CC5212-1

Procesamiento Masivo de Datos

Otoño 2018

Lecture 8

NoSQL: Overview

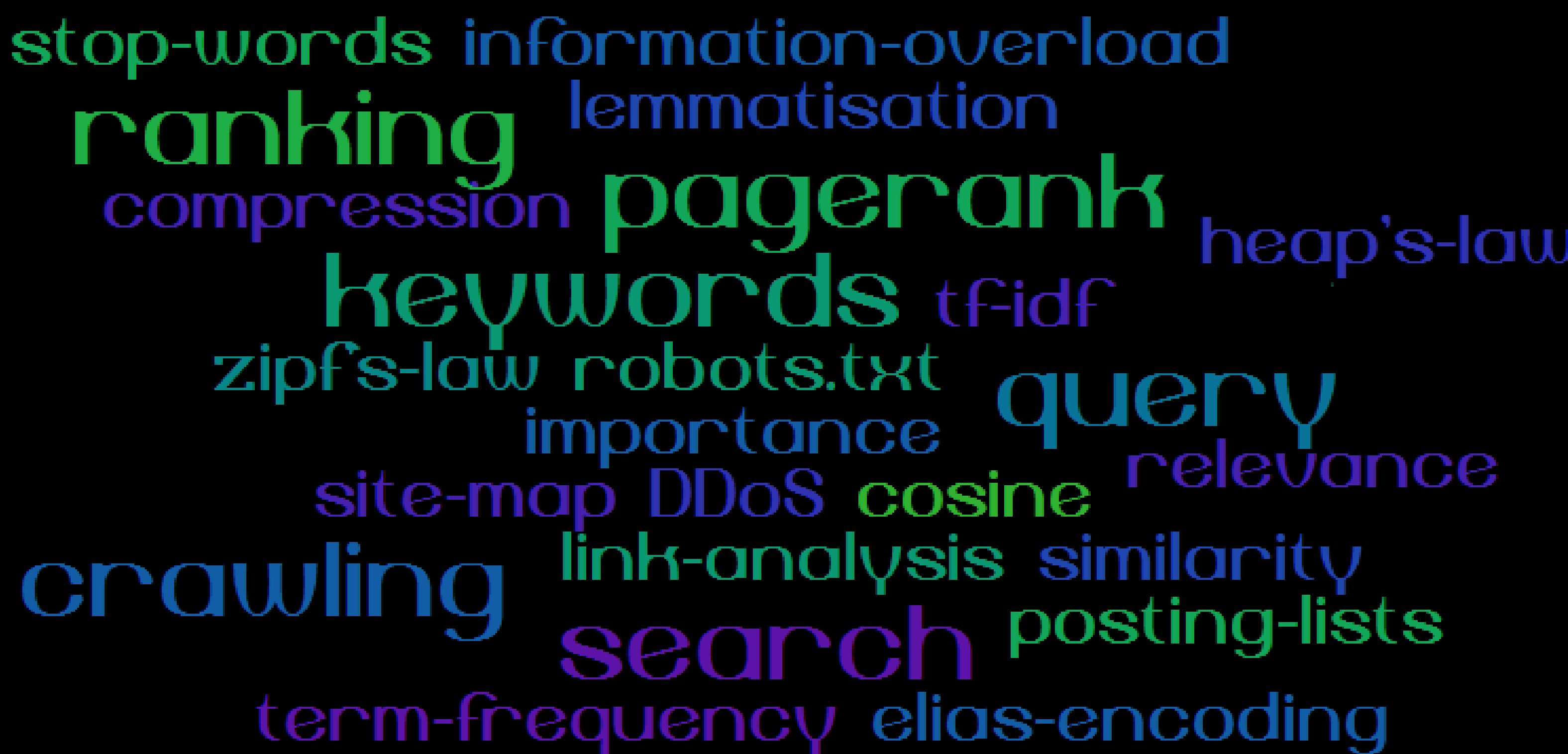
Aidan Hogan
aidhog@gmail.com

Hadoop/MapReduce/Pig/Spark:

Processing Un/Structured Information

Information Retrieval:

Storing Unstructured Information



Storing Structured Information??

?

Big Data:

Storing Structured Information

Relational Databases

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In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines, some of which may be unified by a common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.

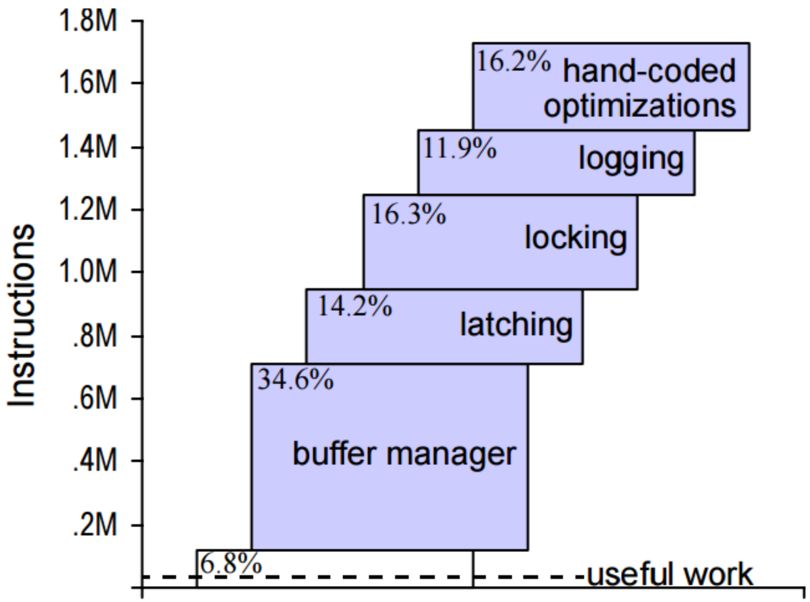
The rest of the paper is structured as follows. In Section 2, we briefly indicate why the single code-line strategy has failed already by citing some of the key characteristics of the data warehouse market. In Section


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

RDBMS: Performance Overheads

- Structured Query Language (SQL):
  - Declarative Language
  - Lots of Rich Features
  - **Difficult to Optimise!**
- Atomicity, Consistency, Isolation, Durability (ACID):
  - Makes sure your database stays correct
    - Even if there's a lot of traffic!
  - Transactions incur a lot of overhead
    - Multi-phase locks, multi-versioning, write ahead logging
- Distribution not straightforward

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

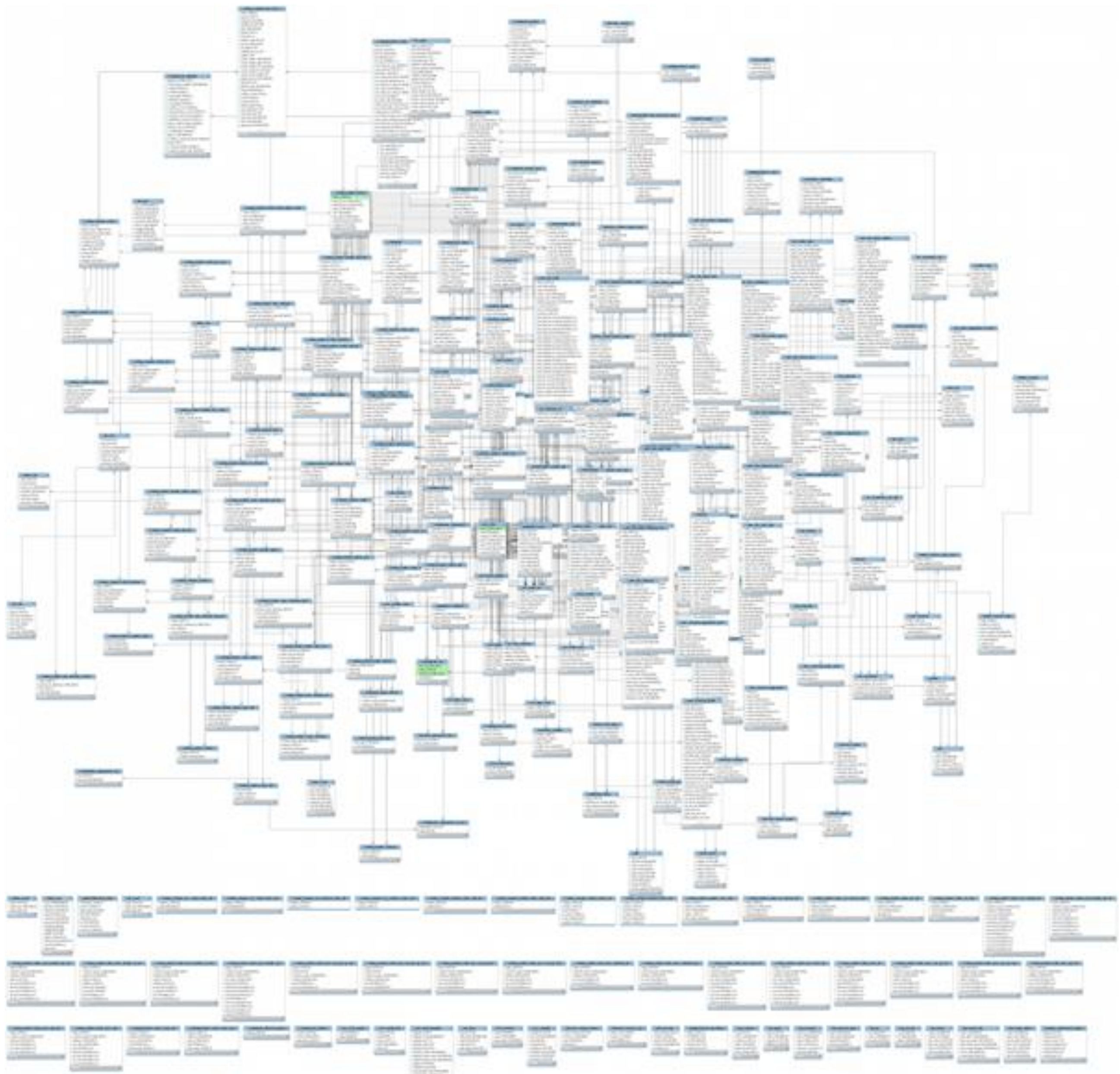
ABSTRACT

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

RDBMS: Complexity



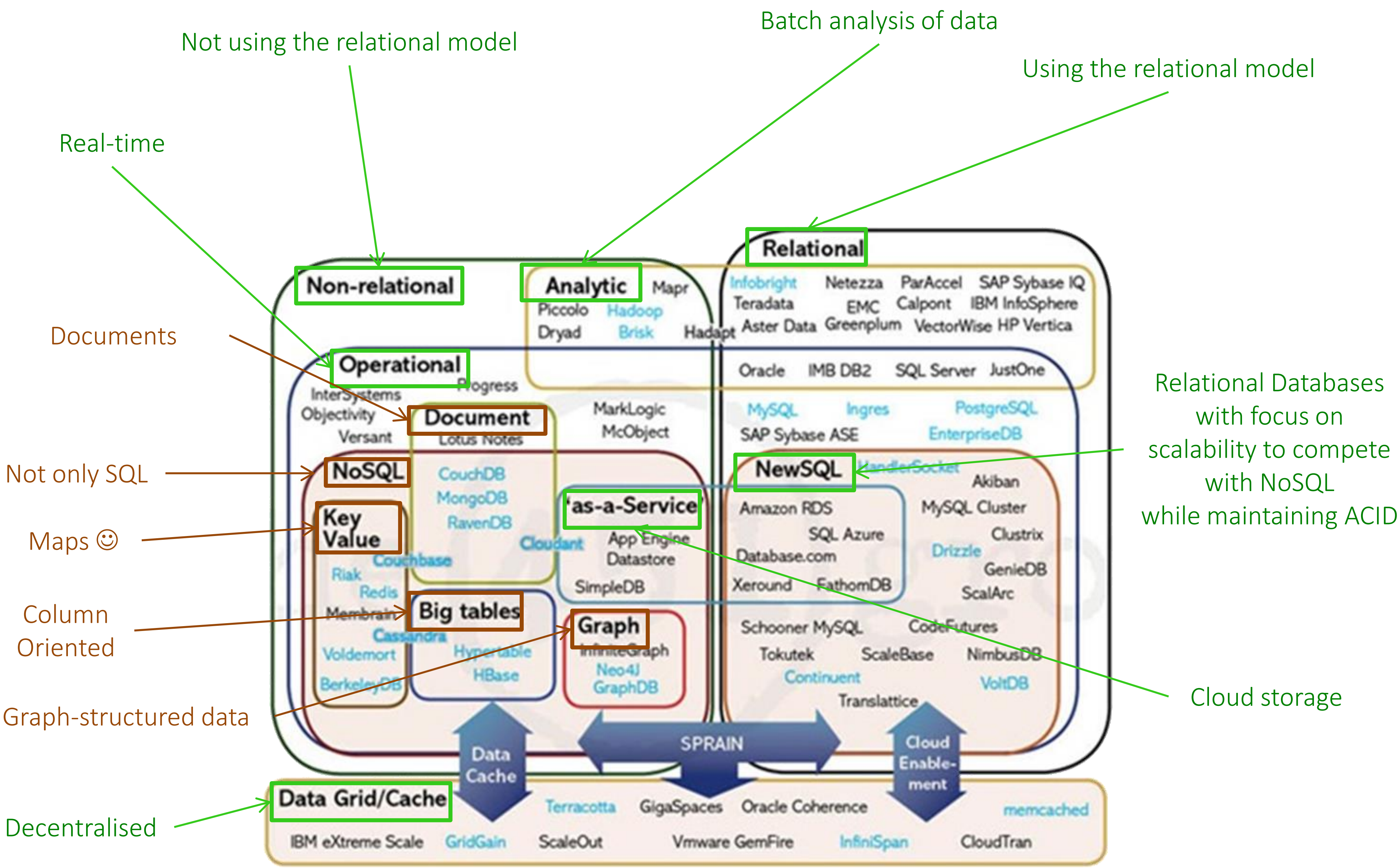
Alternatives to Relational Databases for Big Data?

NoSQL

Anybody know anything about NoSQL?

Not **O**nly SQL

Many types of NoSQL stores

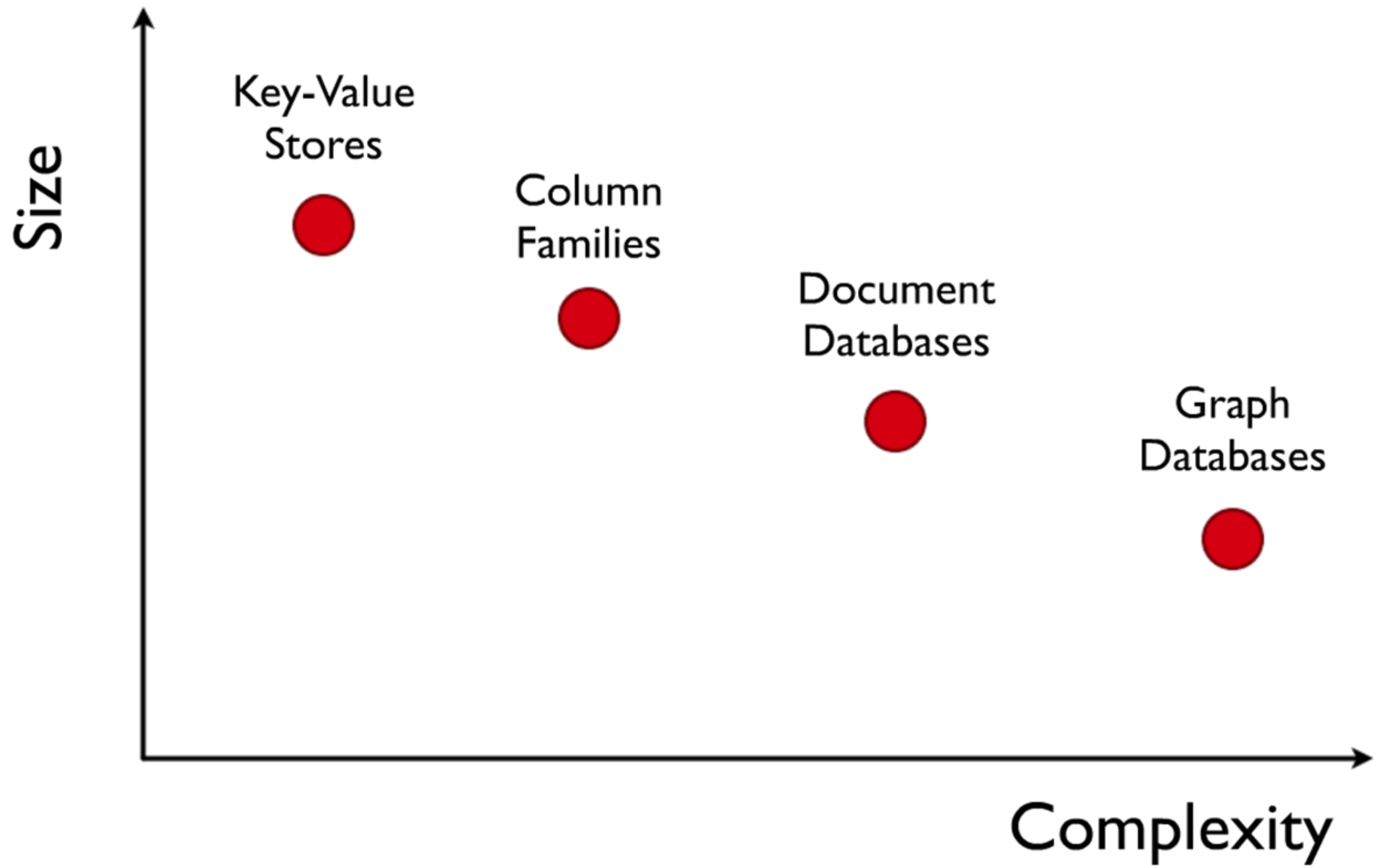


342 systems in ranking, May 2018

Rank May 2018	Rank Apr 2018	Rank May 2017	DBMS	Database Model	Score May 2018	Score Apr 2018	Score May 2017
1.	1.	1.	Oracle	Relational DBMS	1290.42	+0.63	-63.90
2.	2.	2.	MySQL	Relational DBMS	1223.34	-3.06	-116.69
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1085.84	-9.67	-127.96
4.	4.	4.	PostgreSQL	Relational DBMS	400.90	+5.43	+34.99
5.	5.	5.	MongoDB	Document store	342.11	+0.70	+10.53
6.	6.	6.	DB2	Relational DBMS	185.61	-3.34	-3.23
7.	↑ 9.	↑ 9.	Redis	Key-value store	135.35	+5.24	+17.90
8.	↓ 7.	↓ 7.	Microsoft Access	Relational DBMS	133.11	+0.89	+3.24
9.	↓ 8.	↑ 11.	Elasticsearch	Search engine	130.44	-0.92	+21.62
10.	10.	↓ 8.	Cassandra	Wide column store	117.83	-1.26	-5.28
11.	11.	↓ 10.	SQLite	Relational DBMS	115.45	-0.53	-0.61
12.	12.	12.	Teradata	Relational DBMS	74.41	+0.74	-1.91
13.	13.	↑ 16.	Splunk	Search engine	65.09	+0.04	+8.40
14.	14.	↑ 18.	MariaDB	Relational DBMS	64.99	+0.44	+14.01
15.	15.	↓ 14.	Solr	Search engine	61.51	-1.70	-2.26
16.	16.	↓ 13.	SAP Adaptive Server	Relational DBMS	61.51	-0.12	-6.24
17.	17.	↓ 15.	HBase	Wide column store	59.95	+0.26	+0.44
18.	18.	↑ 20.	Hive	Relational DBMS	56.97	-0.43	+13.49
19.	19.	↓ 17.	FileMaker	Relational DBMS	54.67	-0.33	-1.81
20.	20.	↓ 19.	SAP HANA	Relational DBMS	48.37	-0.52	-0.68
21.	21.	↑ 22.	Amazon DynamoDB	Multi-model	44.19	+1.05	+10.99
22.	22.	↓ 21.	Neo4j	Graph DBMS	40.58	-0.32	+4.44
23.	23.	↑ 24.	Memcached	Key-value store	33.56	-0.23	+4.15
24.	24.	↓ 23.	Couchbase	Document store	32.41	+0.07	+0.16
25.	25.	25.	Informix	Relational DBMS	25.79	-0.82	-2.44
26.	26.	↑ 27.	Microsoft Azure SQL Database	Relational DBMS	25.21	+0.74	+3.66
27.	27.	↑ 28.	Vertica	Relational DBMS	21.10	+0.39	+0.41
28.	28.	↓ 26.	CouchDB	Document store	19.42	-0.43	-2.98
29.	29.	↑ 30.	Firebird	Relational DBMS	18.99	+0.35	+0.27
30.	30.	↑ 52.	Microsoft Azure Cosmos DB	Multi-model	17.54	+0.35	+12.70

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

NoSQL


Key-Value
Stores
Column
Families
Document
Databases
Graph
Databases
Size
Complexity

NoSQL: Not only SQL

- Distributed!
  - Sharding: splitting data over servers "horizontally"
  - Replication
  - Different guarantees: typically not ACID

- Often simpler languages than SQL
  - Simpler ad hoc APIs
  - More work for the application

- Different flavours (for different scenarios)
  - Different CAP emphasis
  - Different scalability profiles
  - Different query functionality
  - Different data models

LIMITATIONS OF DISTRIBUTED COMPUTING: CAP THEOREM

But first … ACID

For traditional (non-distributed) databases …

1. **A**tomicity:
   – Transactions all or nothing: fail cleanly
2. **C**onsistency:
   – Doesn't break constraints/rules
3. **I**solation:
   – Parallel transactions act as if sequential
4. **D**urability
   – System remembers changes

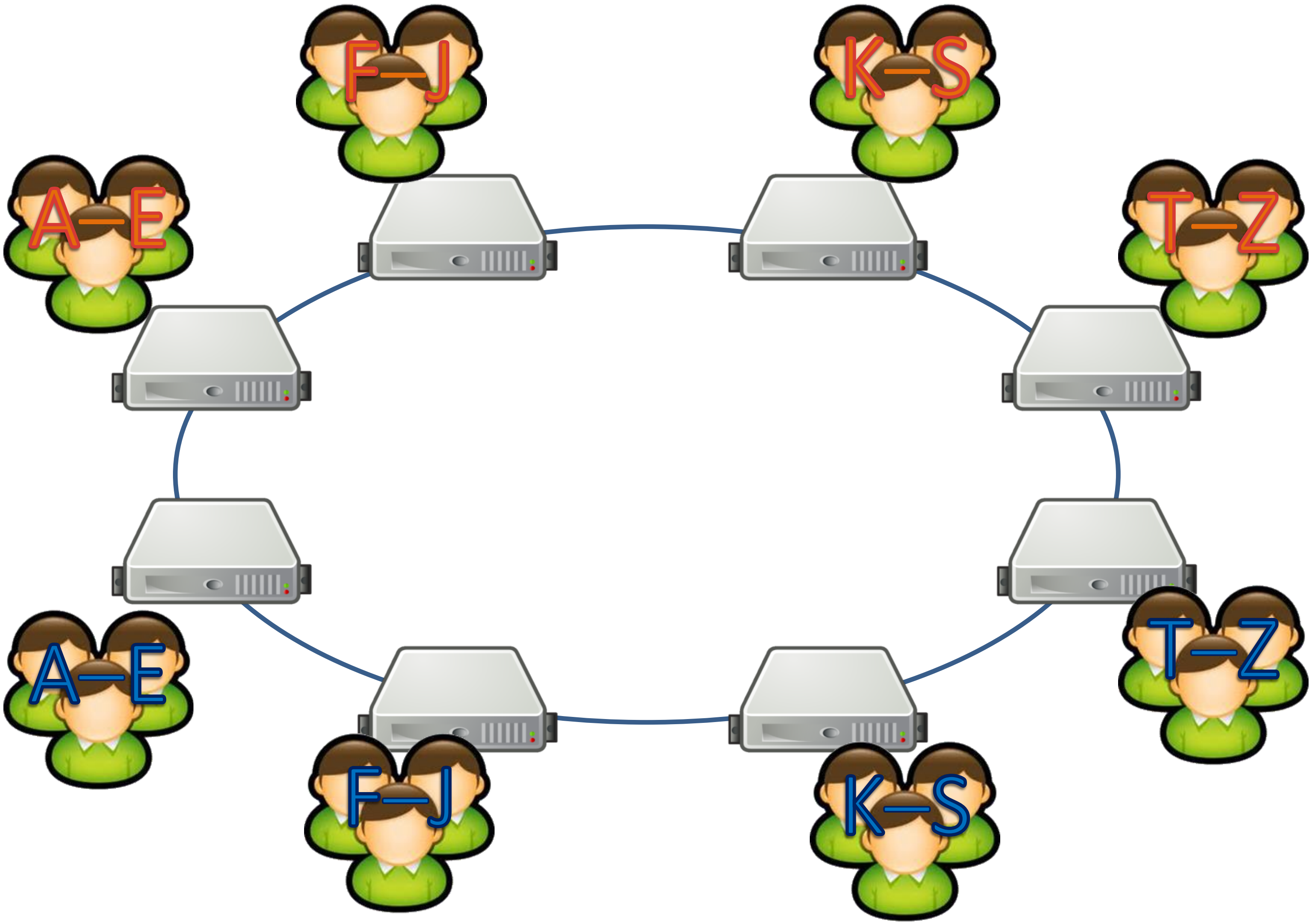
What is CAP?

