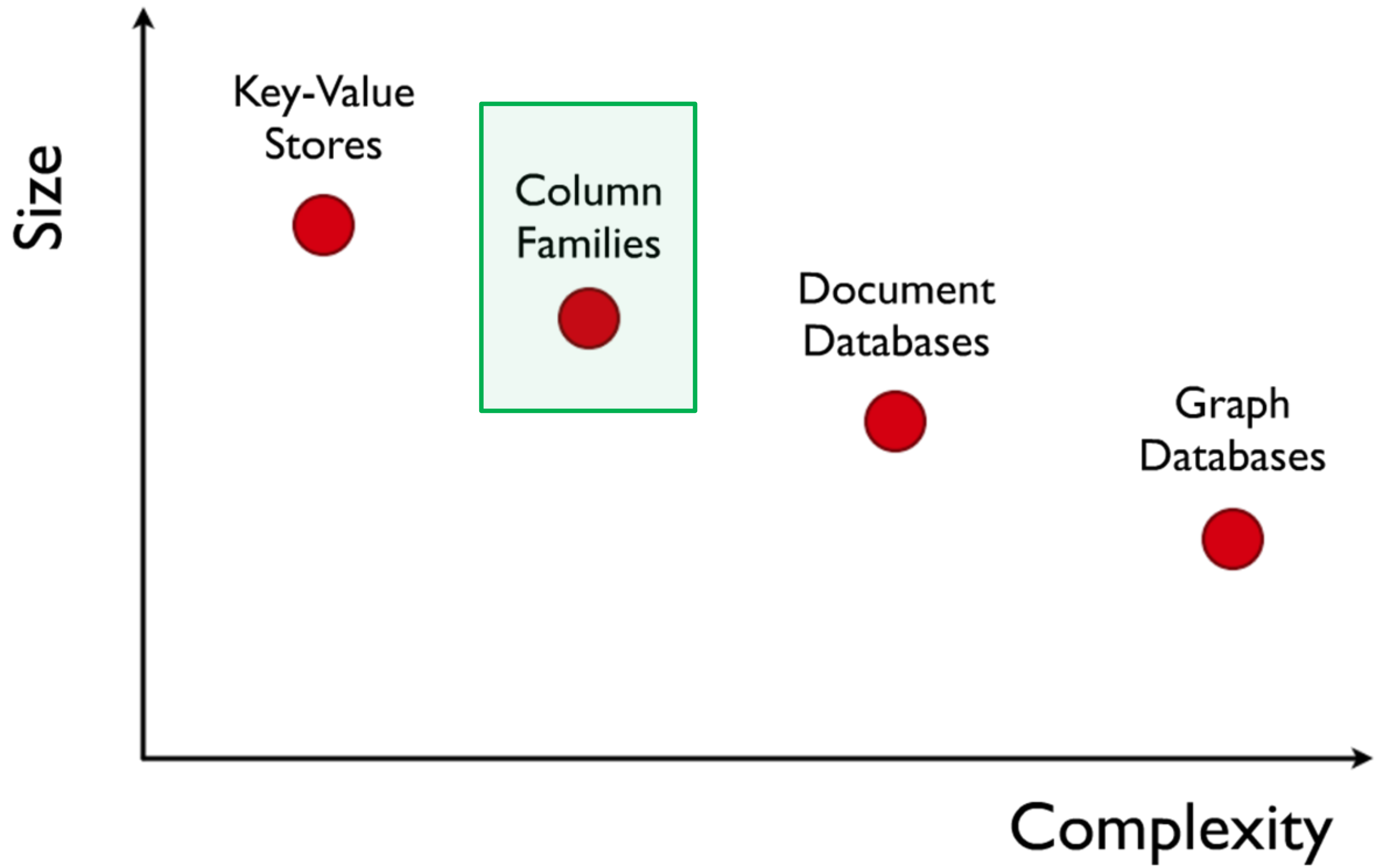
Other Key–Value Stores



Evolved into a
tabular store …

Tabular / Column Family

NoSQL: Column Family Stores



Key–Value = a Distributed Map

Countries	
Primary Key	**Value**
Afghanistan	capital:Kabul,continent:Asia,pop:31108077#2011
Albania	capital:Tirana,continent:Europe,pop:3011405#2013
...	...

Tabular = Multi-dimensional Maps

Countries				
Primary Key	**capital**	**continent**	**pop-value**	**pop-year**
Afghanistan	Kabul	Asia	31108077	2011
Albania	Tirana	Europe	3011405	2013
...	...	...	...	...

Bigtable: The Original Whitepaper

MapReduce authors

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.

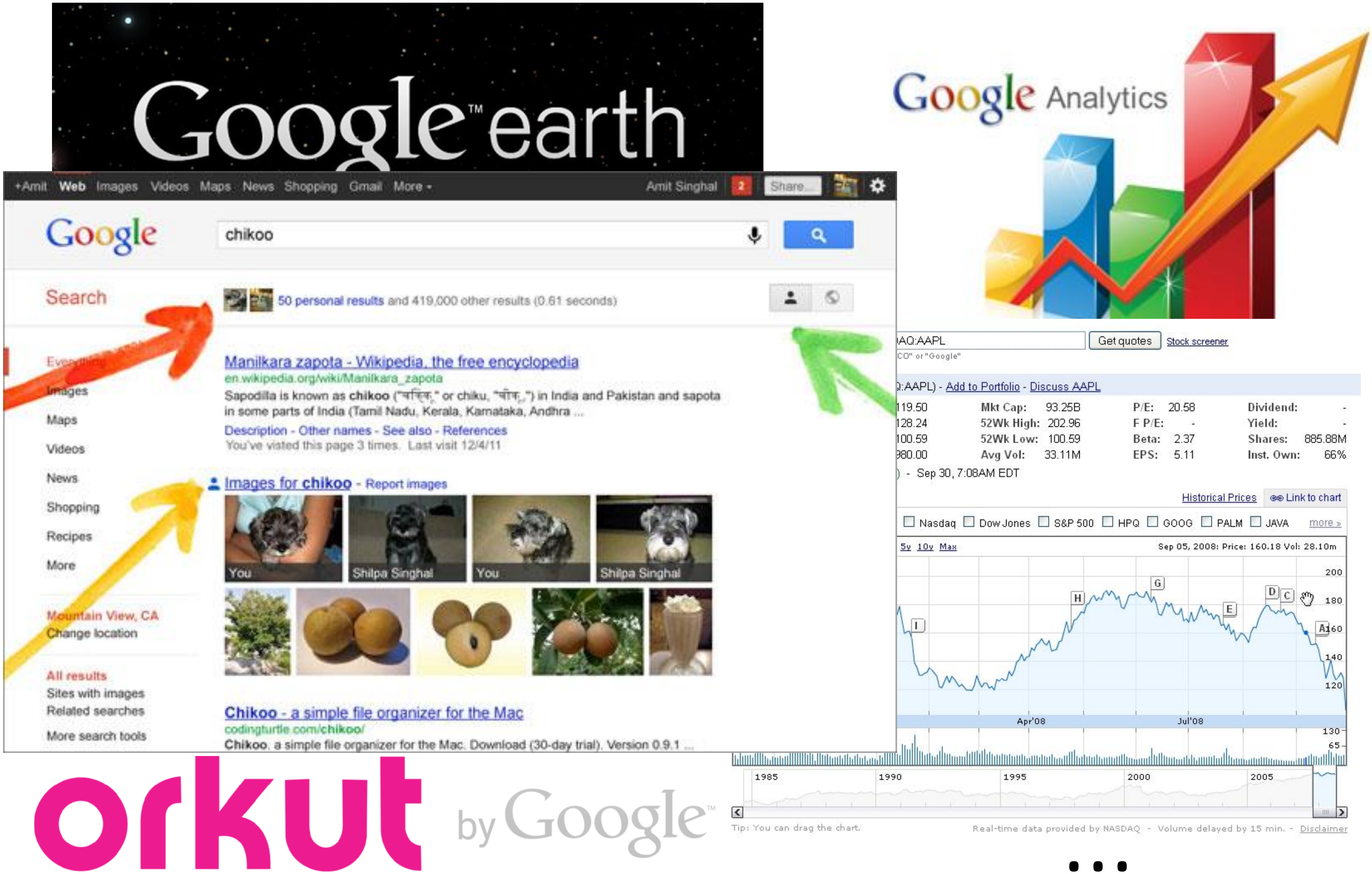
Abstract

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.

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. Sec-

Bigtable used for …



…

Bigtable: in a nutshell

Primary Key value only!

(row, column, time) → value

(Afganistan, pop-value, t_4) → 31108077

Primary Key	capital		continent		pop-value		pop-year	
Afghanistan	t_1	Kabul	t_1	Asia	t_1	31143292	t_1	2009
					t_2	31120978		
					t_4	31108077	t_4	2011
Albania	t_1	Tirana	t_1	Europe	t_1	2912380	t_1	2010