Three *guarantees* a distributed sys. could make

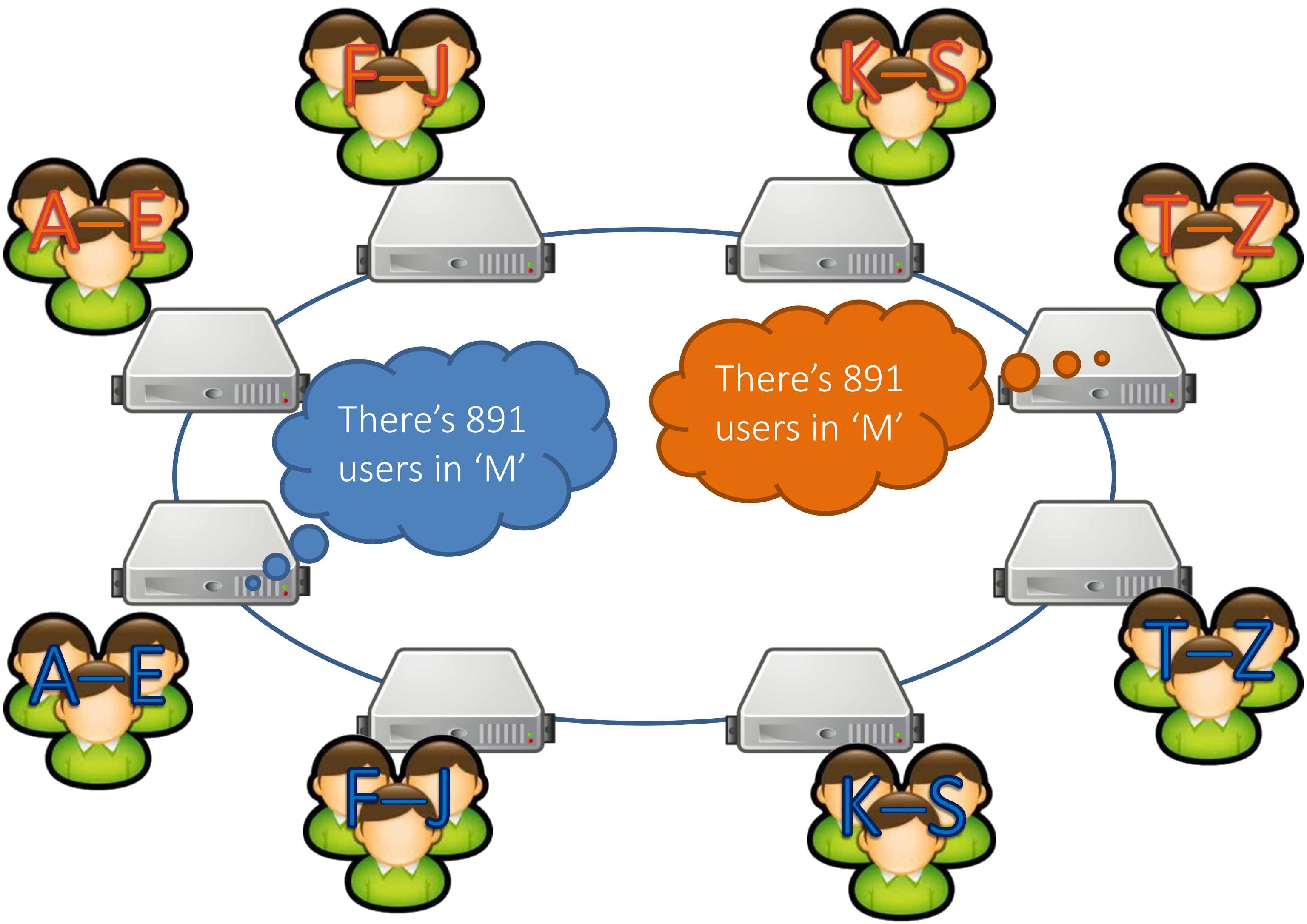
1. Consistency:
   - All nodes have a consistent view of the system
2. Availability:
   - Every read/write is acted upon
3. Partition-tolerance:
   - The system works even if messages are lost

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

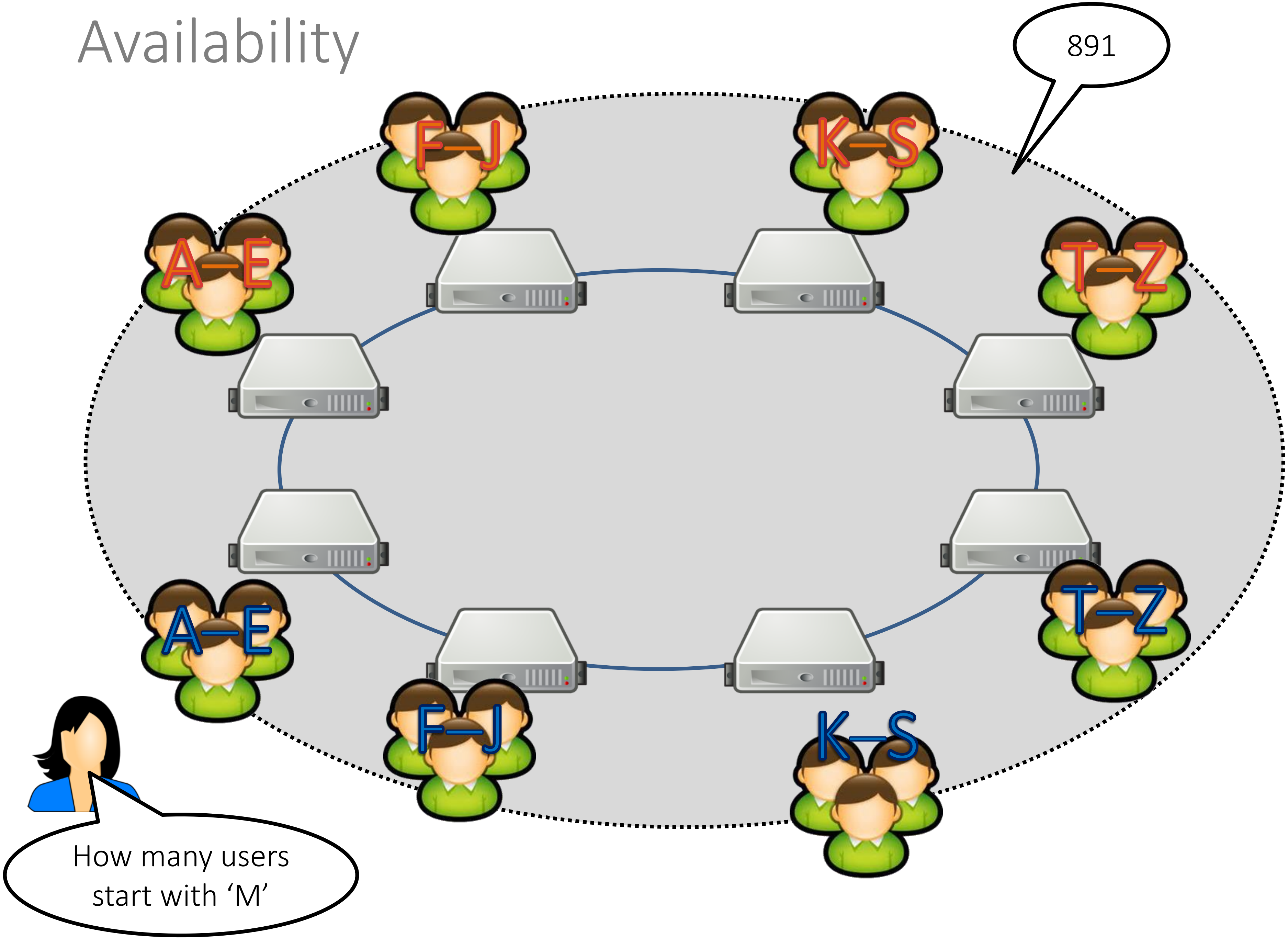
A Distributed System (with Replication)



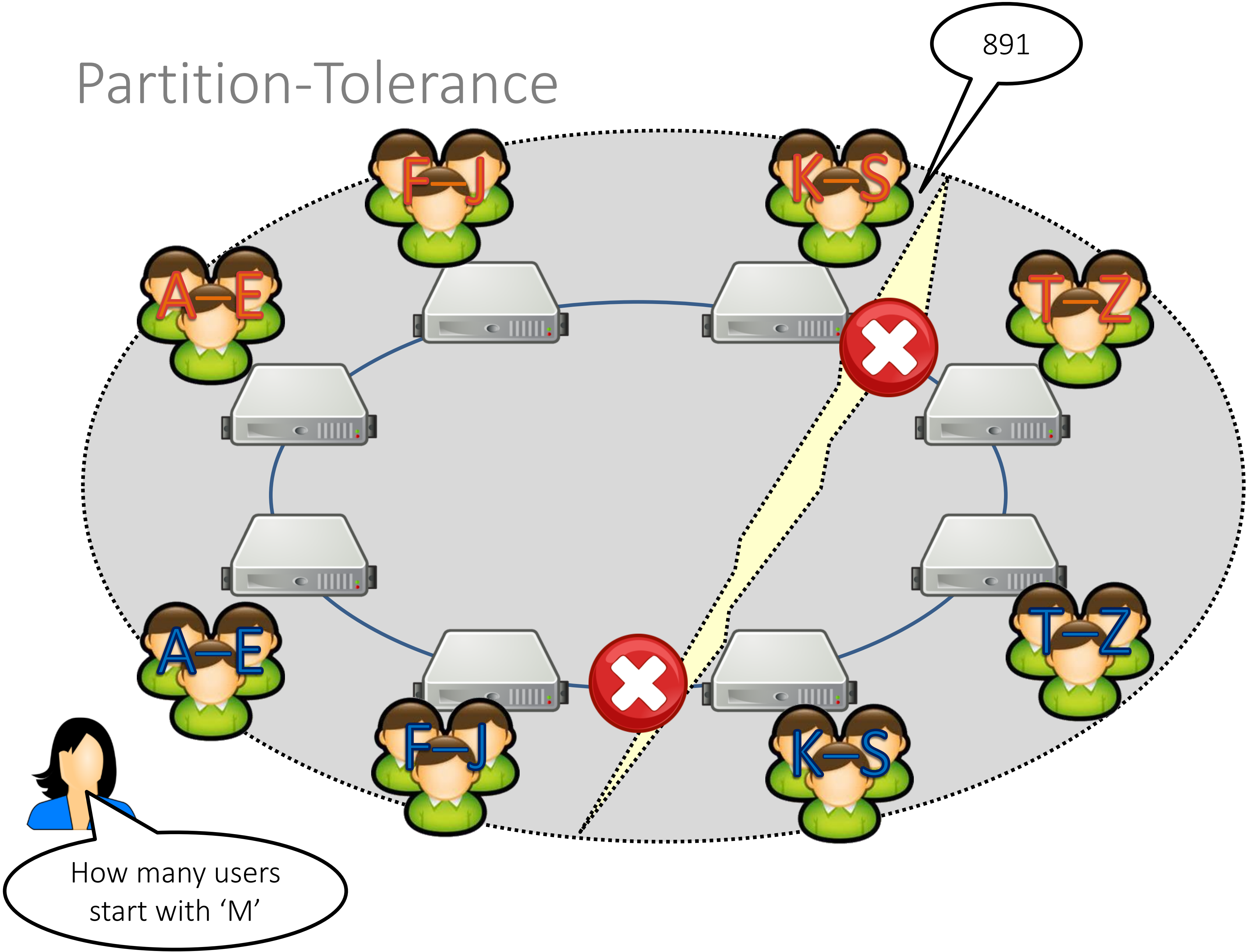
Consistency



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

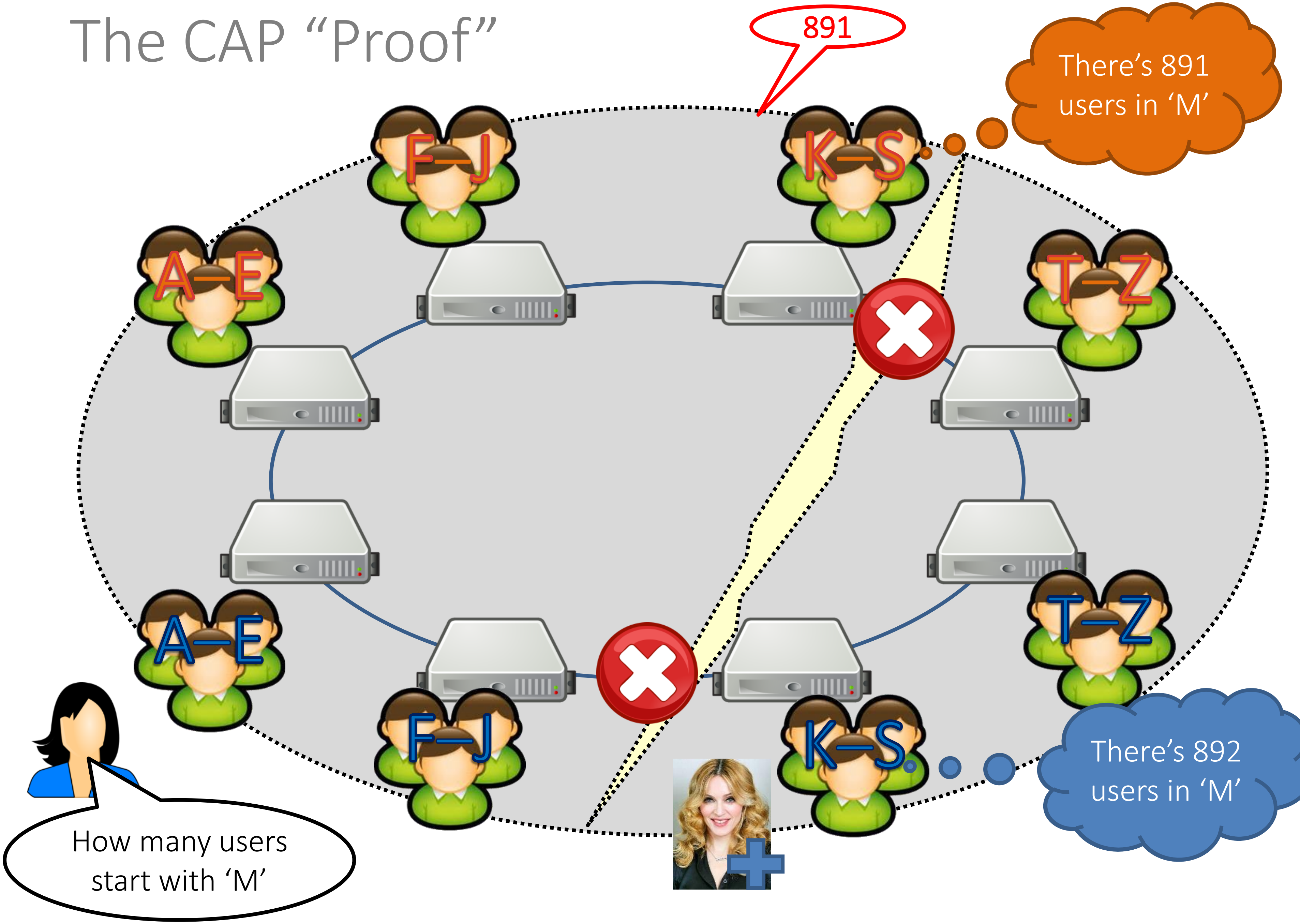
consistency (all nodes have the same up-to-date view),

availability (every read/write is acted upon) and

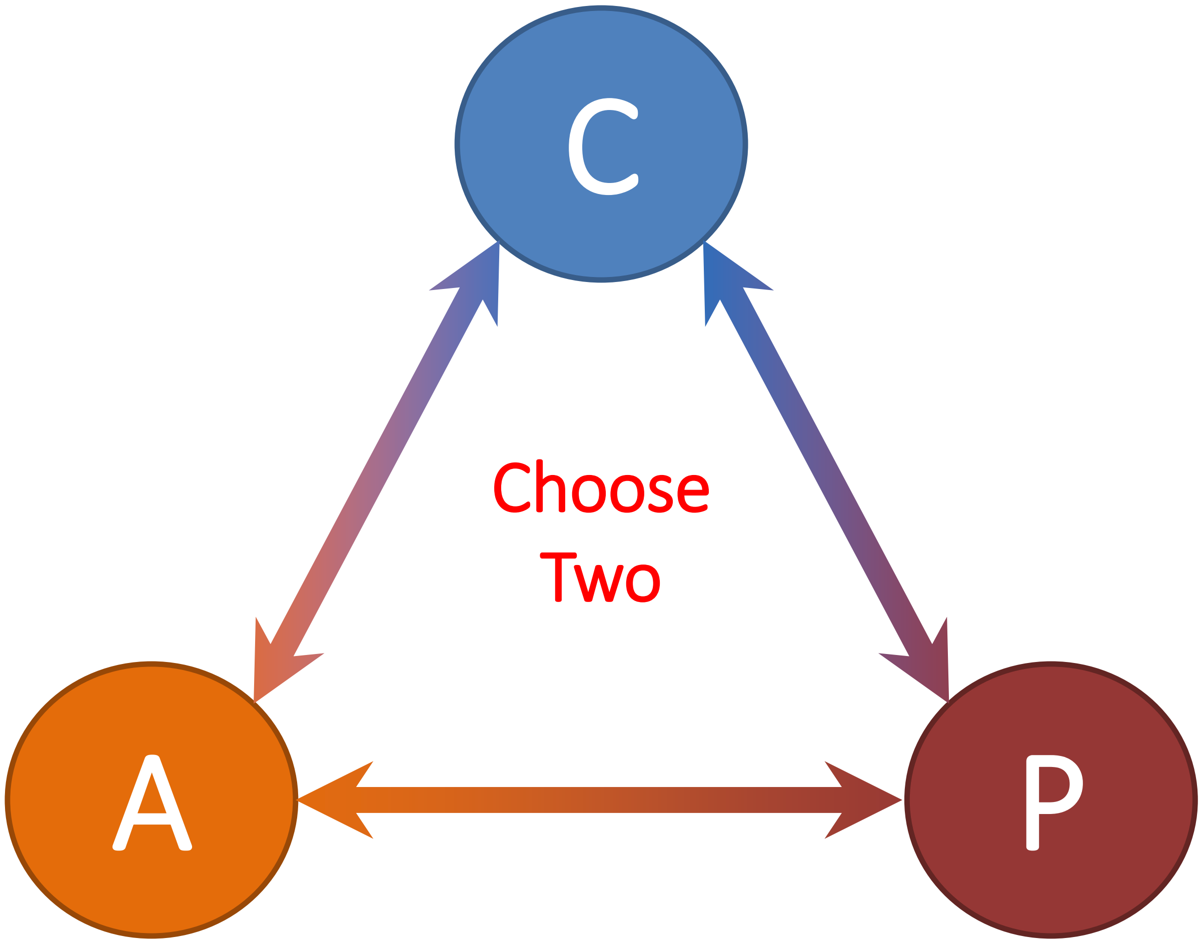
partition-tolerance (the system works if messages are lost)

at the same time!

The CAP "Proof"



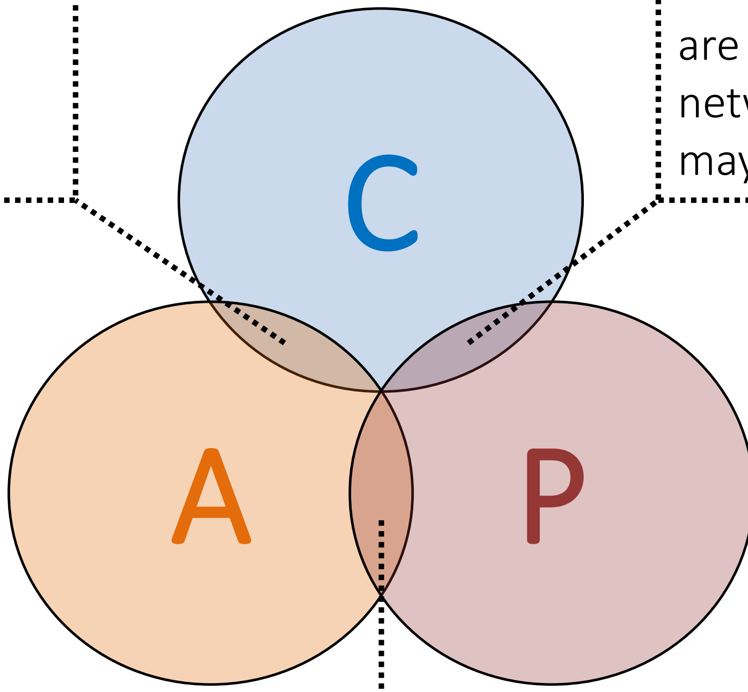
The CAP Triangle



CAP Systems

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

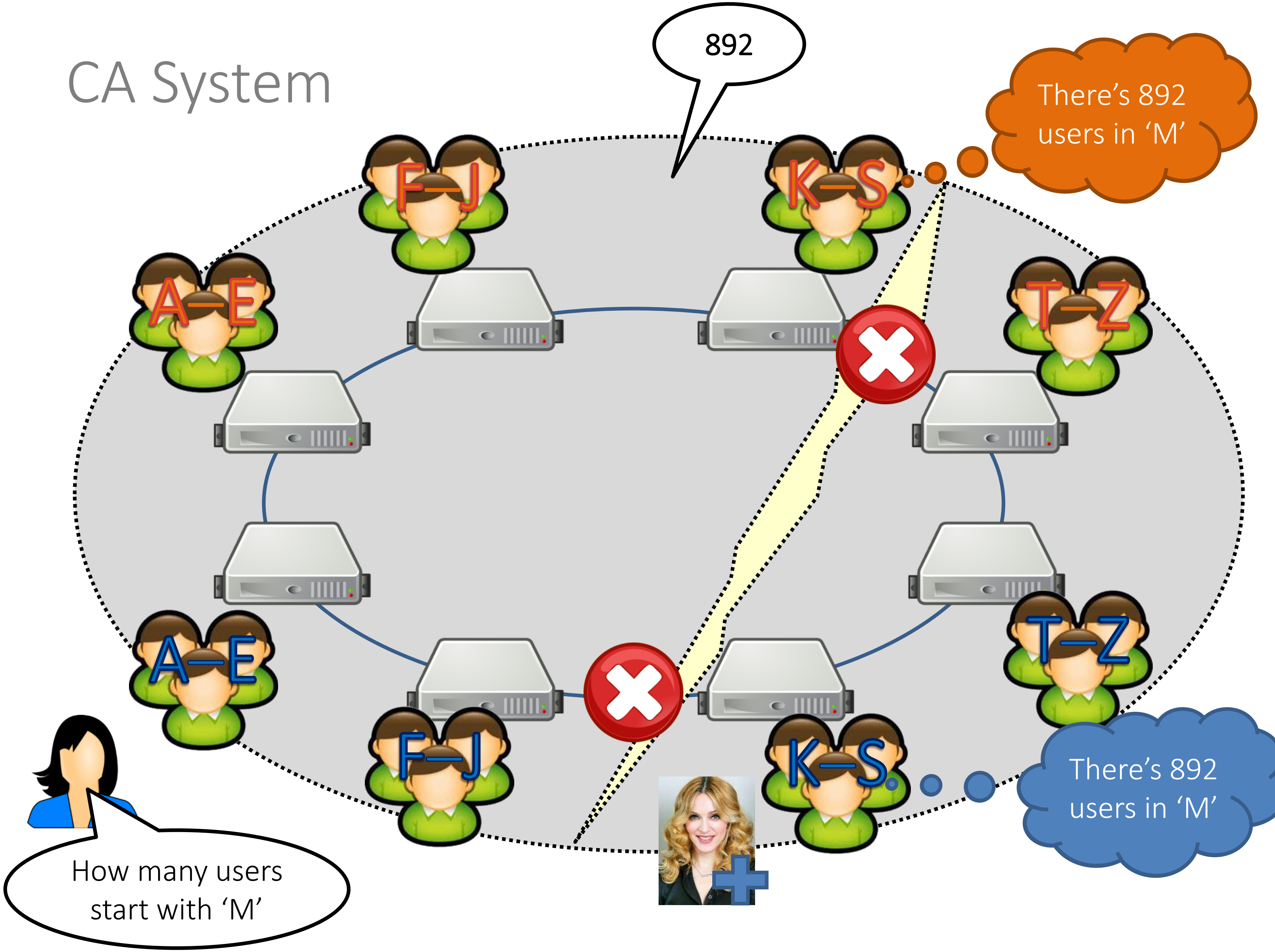
CP: Guarantees responses are correct even if there are network failures, but response may fail (*Weak availability*)