					t_3	3011405	t_3	2013
…	…		…			…		…

Bigtable: Sorted Keys

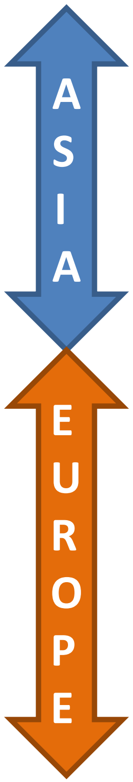
Primary Key	capital		pop-value		pop-year	
Asia:Afghanistan	t_1	Kabul	t_1	31143292	t_1	2009
			t_2	31120978		
			t_4	31108077	t_4	2011
Asia:Azerbaijan	…	…	…	…	…	…
…	…	…	…	…	…	…
Europe:Albania	t_1	Tirana	t_1	2912380	t_1	2010
			t_3	3011405	t_3	2013
Europe:Andorra	…	…	…	…	…	…
…	…	…	…	…	…	…

SORTED

Benefits of sorted vs. hashed keys?

Range queries and …

Bigtable: Tablets



Primary Key	capital		pop-value		pop-year	
Asia:Afghanistan	t_1	Kabul	t_1	31143292	t_1	2009
			t_2	31120978		
			t_4	31108077	t_4	2011
Asia:Azerbaijan	…	…	…	…	…	…
…	…	…	…	…	…	…
Europe:Albania	t_1	Tirana	t_1	2912380	t_1	2010
			t_3	3011405	t_3	2013
Europe:Andorra	…	…	…	…	…	…
…	…	…	…	…	…	…

Benefits of sorted vs. hashed keys?