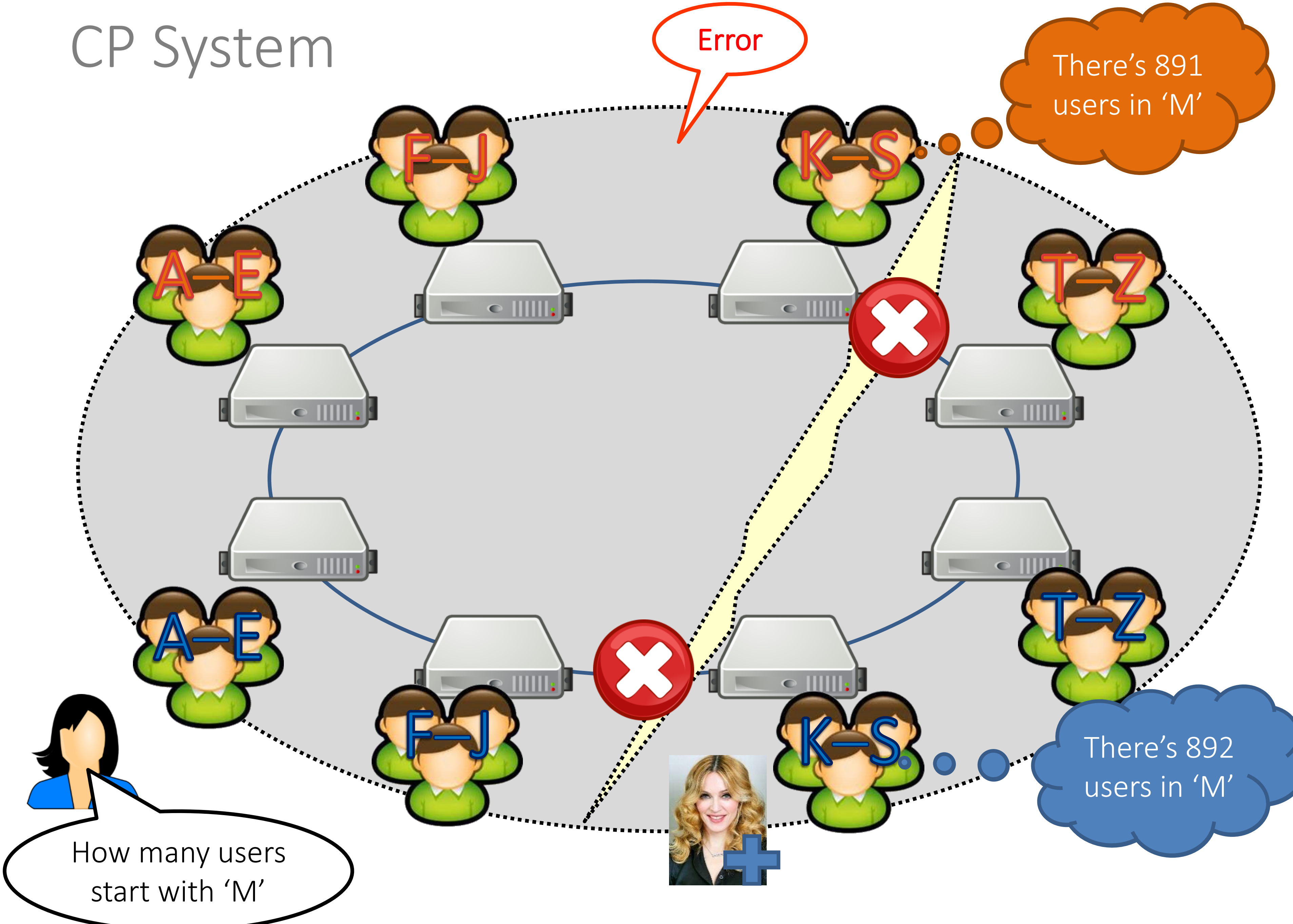
(No intersection)

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

CA System

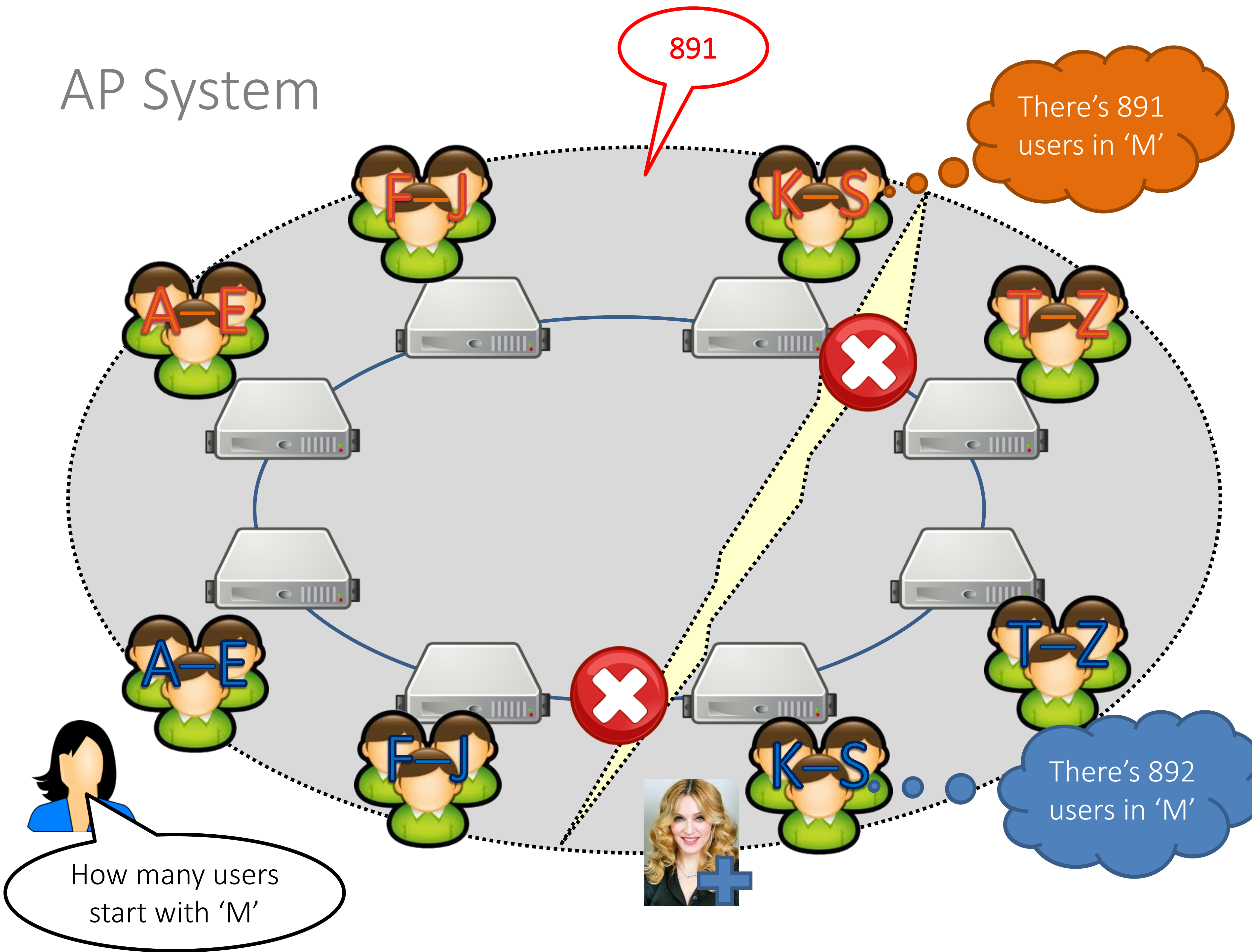

892
There's 892 users in 'M'
F–J
K–S
A–E
T–Z
A–E
F–J
K–S
T–Z
There's 892 users in 'M'
How many users start with 'M'

CP System



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

AP System



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

BASE (AP)

- Basically Available
  - Pretty much always "up"
- Soft State
  - Replicated, cached data
- Eventual Consistency
  - Stale data tolerated, for a while

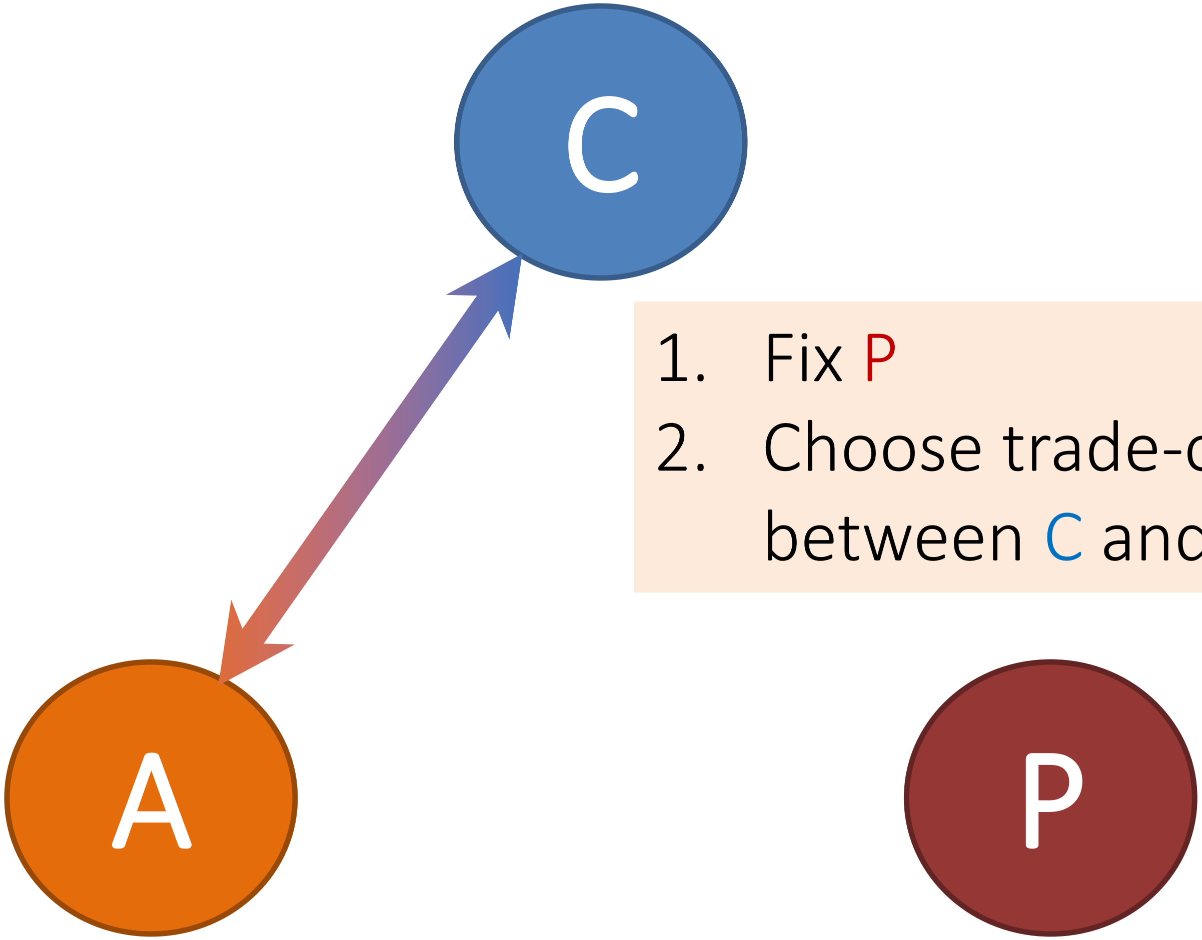
In what way does Twitter act as a BASE (AP) system?

High-fanout creates a “partition”

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



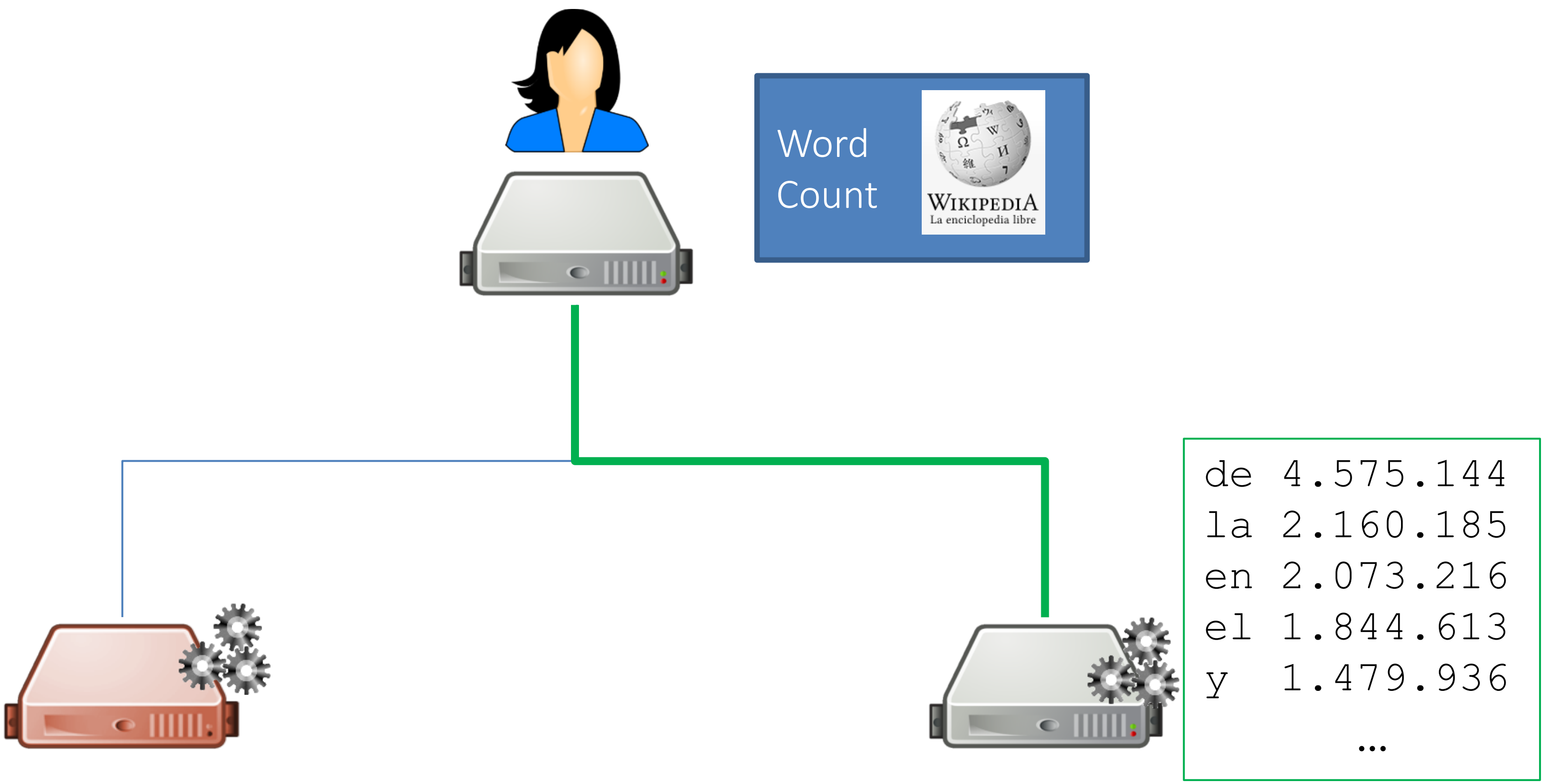
1. Fix P
2. Choose trade-off point between C and A

Partition Tolerance

Faults

Fail–Stop Fault

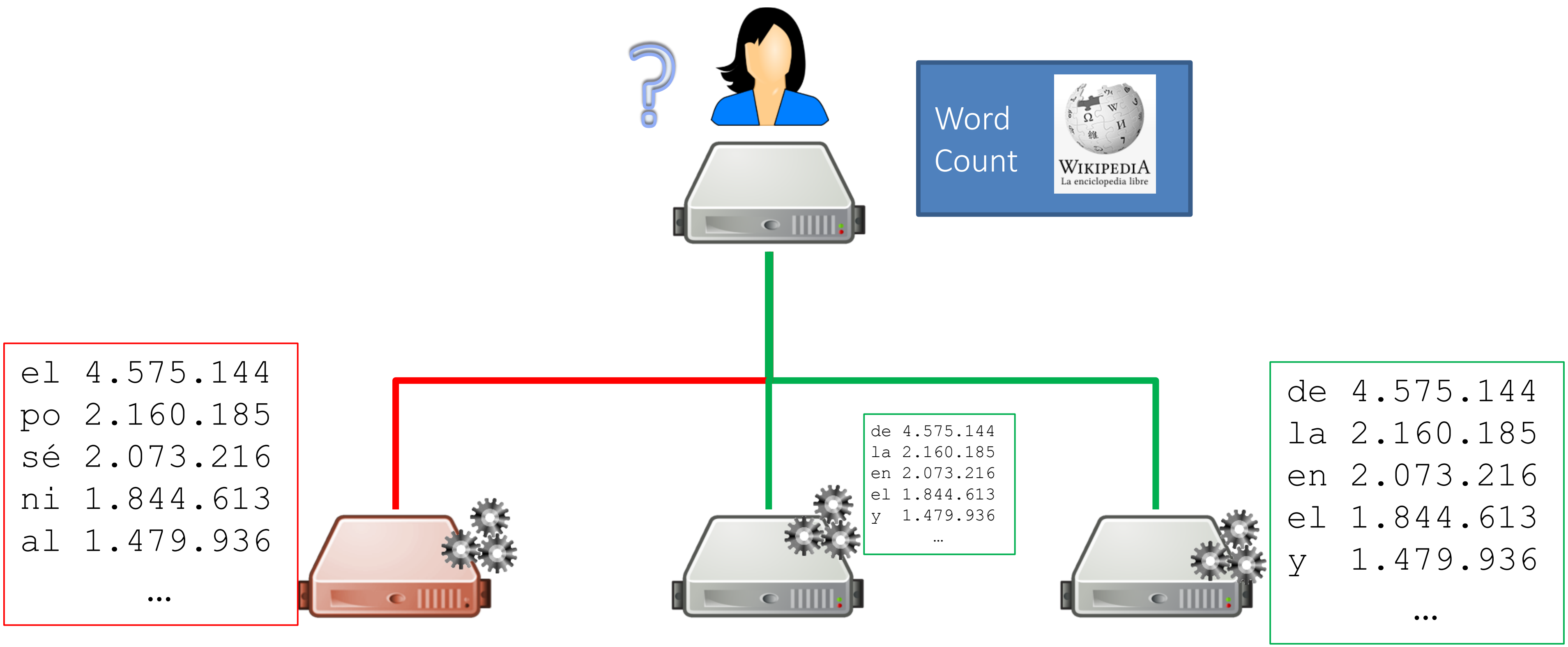
- A machine fails to respond or times-out
  - often hardware or load
  - need at least *f* + 1 <u>replicated</u> machines
    - *f* = number of fail-stop failures



Byzantine Fault

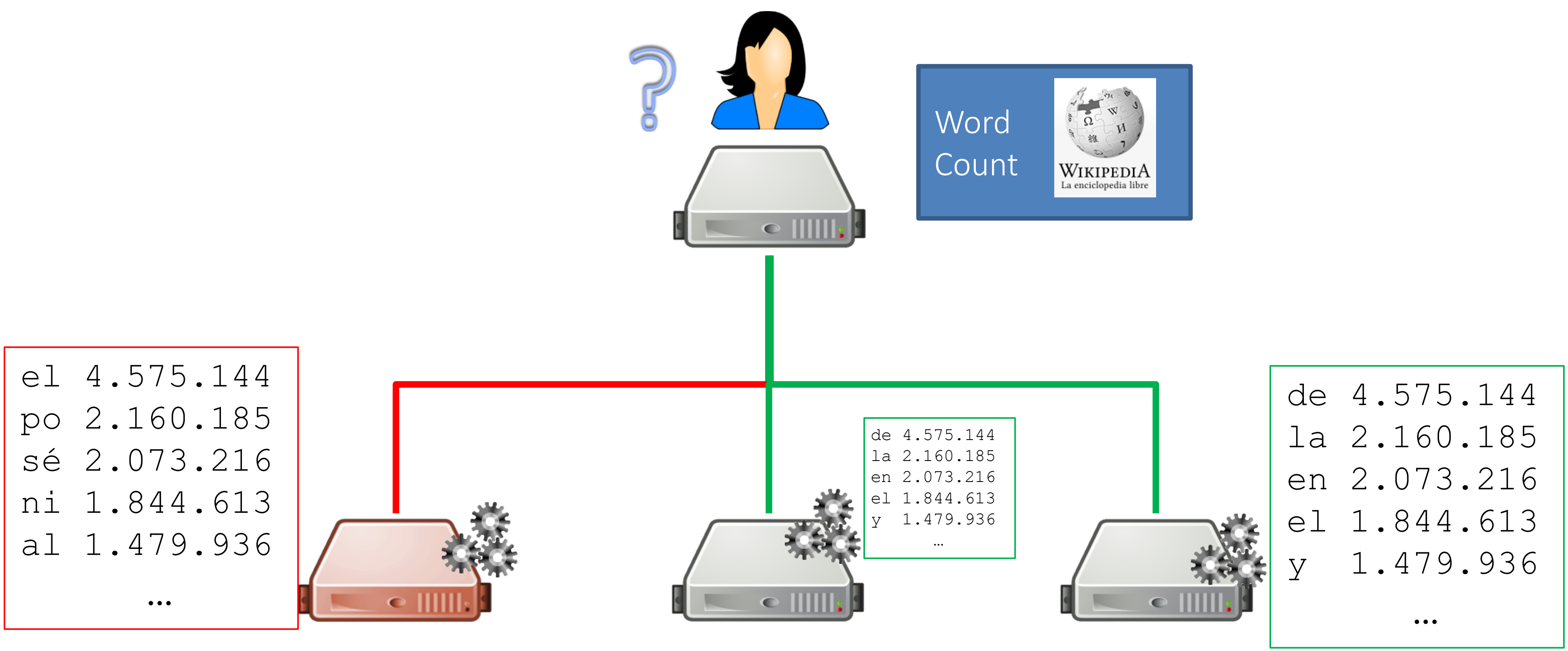
- A machine responds incorrectly/maliciously

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



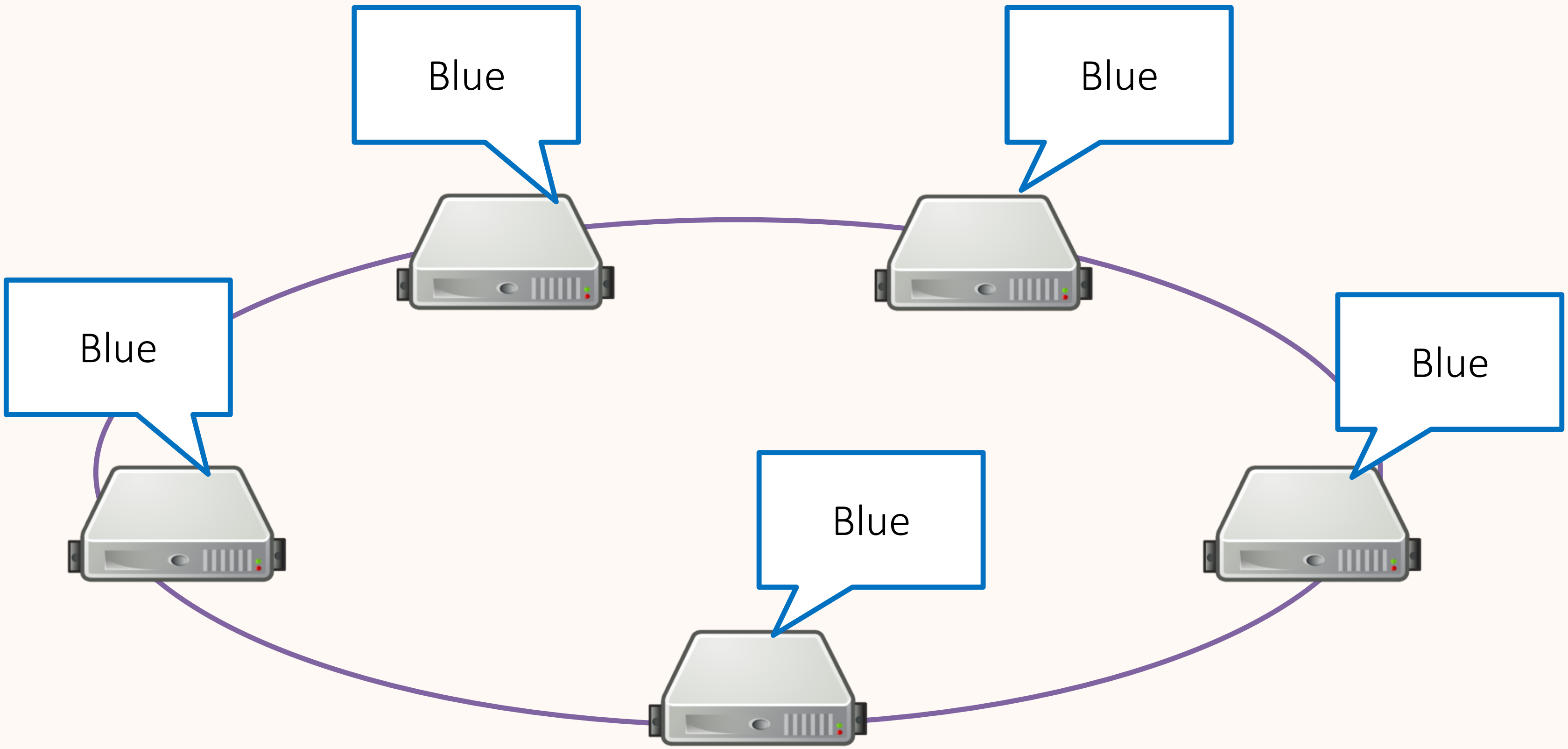
DISTRIBUTED CONSENSUS

Distributed Consensus

Colour of the dress?

Distributed Consensus

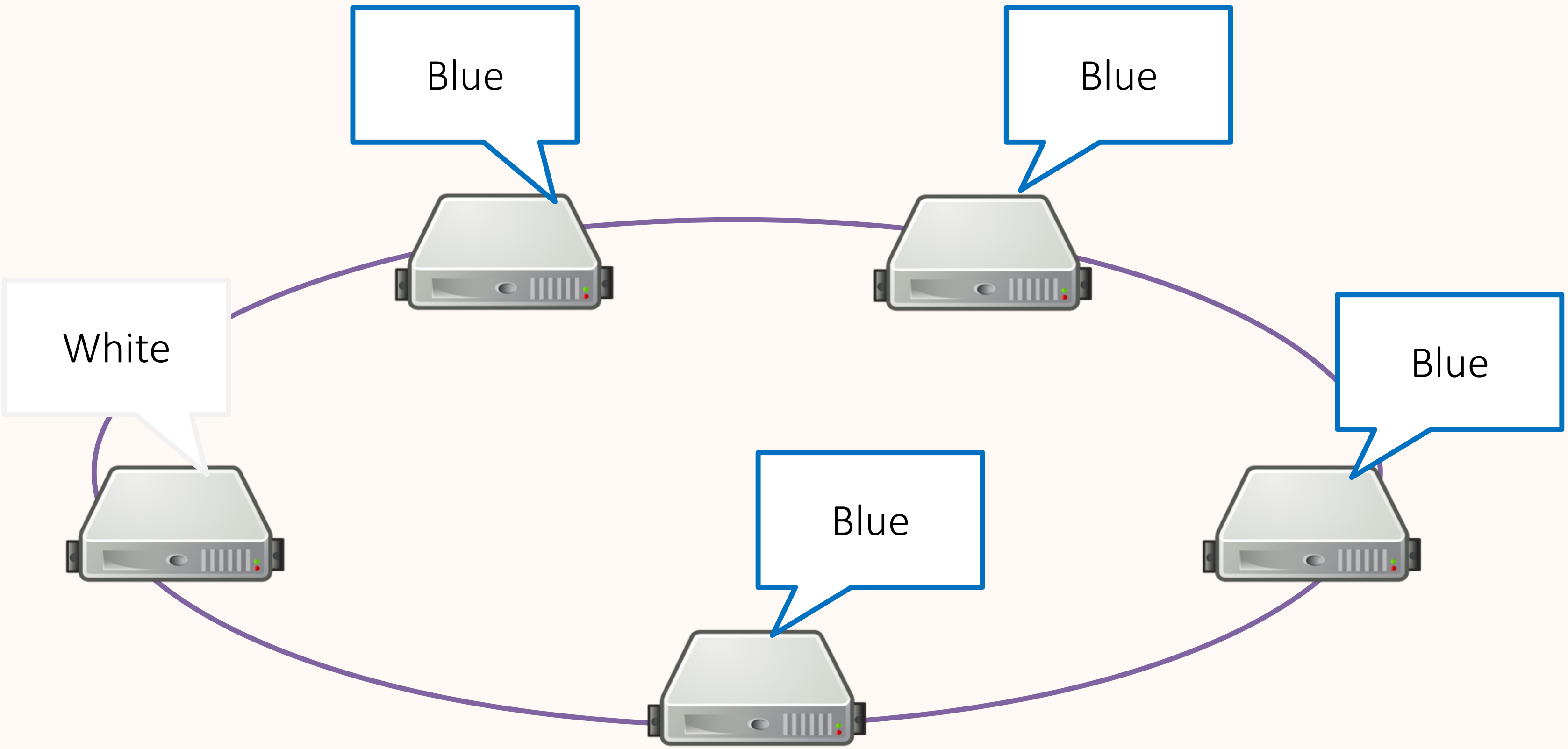
Strong consensus: All nodes need to agree



Consensus.

Distributed Consensus

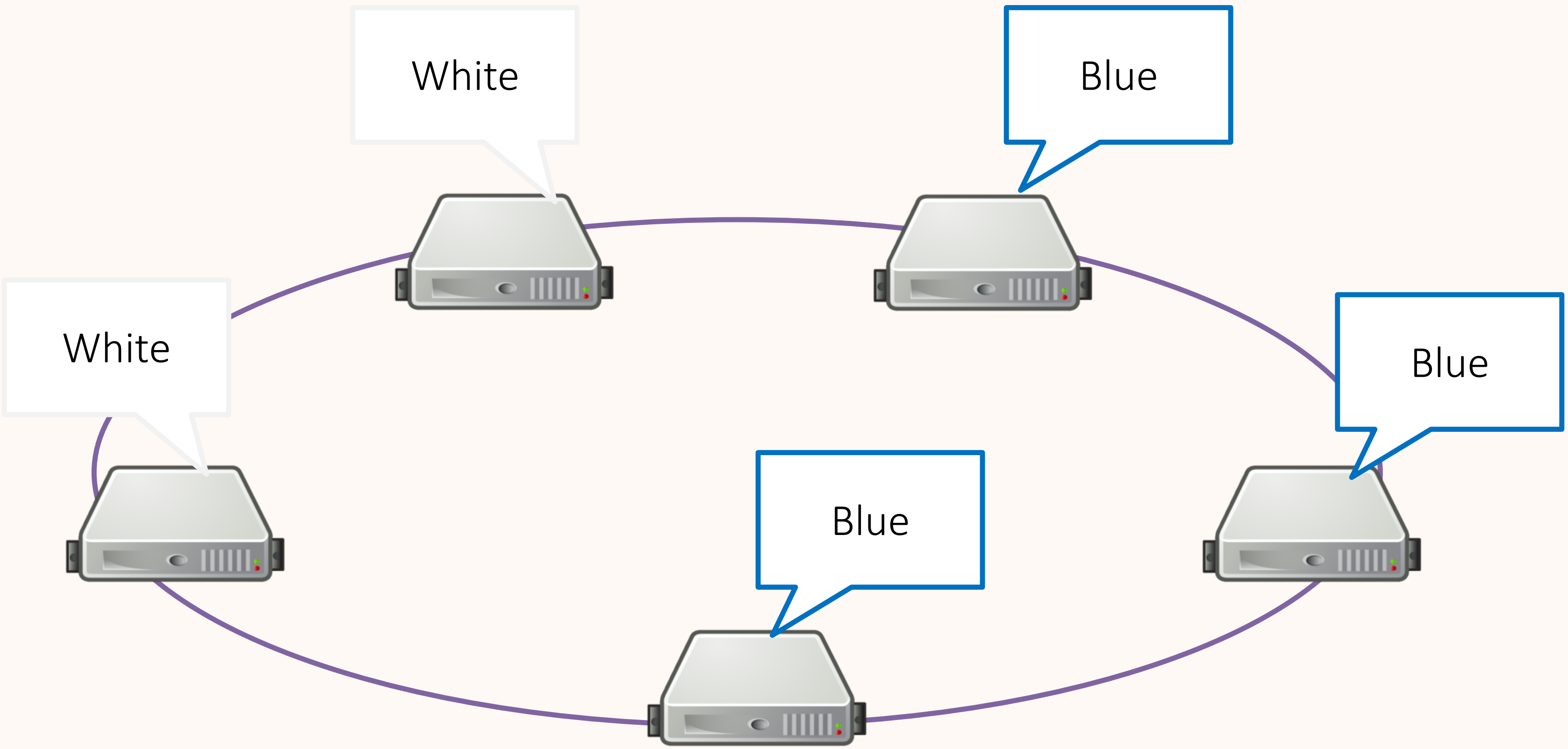
Strong consensus: All nodes need to agree



No consensus.

Distributed Consensus

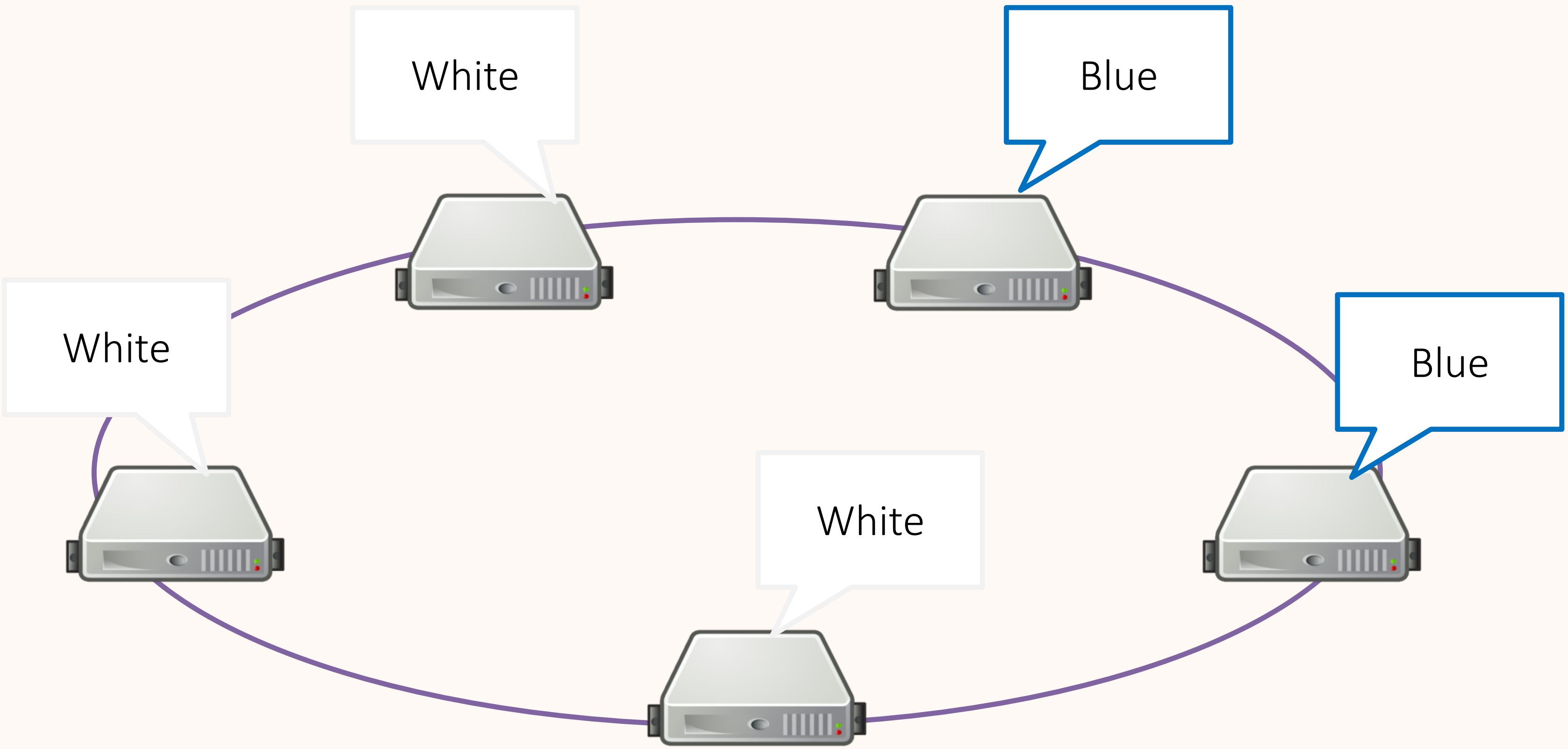
Majority consensus: A majority of nodes need to agree



Consensus.

Distributed Consensus

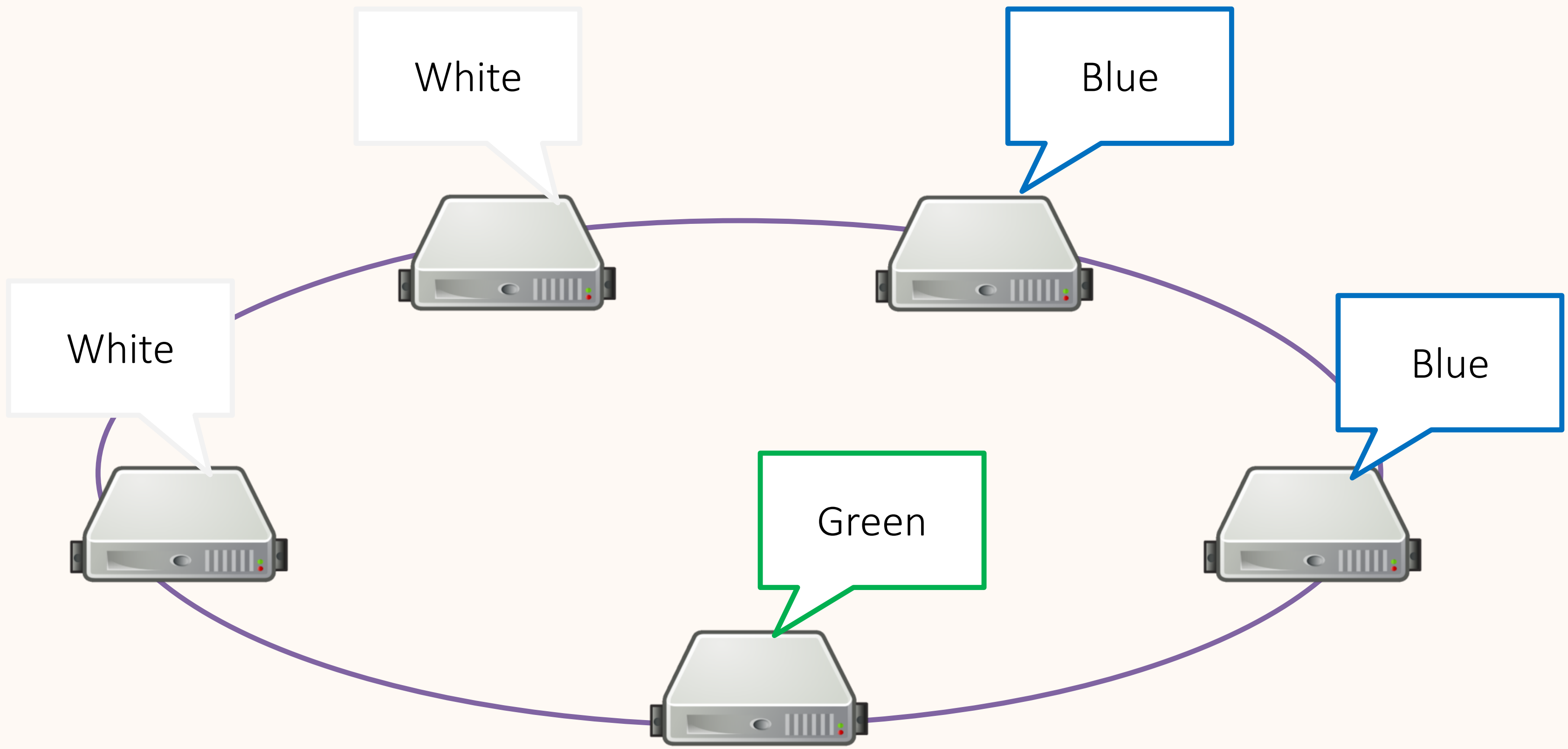
Majority consensus: A majority of nodes need to agree



Consensus.

Distributed Consensus

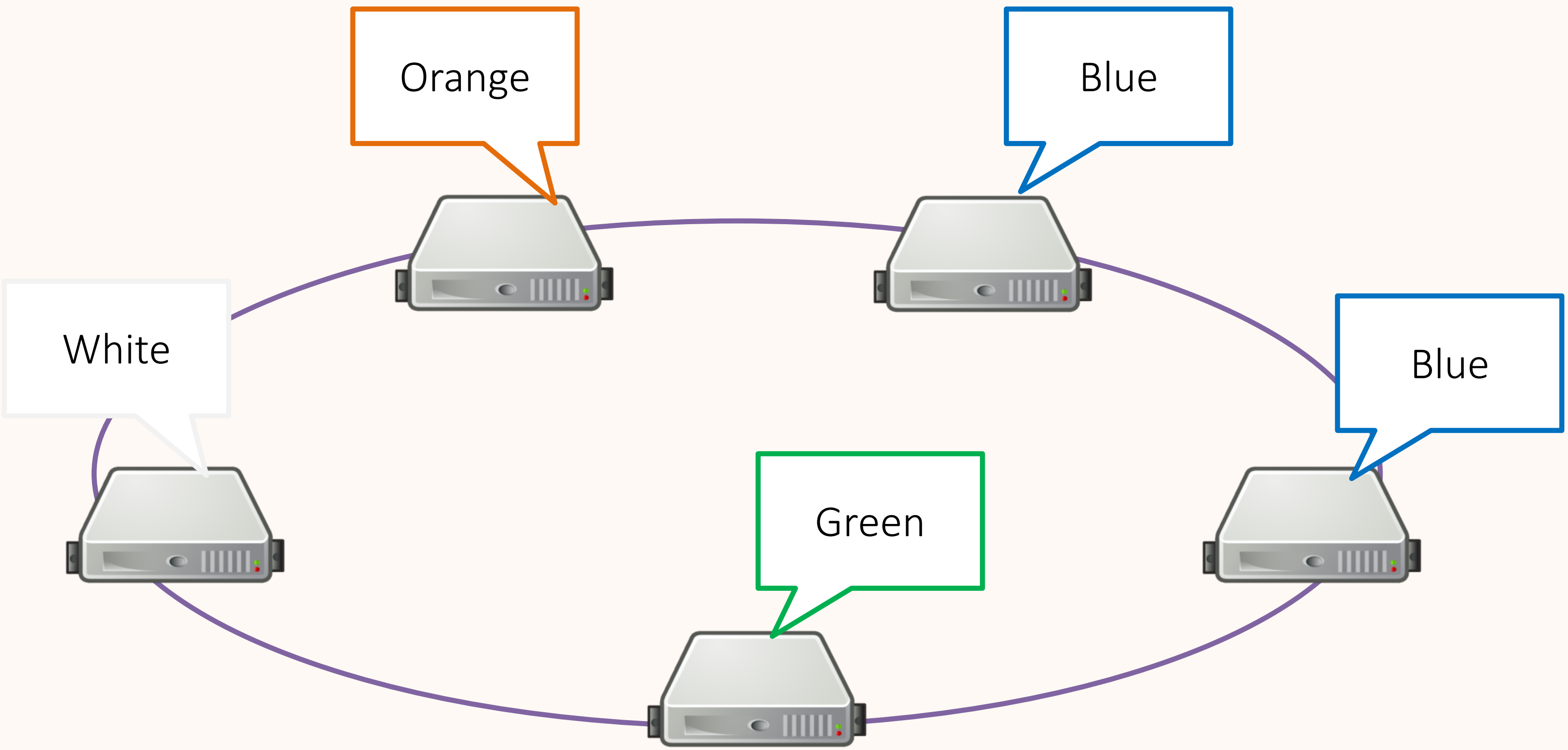
Majority consensus: A majority of nodes need to agree



No consensus.

Distributed Consensus

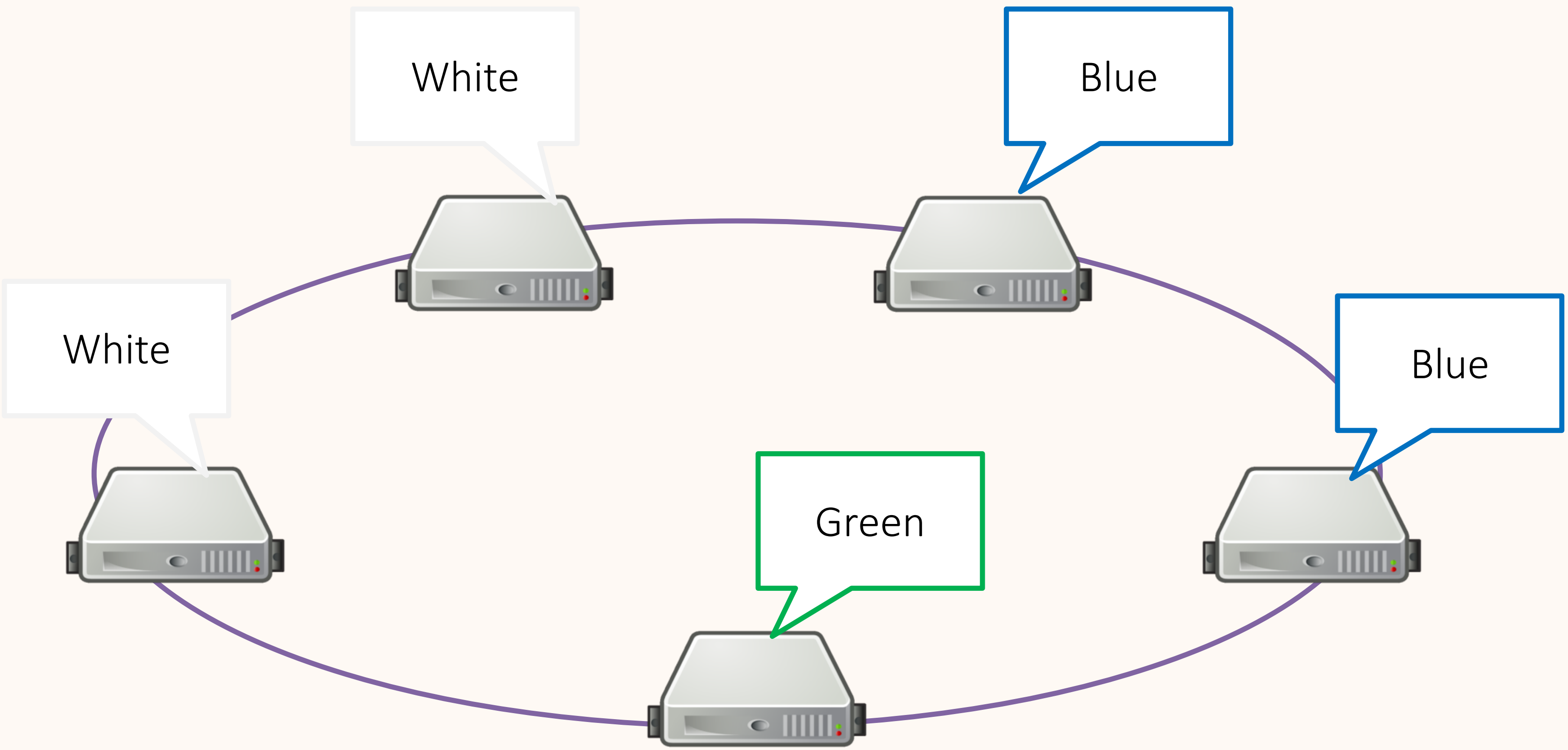
Plurality consensus: A plurality of nodes need to agree



Consensus.