Range queries and …

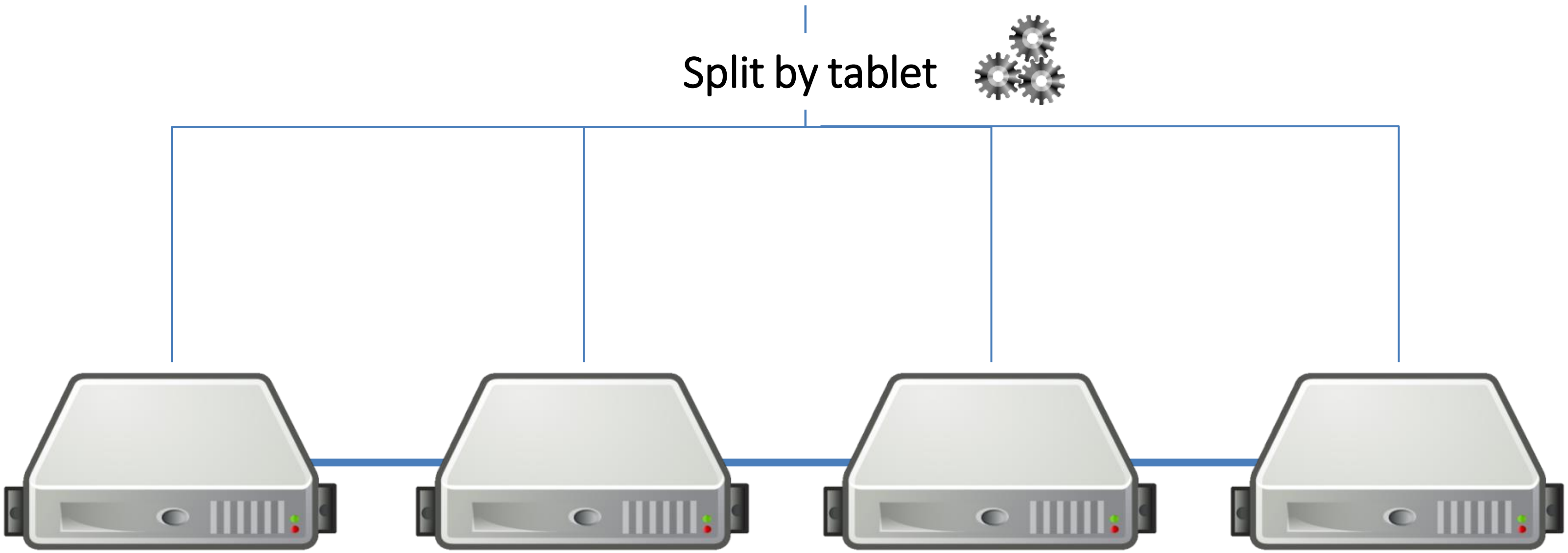
… locality of processing

A real-world example of locality/sorting

Primary Key	language		title		links	
com.imdb	t_1	en	t_1	IMDb Home	t_1	…
			t_2	IMDB - Movies		
			t_4	IMDb	t_4	…
com.imdb/title/tt2724064/	t_1	en	t_2	Sharknado	t_2	…
com.imdb/title/tt3062074/	t_1	en	t_2	Sharknado II	t_2	
…	…	…	…	…	…	…
org.wikipedia	t_1	multi	t_1	Wikipedia	t_1	…
			t_3	Wikipedia Home	t_3	…
org.wikipedia.ace	t_1	ace	t_1	Wikipèdia bahsa Acèh	…	…
…	…	…	…	…	…	…

Bigtable: Distribution

Asia:Afghanistan	t_1	Kabul	t_1	31143292	t_1	2009
			t_2	31120978		
			t_4	31108077	t_4	2011
Asia:Azerbaijan	...	...	...	...	...	...
...	...	...	...	...	...	...
Europe:Albania	t_1	Tirana	t_1	2912380	t_1	2010
			t_3	3011405	t_3	2013
Europe:Andorra	...	...	...	...	...	...
...	...	...	...	...	...	...



Horizontal range partitioning

Bigtable: Column Families

Primary Key	pol:capital		demo:pop-value		demo:pop-year	
Asia:Afghanistan	t_1	Kabul	t_1	31143292	t_1	2009
			t_2	31120978		
			t_4	31108077	t_4	2011
Asia:Azerbaijan	…	…	…	…	…	…
…	…	…	…	…	…	…
Europe:Albania	t_1	Tirana	t_1	2912380	t_1	2010
			t_3	3011405	t_3	2013
Europe:Andorra	…	…	…	…	…	…
…	…	…	…	…	…	…

- Group logically similar columns together
  - Accessed efficiently together
  - Access-control and storage: column family level
  - If of same type, can be compressed

Read More …

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.

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

Tabular Store: Apache HBase

Tabular Store: Cassandra

Questions?