Distributed Consensus

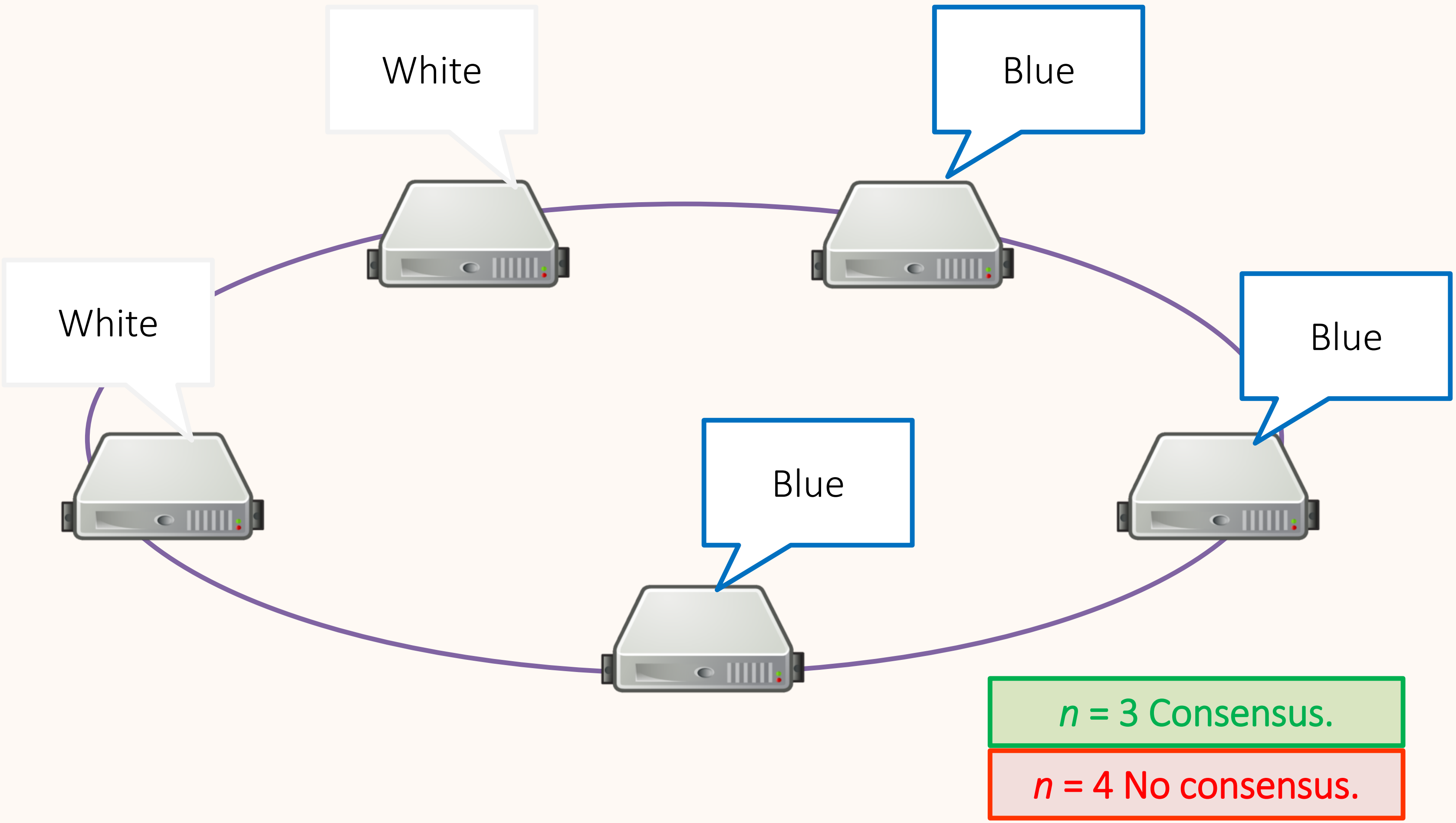
Plurality consensus: A plurality of nodes need to agree



No consensus.

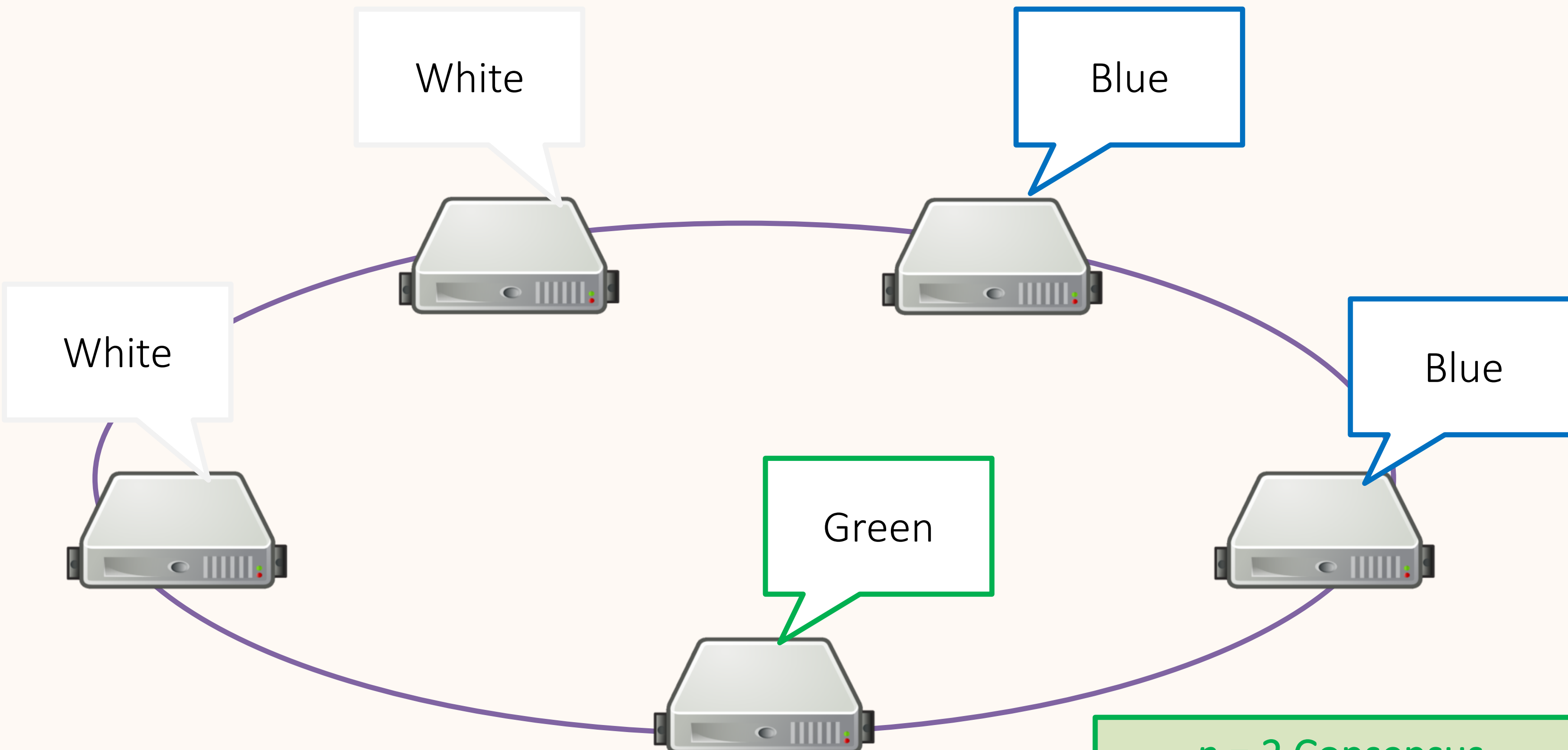
Distributed Consensus

Quorum consensus: n nodes need to agree



Distributed Consensus

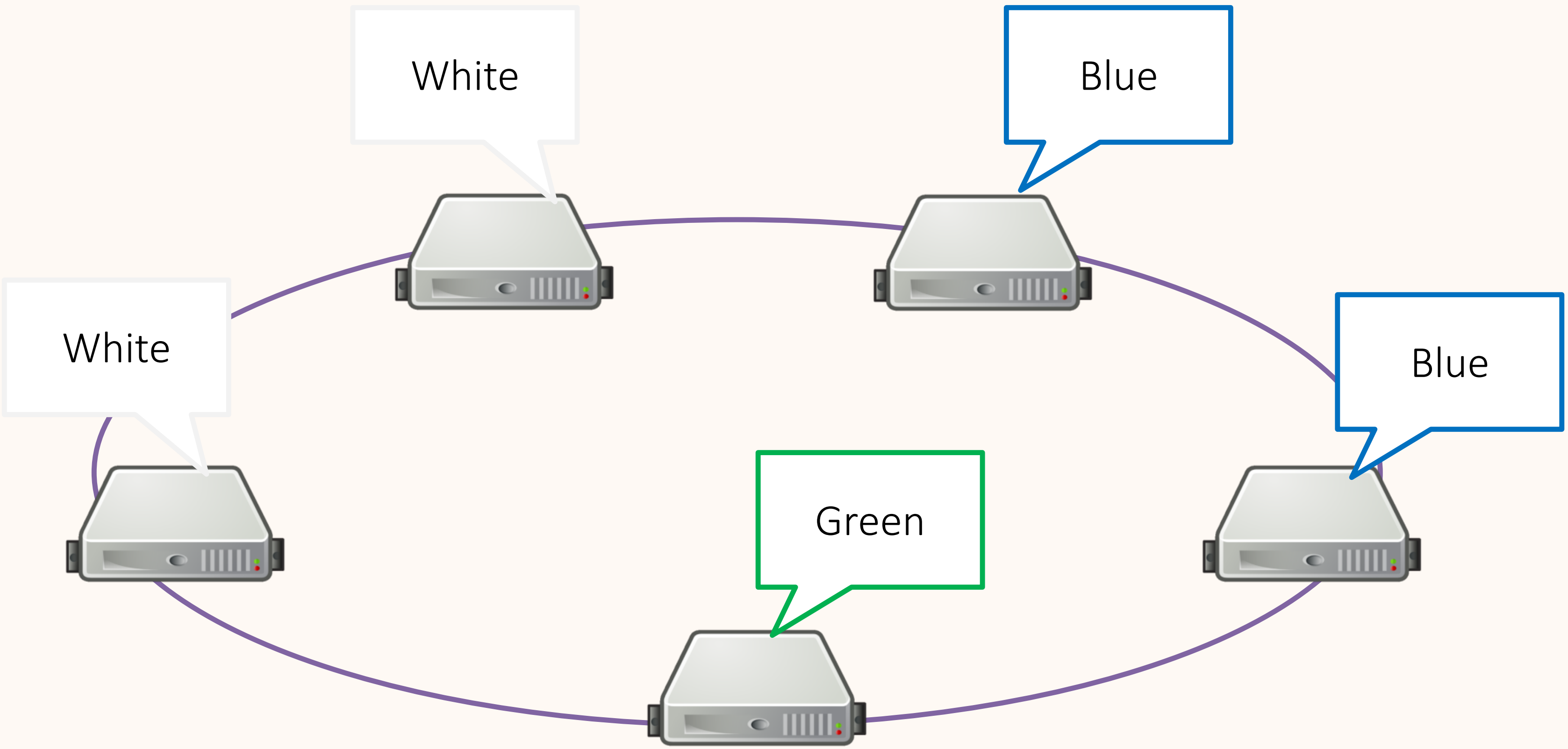
Quorum consensus: n nodes need to agree



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

Distributed Consensus

Quorum consensus: n nodes need to agree

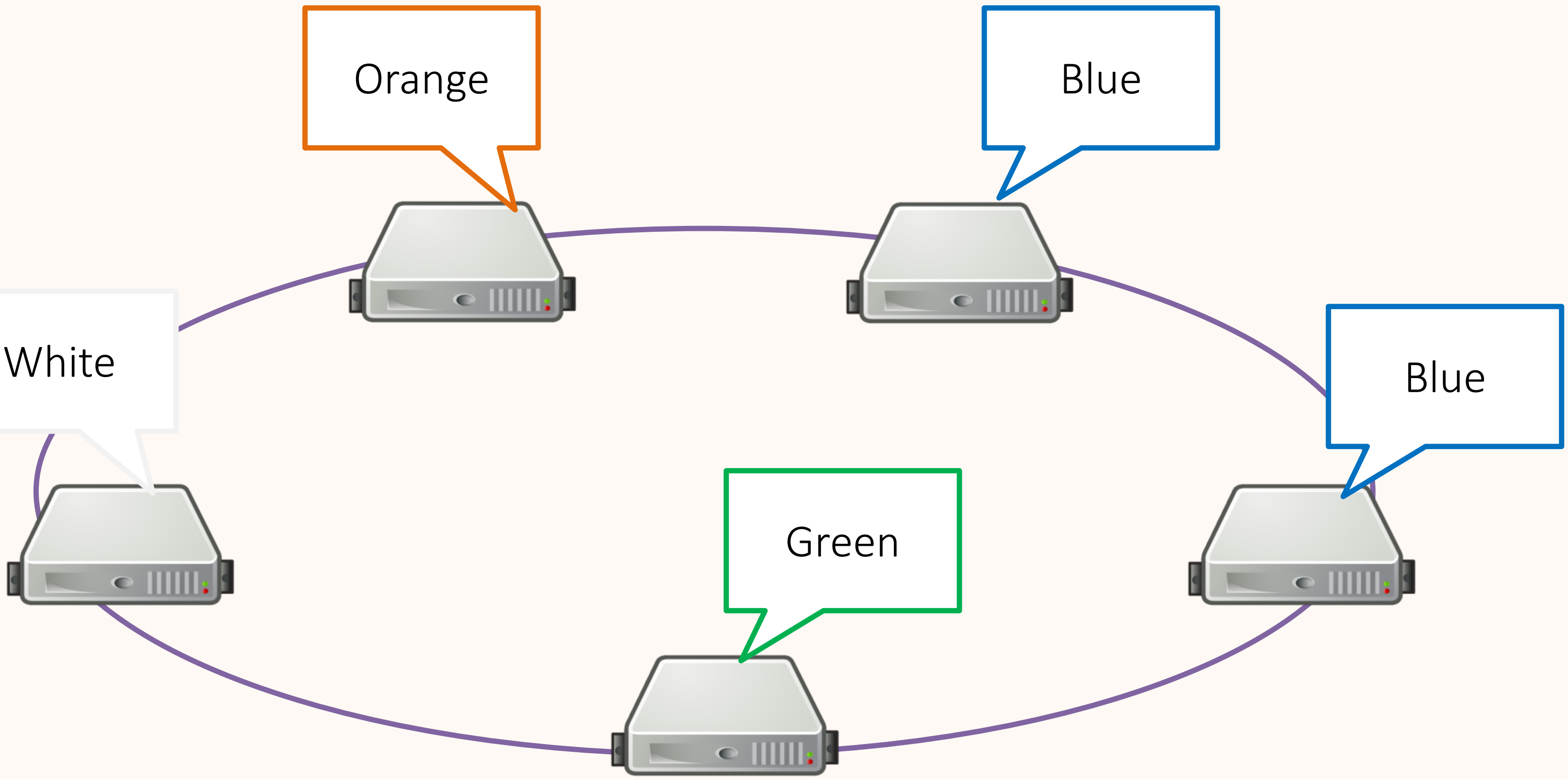


Value of n needed for unique consensus with N nodes?

$n > N/2$

Distributed Consensus

Consensus off: Take first answer



Consensus.

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

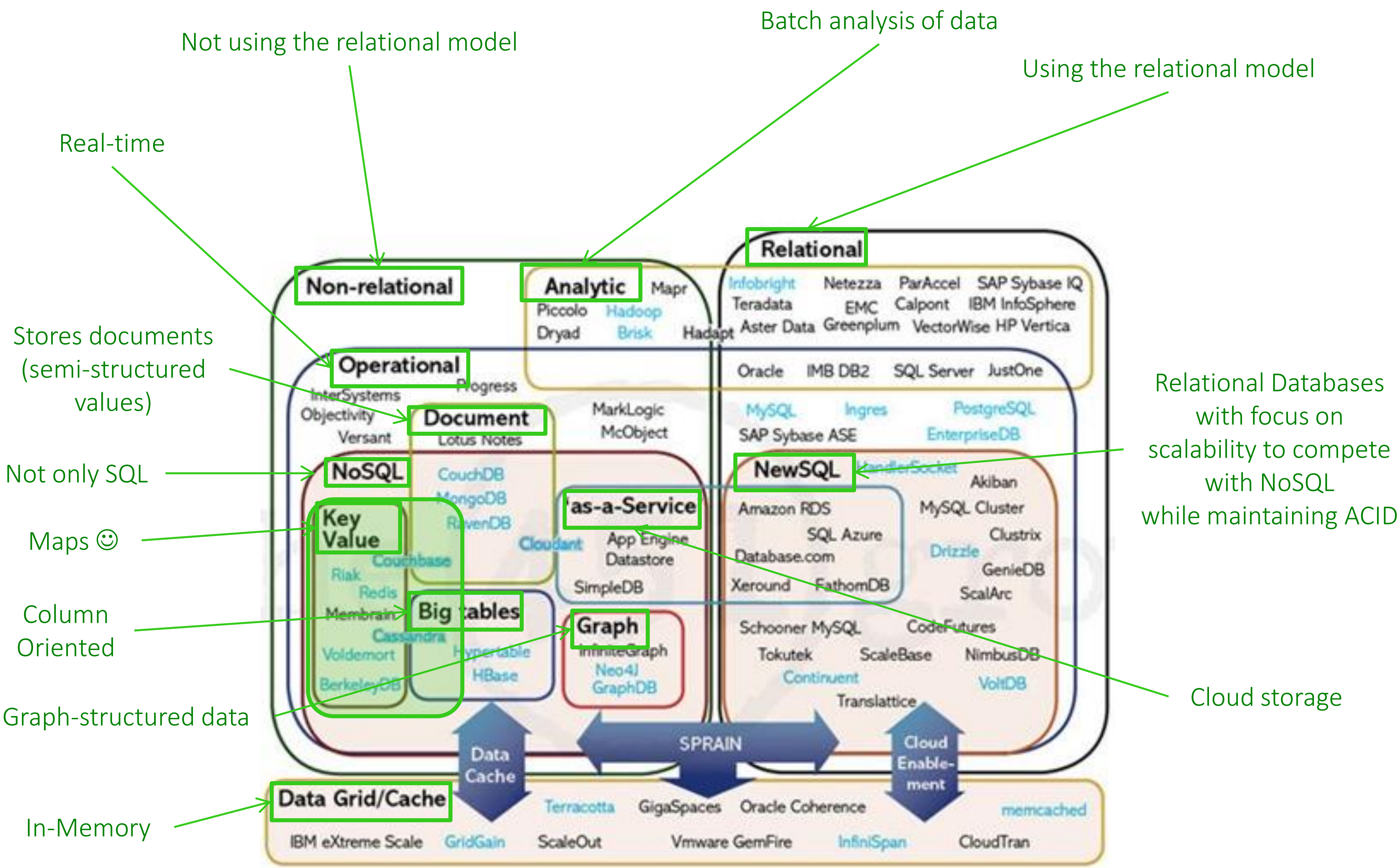
Consensus off: Take first answer

Choice is application dependent:

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

NoSQL: KEY–VALUE STORE

The Database Landscape



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

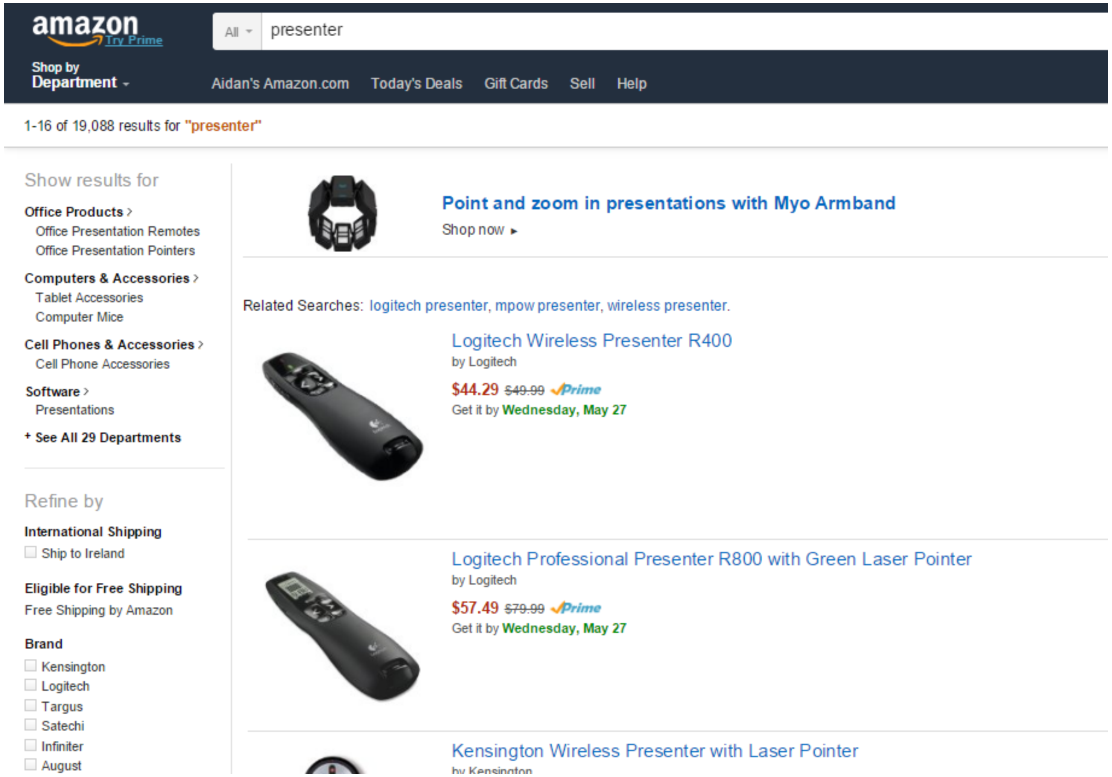
Key	Value
country:Afghanistan	capital@city:Kabul,continent:Asia,pop:31108077#2011
country:Albania	capital@city:Tirana,continent:Europe,pop:3011405#2013
…	…
city:Kabul	country:Afghanistan,pop:3476000#2013
city:Tirana	country:Albania,pop:3011405#2013
…	…
user:10239	basedIn@city:Tirana,post:{103,10430,201}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

… 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

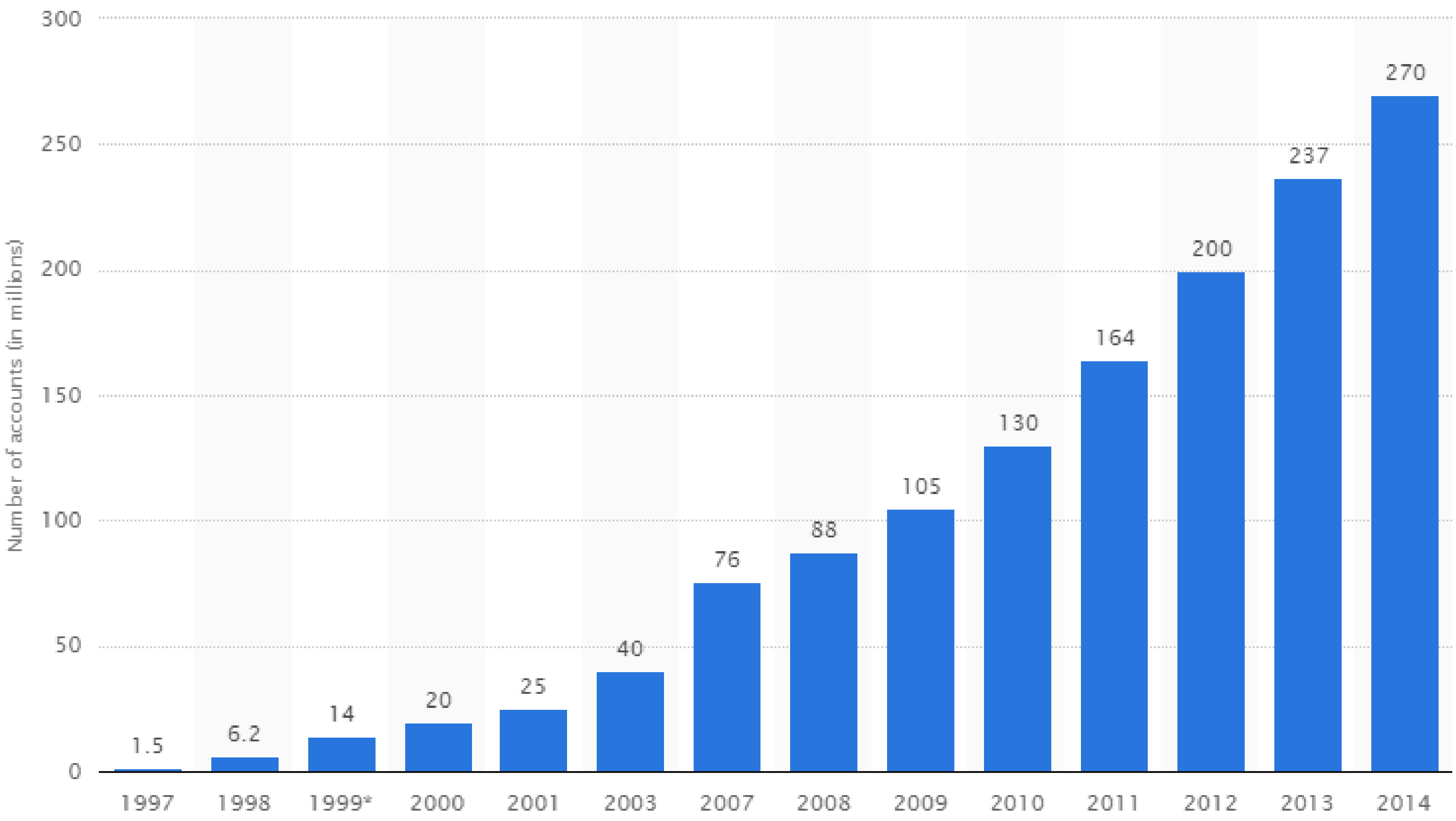
Customer info: shopping cart, account, etc.

The Amazon Scenario

Recommendations, etc.:

The Amazon Scenario

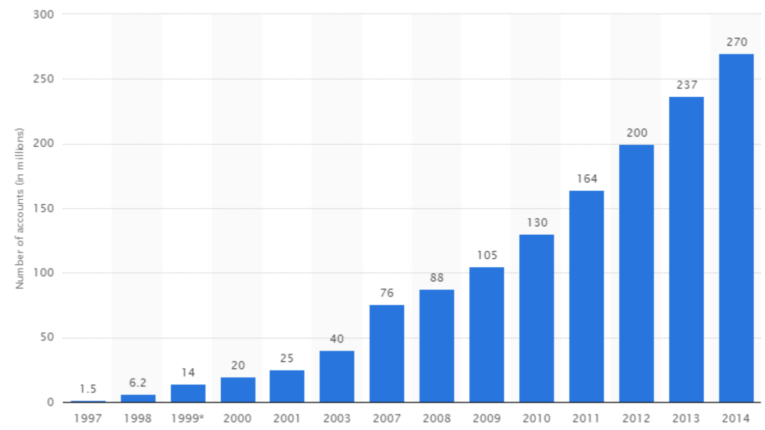
- Amazon customers:



The Amazon Scenario

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

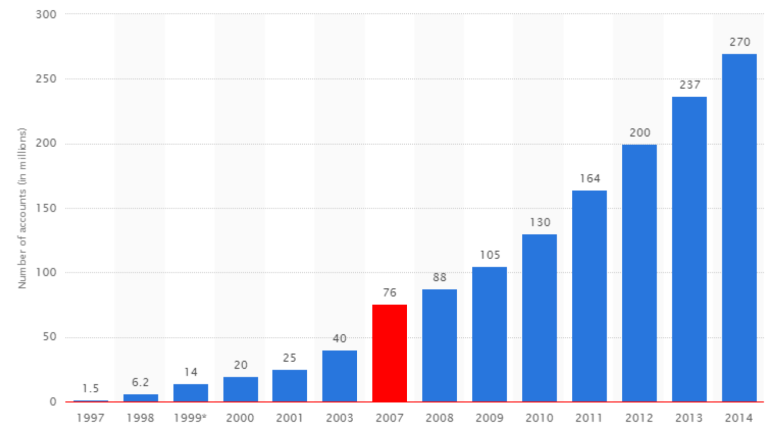
Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

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)

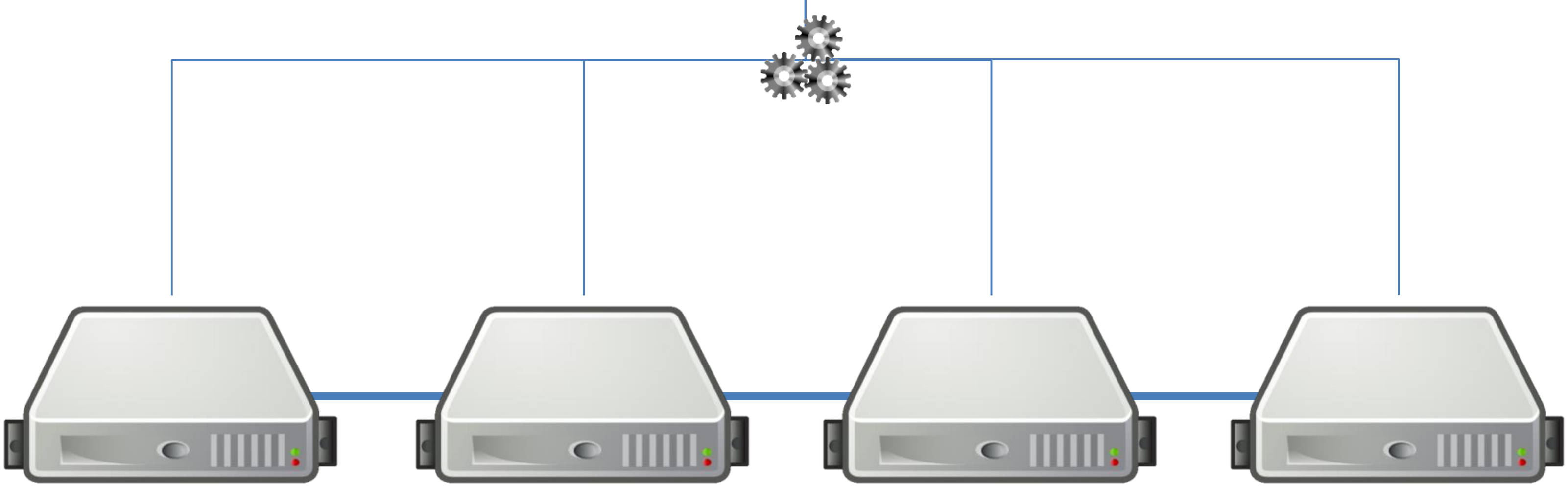
Don't need full SQL, don't need full ACID

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}
...	...

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



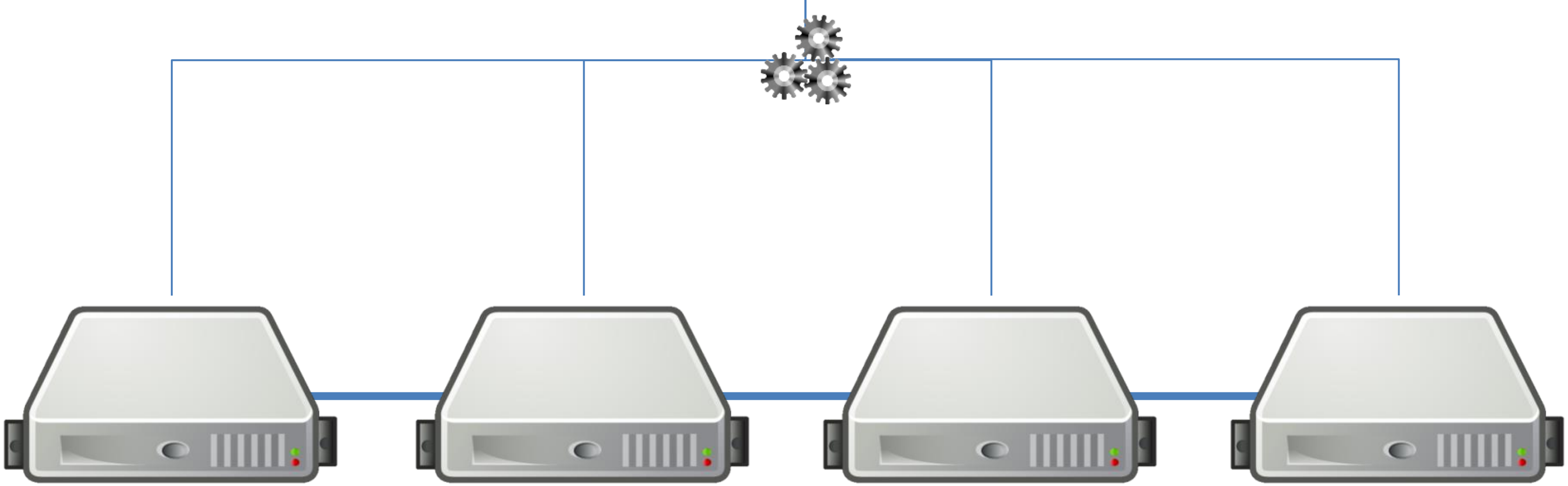
Key–Value Store: Distribution

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}
...	...

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

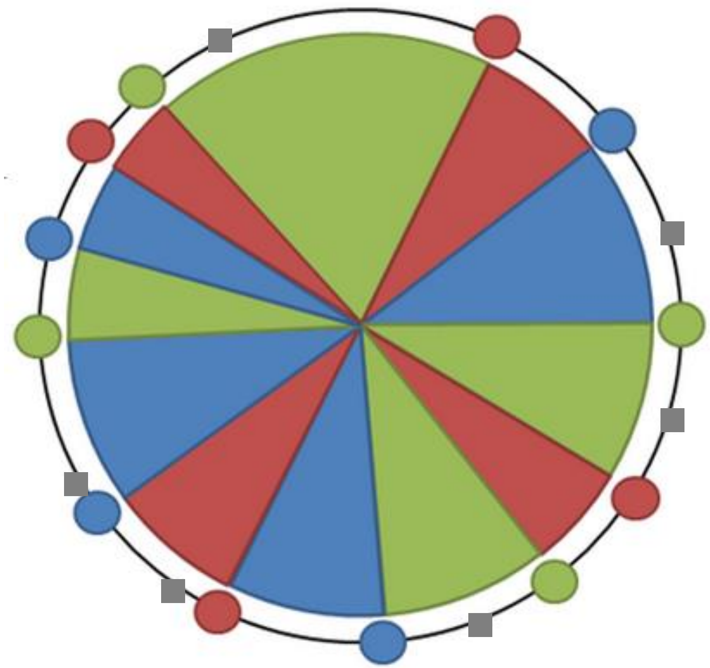


Consistent Hashing

Avoid re-hashing everything

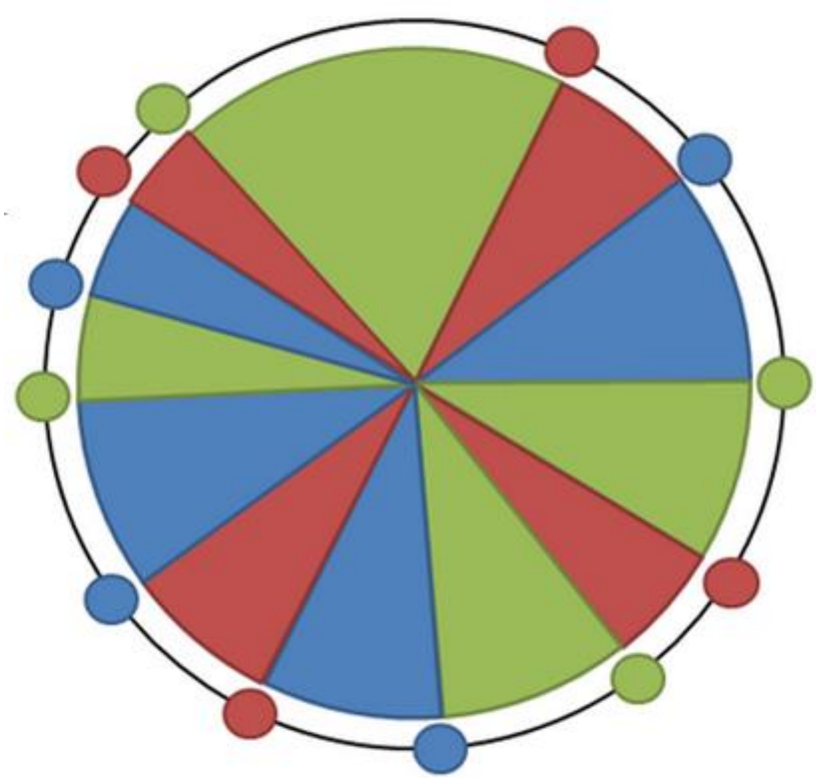
- Hash using a ring
- Each machine picks n pseudo-random points on the ring
- Machine responsible for arc after its point
- If a machine leaves, its range moves to previous machine
- If machine joins, it picks new points
- Objects mapped to ring ☺

How many keys (on average) would need to be moved if a machine joins or leaves?



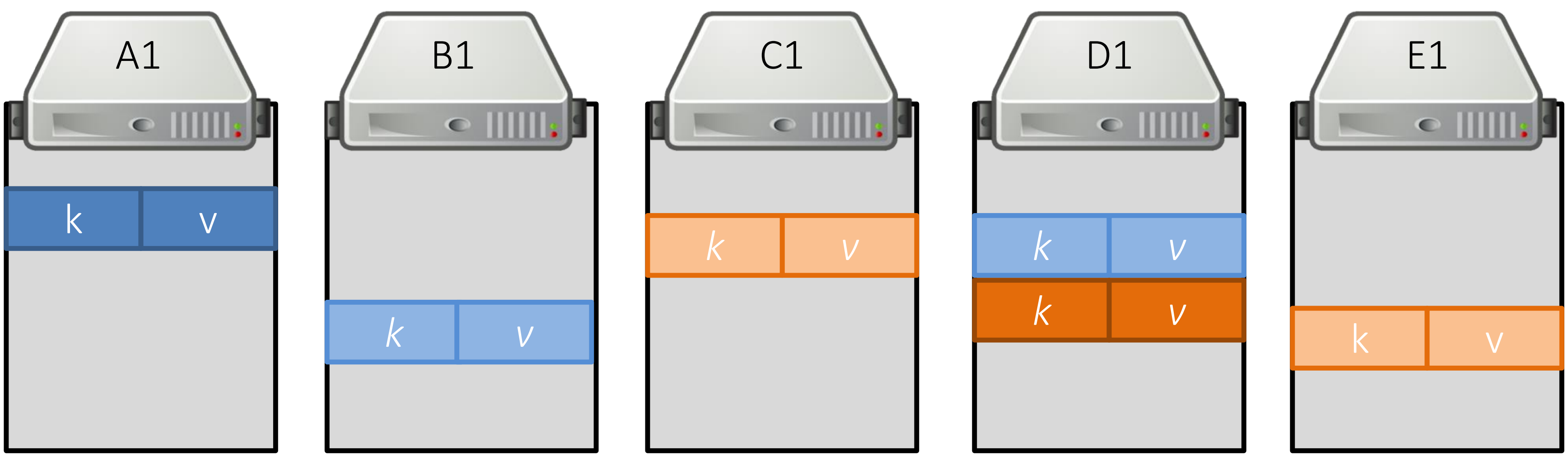
Amazon Dynamo: Hashing

- Consistent Hashing (128-bit MD5)



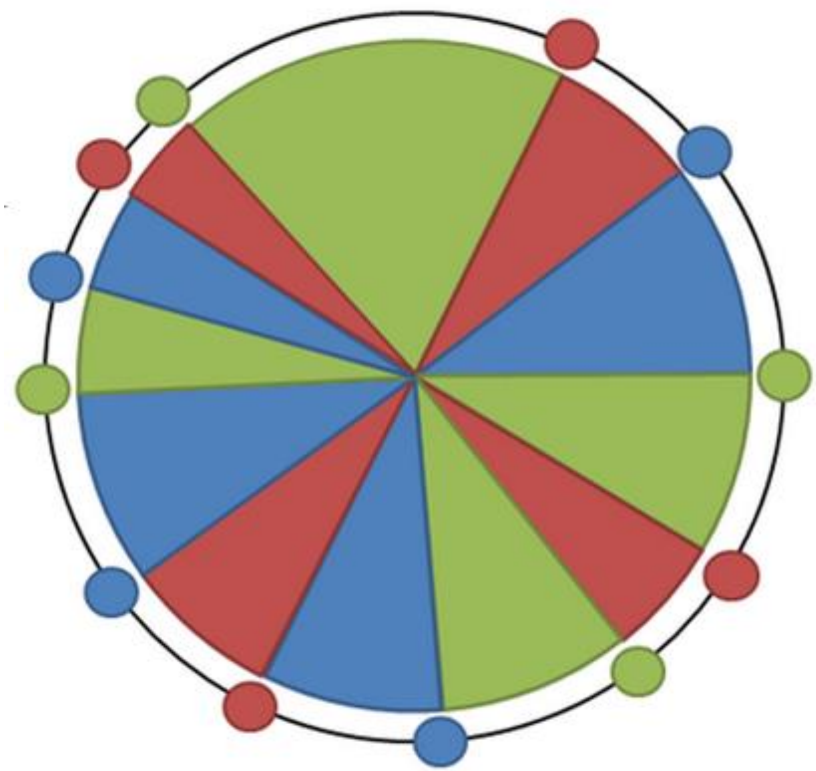
Amazon Dynamo: Replication

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



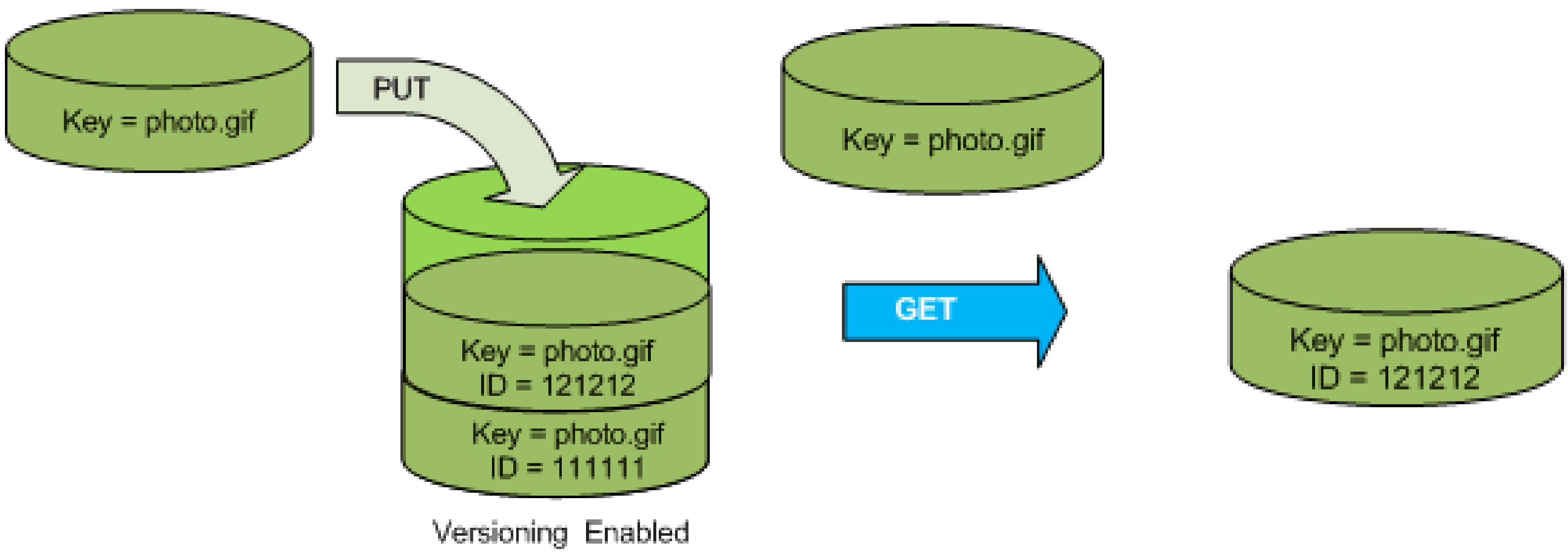
Amazon Dynamo: Replication

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



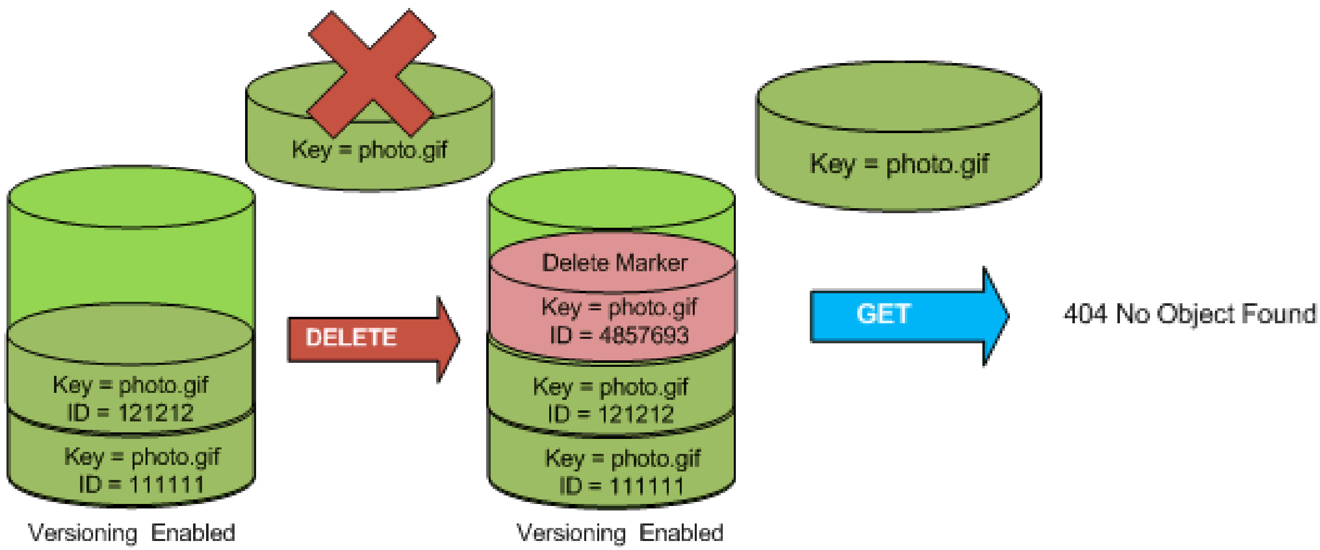
Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - PUT doesn't overwrite: pushes version
  - GET returns most recent version



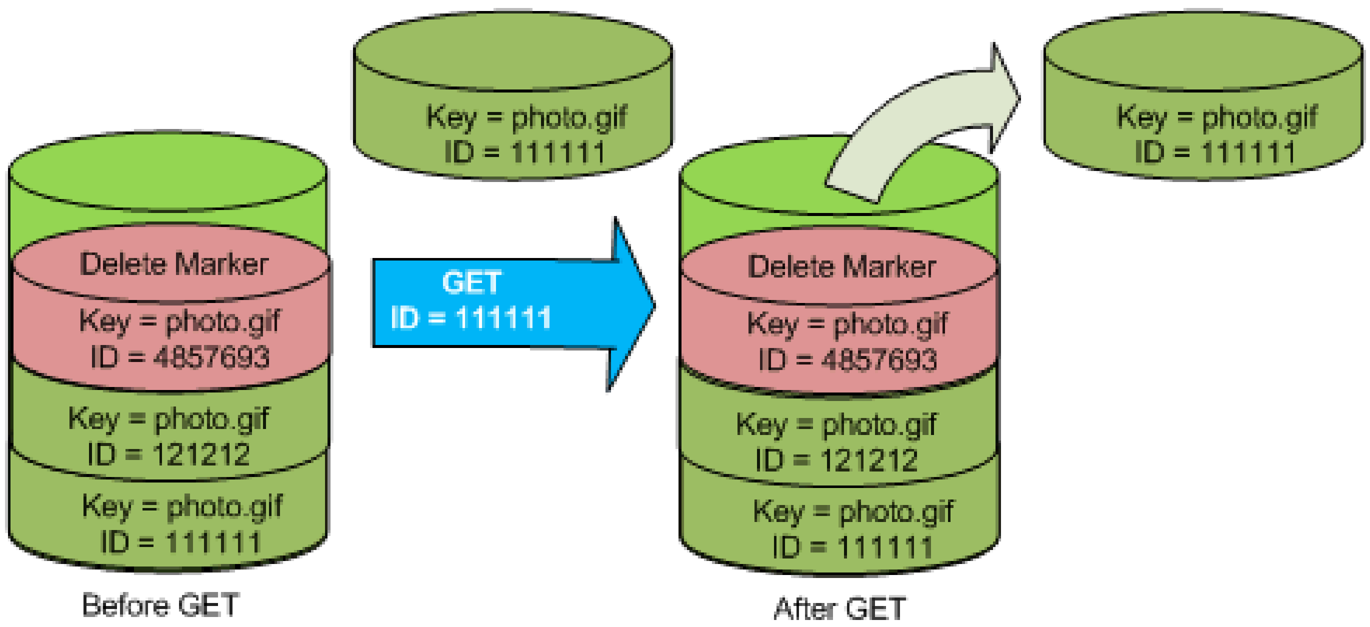
Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - DELETE doesn't wipe
  - GET will return not found



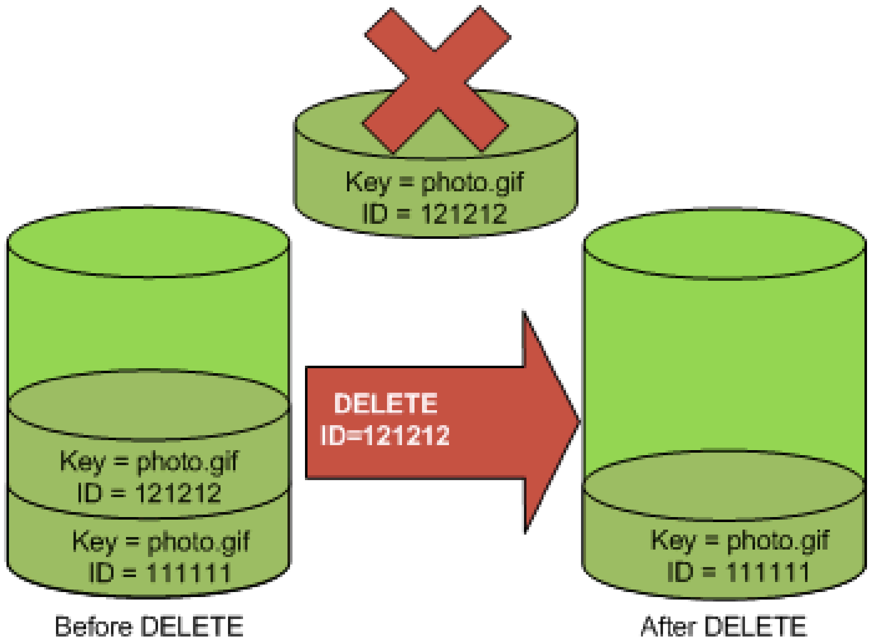
Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - GET by version



Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - PERMANENT DELETE by version … wiped



Amazon Dynamo: Model

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

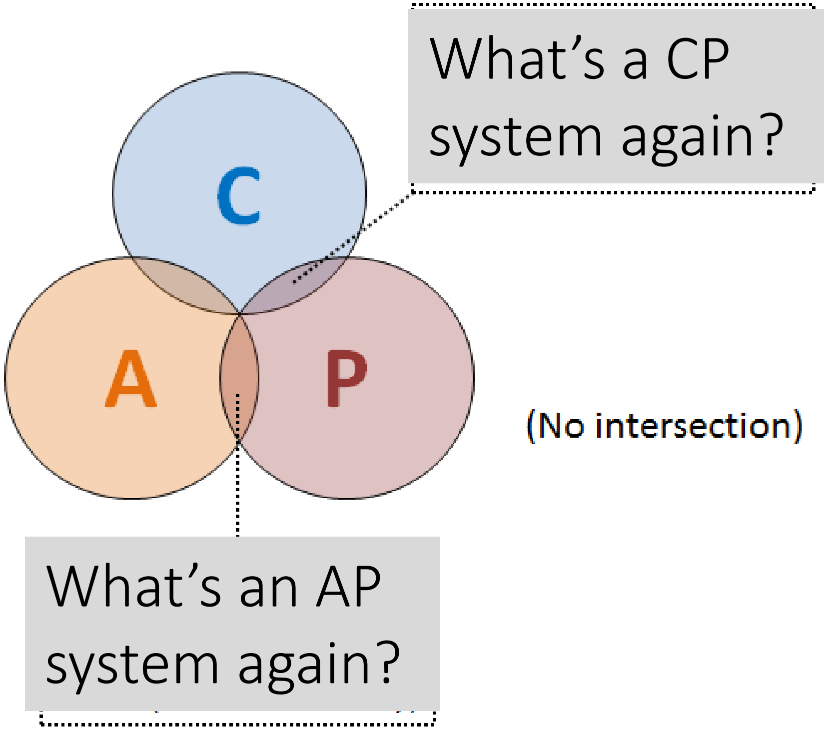
Countries	
Primary Key	**Value**
Afghanistan	capital:Kabul,continent:Asia,pop:31108077#2011
Albania	capital:Tirana,continent:Europe,pop:3011405#2013
…	…

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

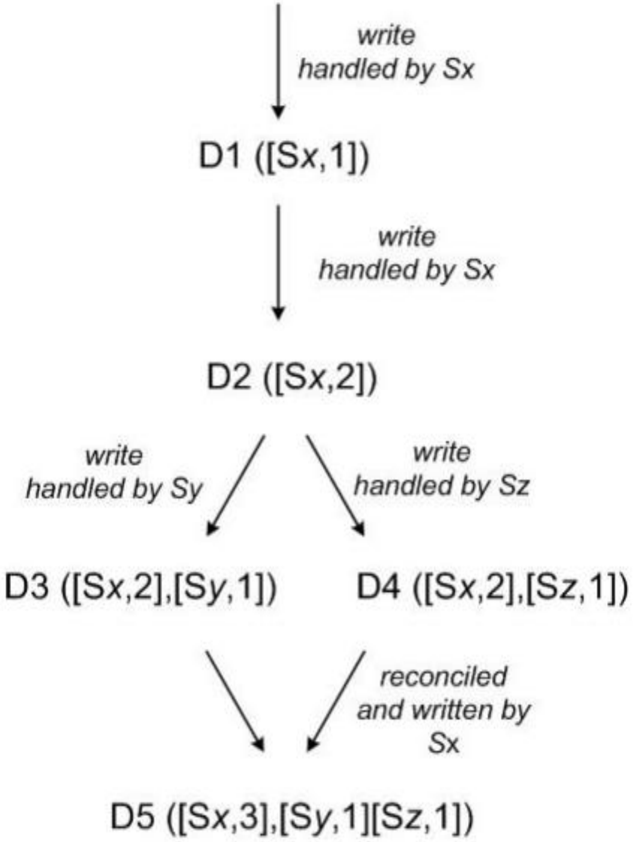
- Gossiping
  - Keep-alive messages sent between nodes with state
  - Dynamo largely decentralised (no master node)

- Quorums:
  - Multiple nodes responsible for a read (*R*) or write (*W*)
  - At least *R* or *W* nodes acknowledge for success
  - Higher *R* or *W* = Higher consistency, lower availability

- Hinted Handoff
  - For transient failures
  - A node "covers" for another node while it is down

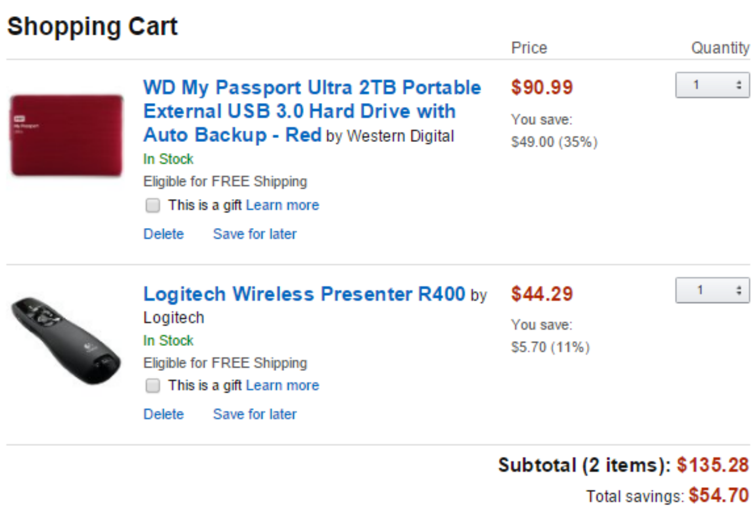
Amazon Dynamo: Consistency

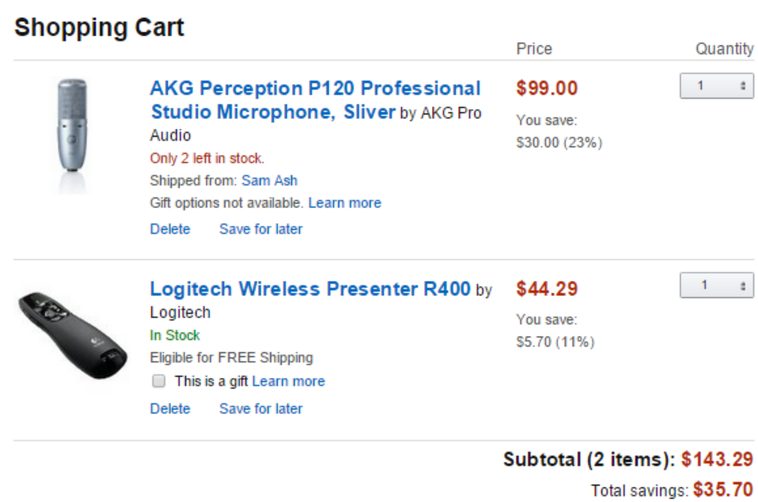
- Vector Clock:
  - A list of pairs indicating a node and time stamp
  - Used to track branches of revisions



Amazon Dynamo: Consistency

- Two versions of one shopping cart:





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

Application knows best

(... and must support multiple versions being returned)

Amazon Dynamo: Consistency

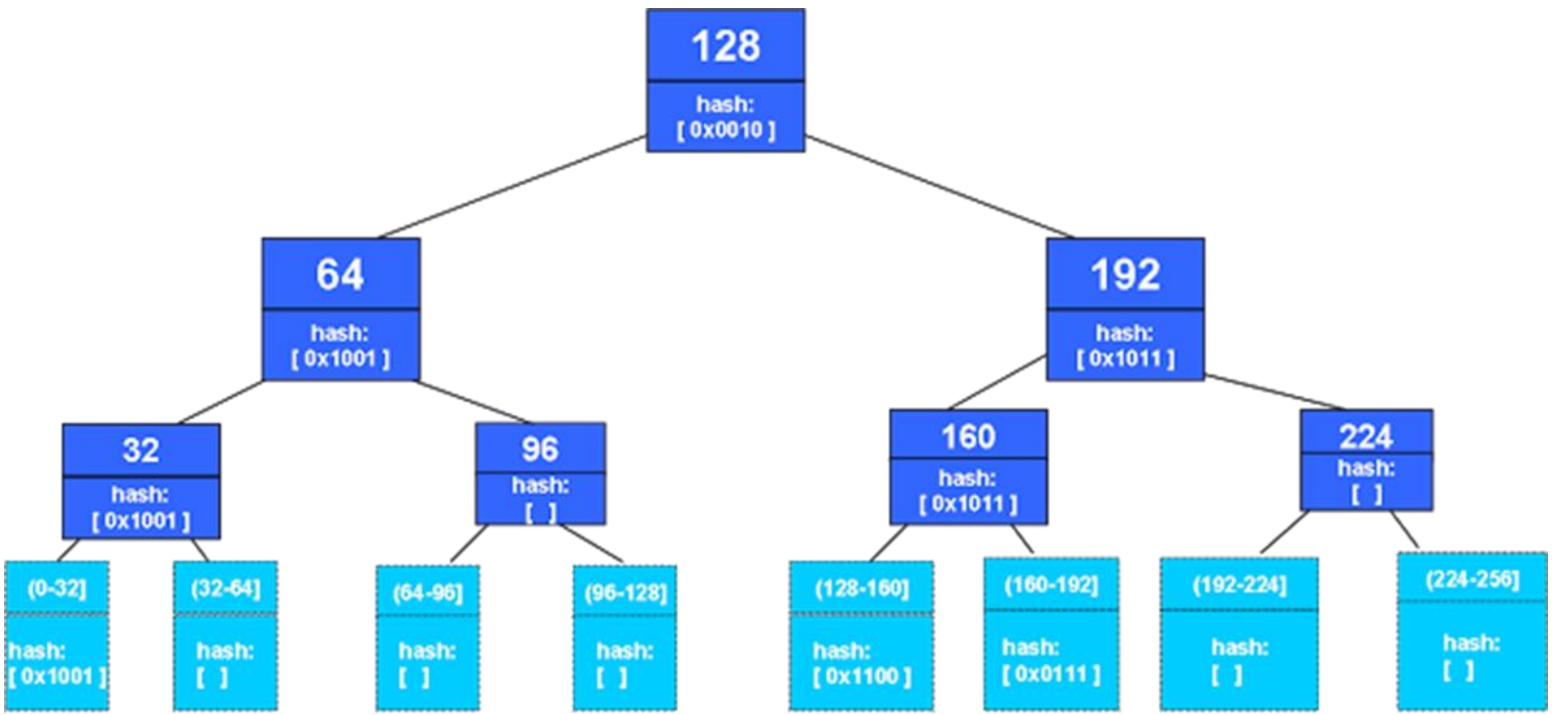
Key	Value
country:Afghanistan	capital@city:Kabul,continent:Asia,pop:31108077#2011
country:Albania	capital@city:Tirana,continent:Europe,pop:3011405#2013
...	...
city:Kabul	country:Afghanistan,pop:3476000#2013
city:Tirana	country:Albania,pop:3011405#2013
...	...
user:10239	basedIn@city:Tirana,post:{103,10430,201,408}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

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}
...	...

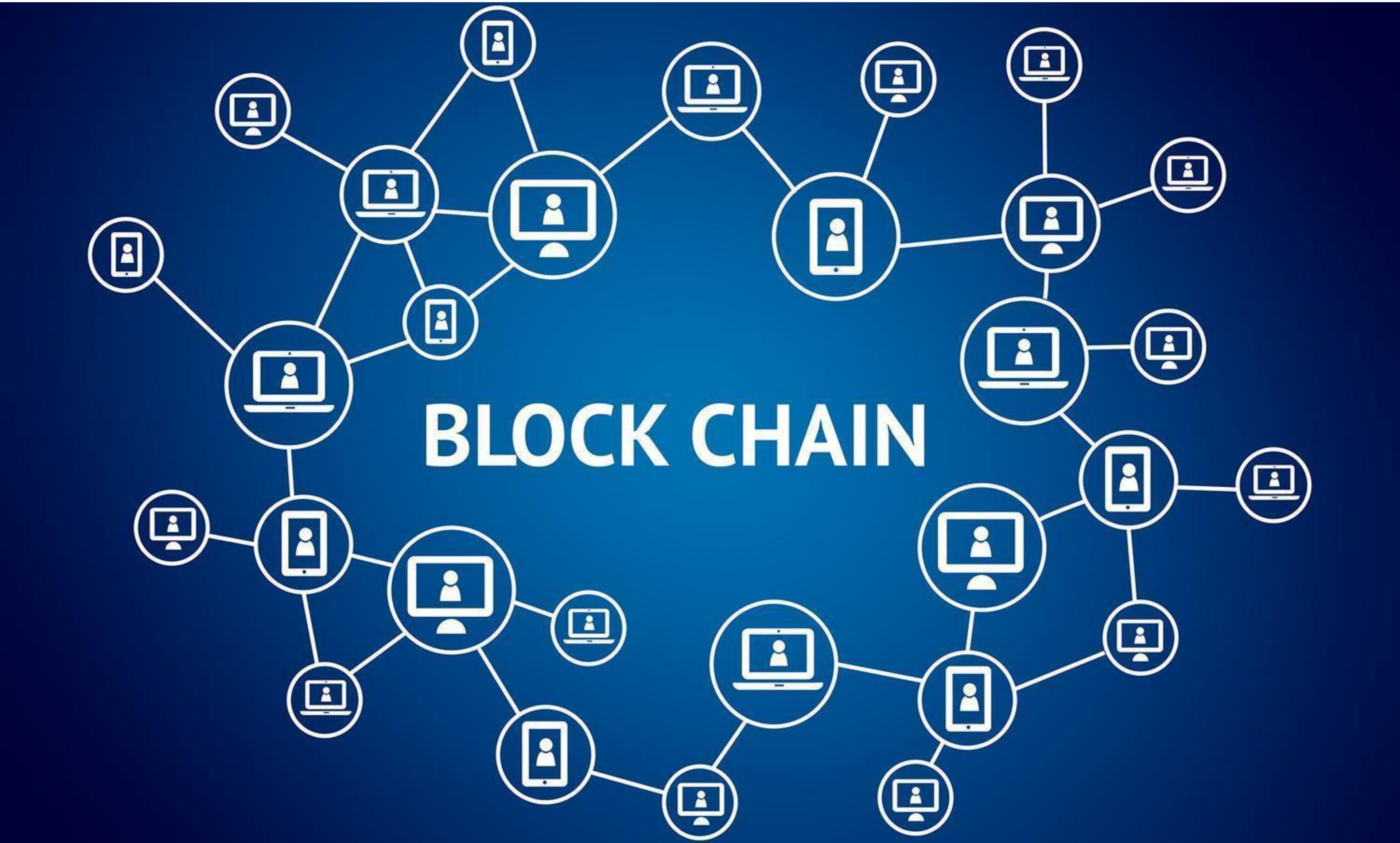
How can we efficiently verify that two copies of a block of data are the same (and find where the differences are)?

Amazon Dynamo: Merkle Trees

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



Aside: Merkle Trees also used in ...



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

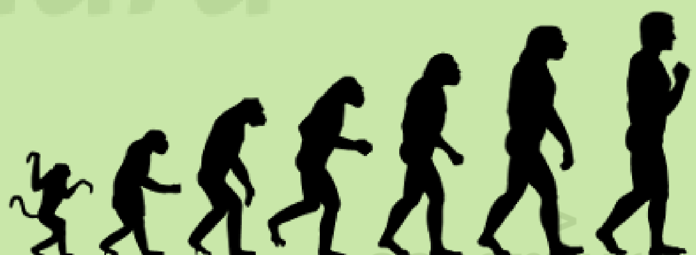
OTHER KEY–VALUE STORES

Other Key–Value Stores

Other Key–Value Stores

Other Key–Value Stores

Other Key–Value Stores



Evolved into a
tabular store …

Tabular / Column Family

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 [illegible] indexing, Google Earth, and Google Finance. [illegible] applications place very different demands on B[illegible]able, both in terms of data size (from URLs to web [illegible]ges to satellite imagery) and [illegible] latency requirements (fro[illegible] backend bulk processing t[illegible] real-time da[illegible] serv[illegible]). Des[illegible] these varied demands, [illegible]able has [illegible] prov[illegible] flexible, high[illegible] for[illegible]ce solutio[illegible] these G[illegible] products. I[illegible] paper[illegible] describe the sim[illegible]ple data mode[illegible] 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 t[illegible]ocality properties of the data represented in the unde[illegible]ng storage. Data is indexed using row and column [illegible]es that can be arbitrary st[illegible]s. Bigta[illegible] treats d[illegible] as unin[illegible] strings, alth[illegible]h c[illegible]nts ofte[illegible] vari[illegible]s forms [illegible] structured [illegible] semi-struct[illegible]d da[illegible] into t[illegible] strings. Clients can [illegible]trol [illegible] locality of [illegible]ir d[illegible] through careful choi[illegible]s in their s[illegible]mas. Fi[illegible]ly, Bi[illegible]le schema parameters let client[illegible]amically control wh[illegible] to serve data out o[illegible]emory or [illegible]m disk.

Sectio[illegible] describes t[illegible] data model in more detail, and Section 3 p[illegible] overview of the client API. Sec-

Bigtable used for …

…

Bigtable: Data Model

"a sparse, distributed, persistent, multi-dimensional, sorted map."

- sparse: not all values form a dense square
- distributed: lots of machines
- persistent: disk storage (GFS)
- multi-dimensional: values with columns
- sorted: sorting lexicographically by row key
- map: look up a key, get a value

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

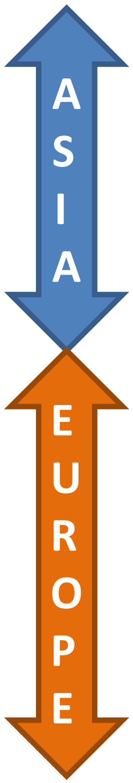
SORTED

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 …

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

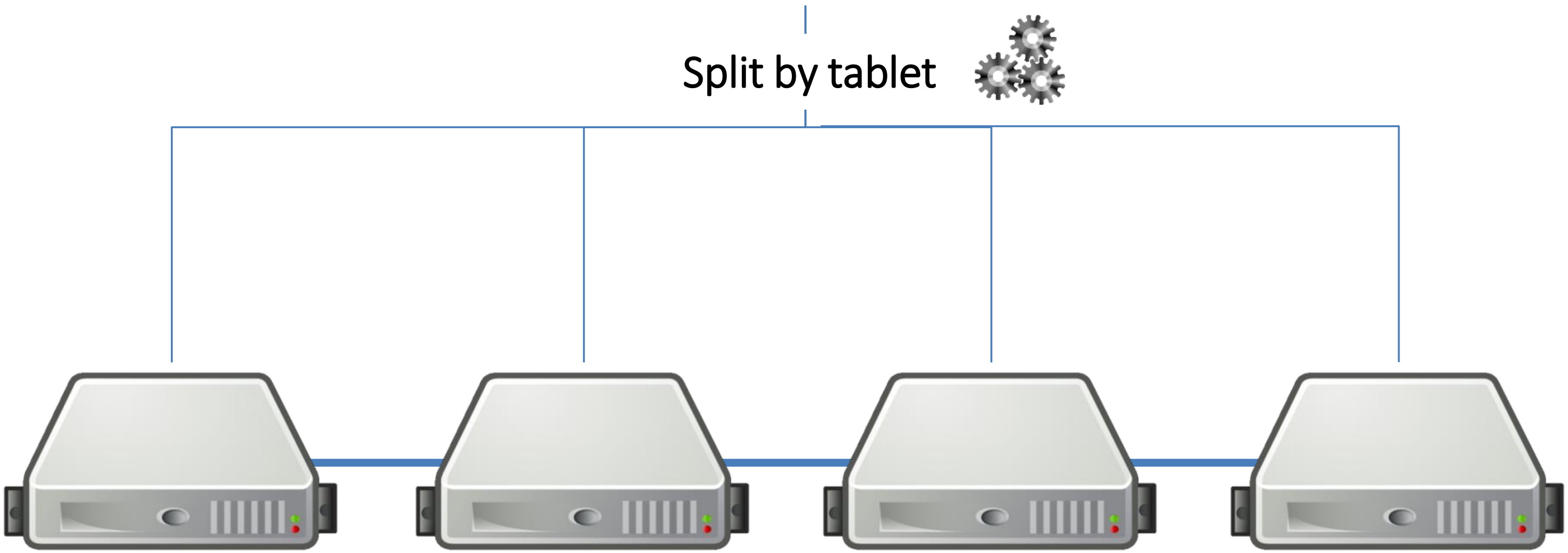
A real-world example of locality/sorting

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

Bigtable: Versioning

- Similar to Apache Dynamo
  - Cell-level
  - 64-bit integer time stamps
  - Inserts push down current version
  - Lazy deletions / periodic garbage collection
  - Two options:
    - keep last n versions
    - keep versions newer than t time

Bigtable: SSTable Map Implementation

- 64k blocks (default) with index in footer (GFS)
- Index loaded into memory, allows for seeks
- Can be split or merged, as needed

Writes? ?

	Primary Key	pol:capital		demo:pop-value		demo:pop-year	
0	Asia:Afghanistan	t_1	Kabul	t_1	31143292	t_1	2009
				t_2	31120978		
				t_4	31108077	t_4	2011
	Asia:Azerbaijan	…	…	…	…	…	…
	…	…	…	…	…	…	…
65536	Asia:Japan	…	…	…	…	…	…
	Asia:Jordan	…	…	…	…	…	…
	…	…	…	…	…	…	…

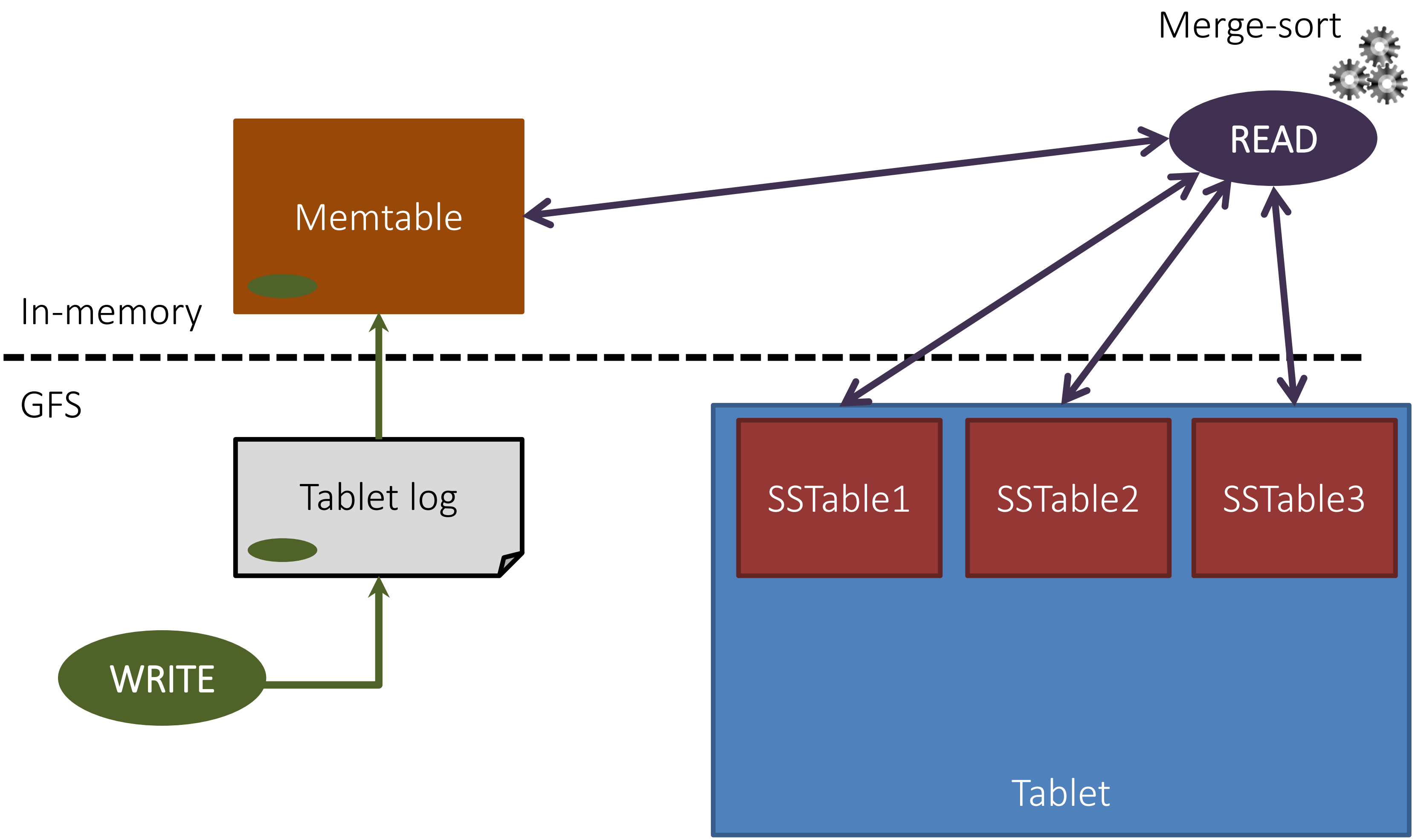
Index:

Block 0 / Offset 0 / Asia:Afghanistan

Block 1 / Offset 65536 / Asia: Japan

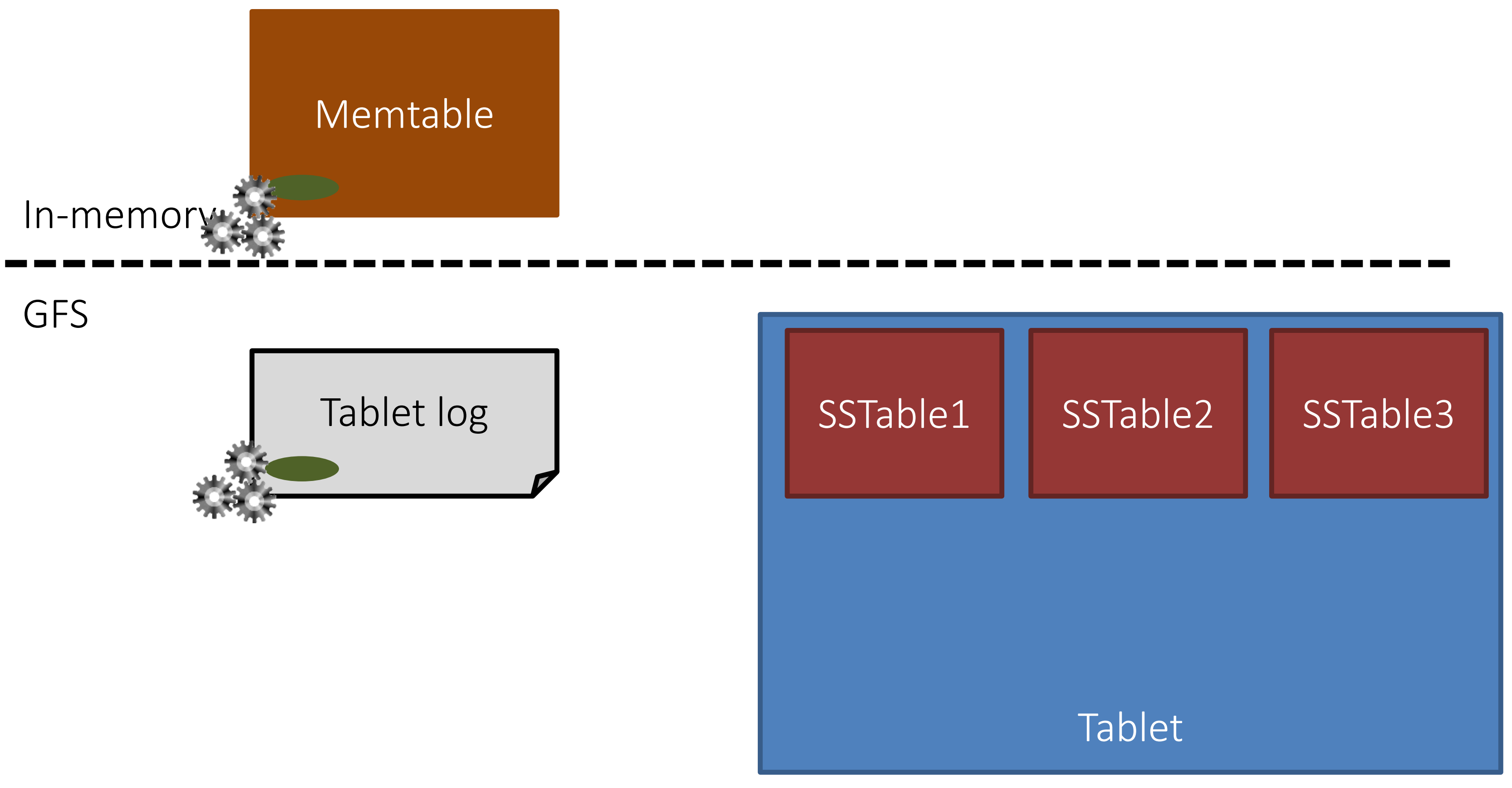
Bigtable: Buffered/Batched Writes

What's the danger?



Bigtable: Redo Log

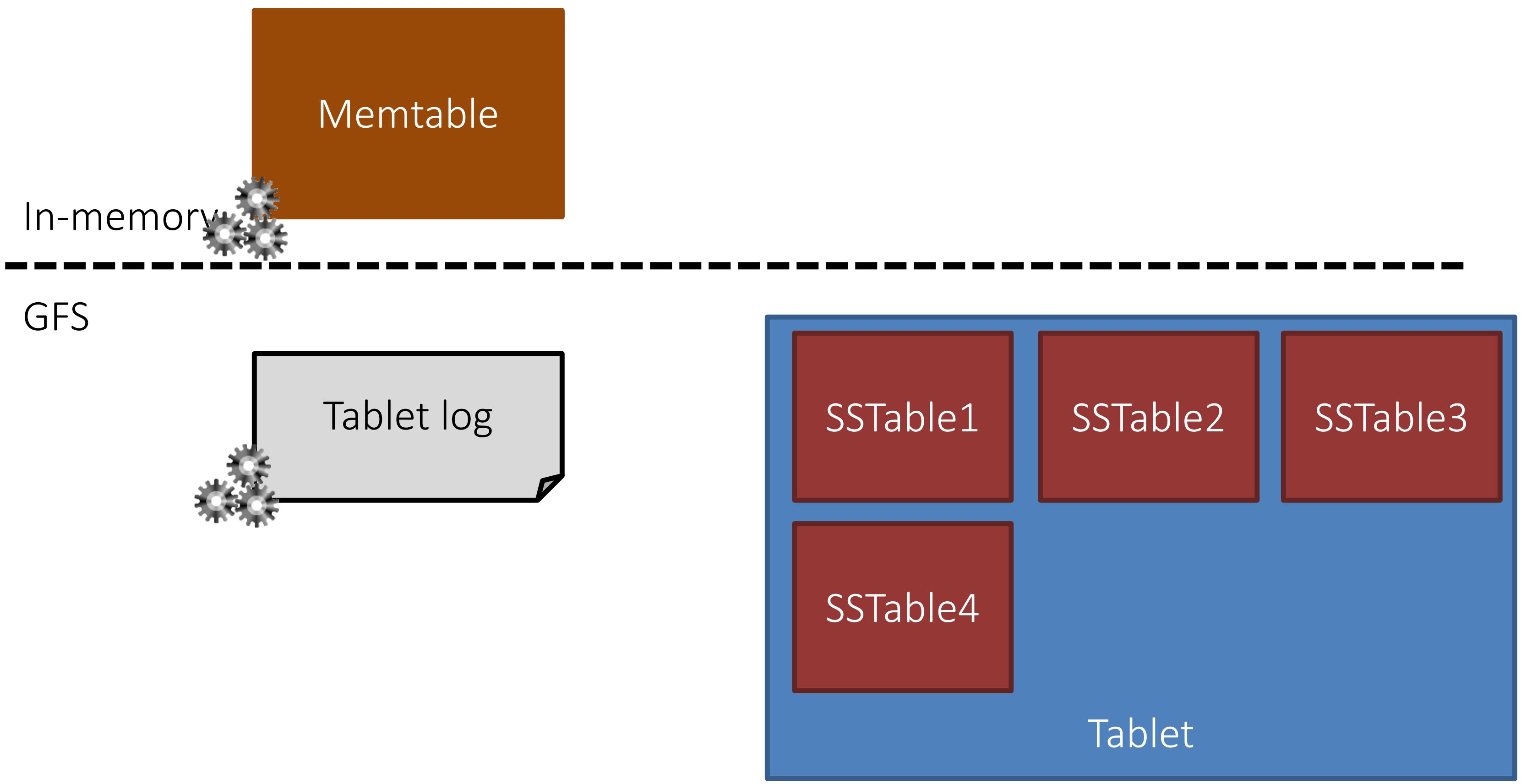
- If machine fails, Memtable redone from log



Bigtable: Minor Compaction

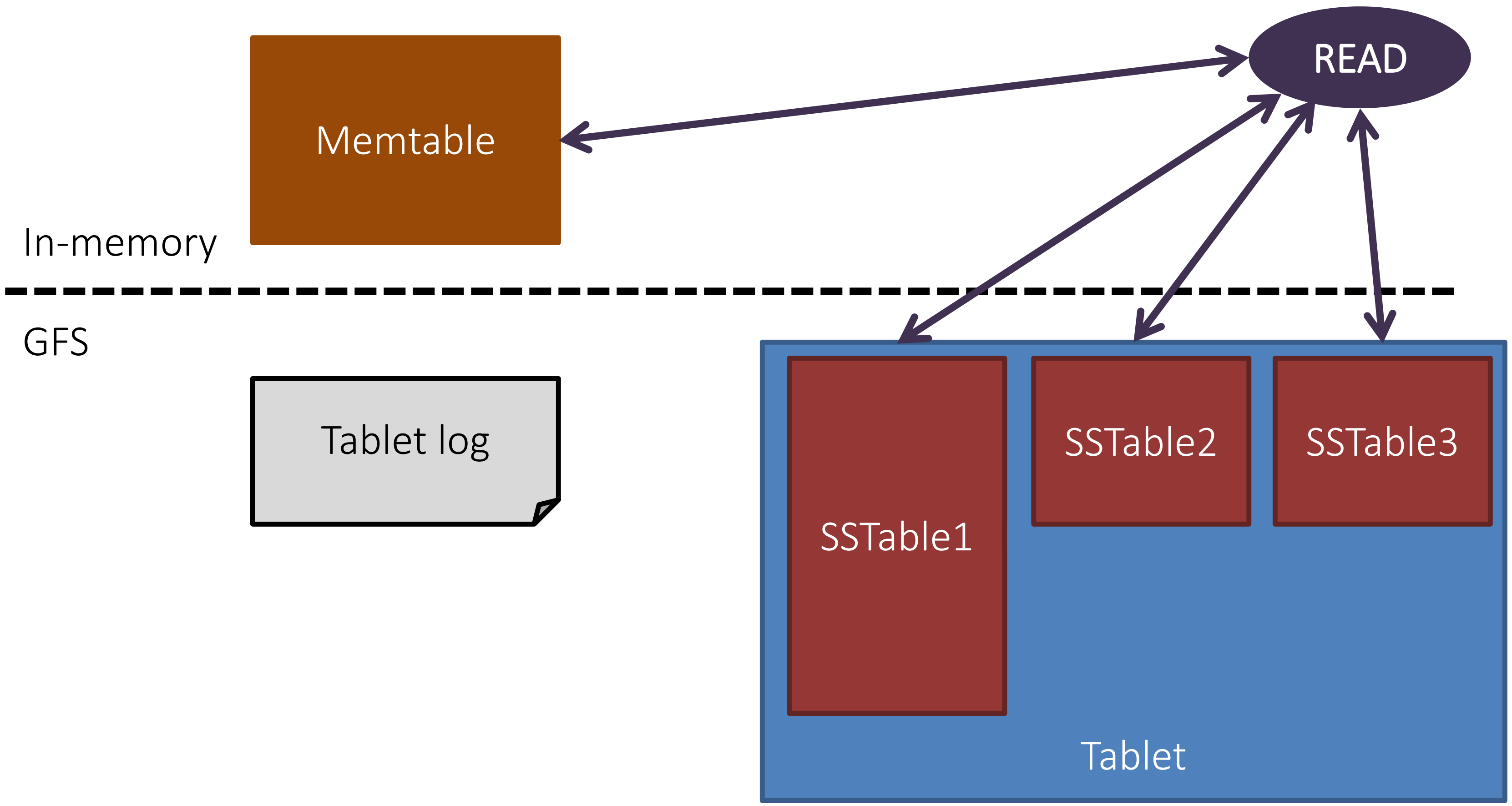
- When full, write Memtable as SSTable

Problem with performance? ?



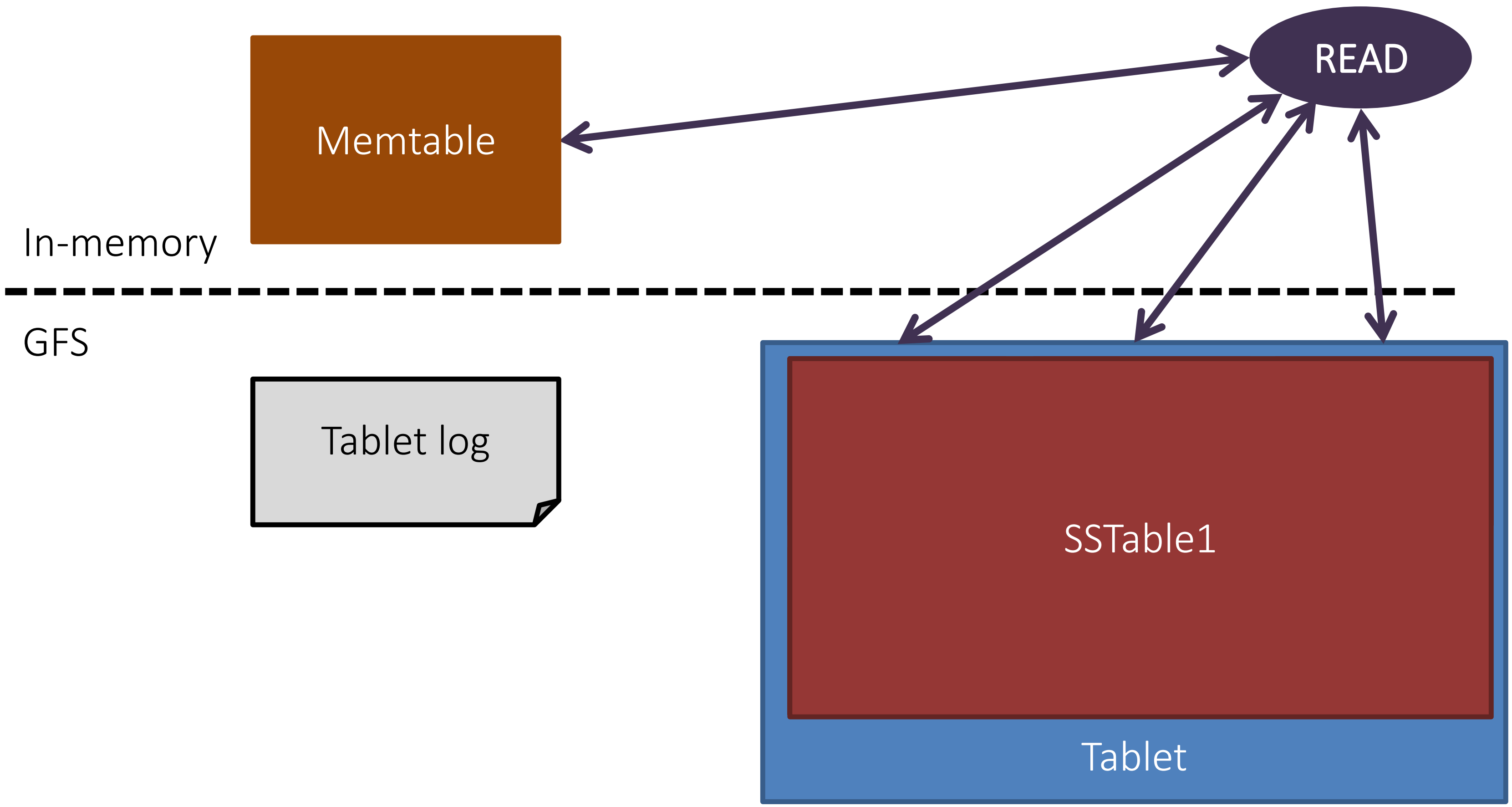
Bigtable: Merge Compaction

- Merge some of the SSTables (and the Memtable)



Bigtable: Major Compaction

- Merge all SSTables (and the Memtable)
- Makes reads more efficient!



Bigtable: A Bunch of Other Things

- Hierarchy and locks: how to find and lock tablets
- Locality groups: Group multiple column families together; assigned a separate SSTable
- Select storage: SSTables can be persistent or in-memory
- Compression: Applied on SSTable blocks; custom compression can be chosen
- Caches: SSTable-level and block-level
- Bloom filters: Find negatives cheaply …

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

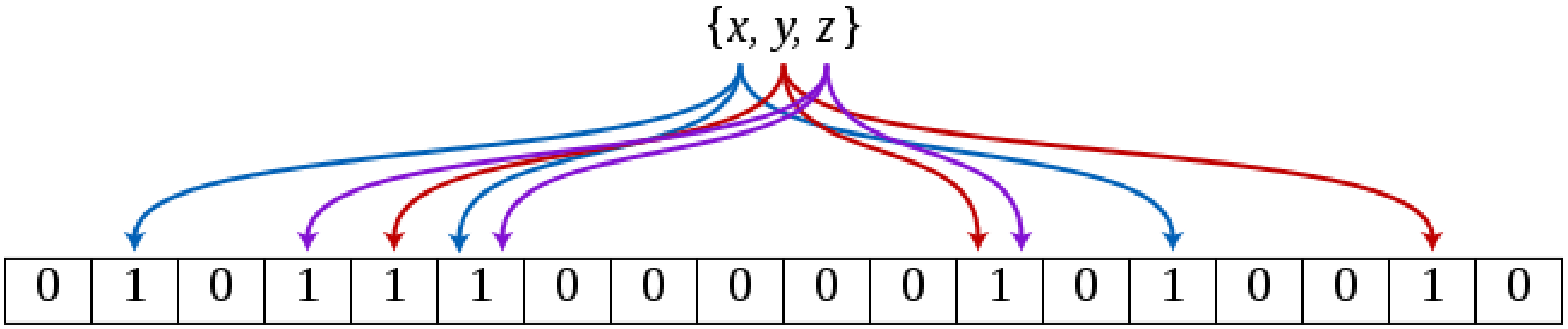
Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place 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

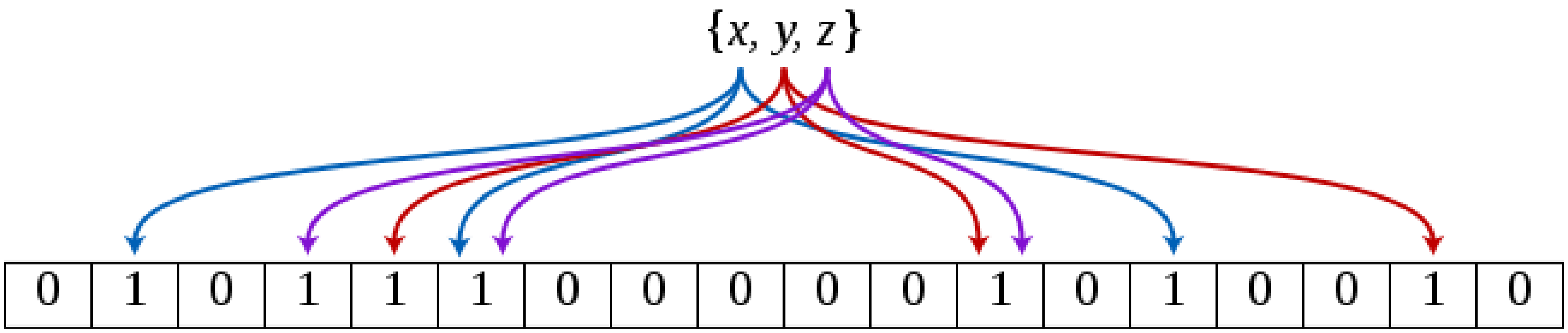
Aside: Bloom Filter

- Create a bit array of length m (init to 0's)
- Create k hash functions that map an object to an index of m
- Index x: set $m[\text{hash}_1(x)], \ldots, m[\text{hash}_k(x)]$ to 1



Aside: Bloom Filter

- Create a bit array of length m (init to 0's)
- Create k hash functions that map an object to an index of m
- Index x: set $m[\text{hash}_1(x)]$, …, $m[\text{hash}_k(x)]$ to 1

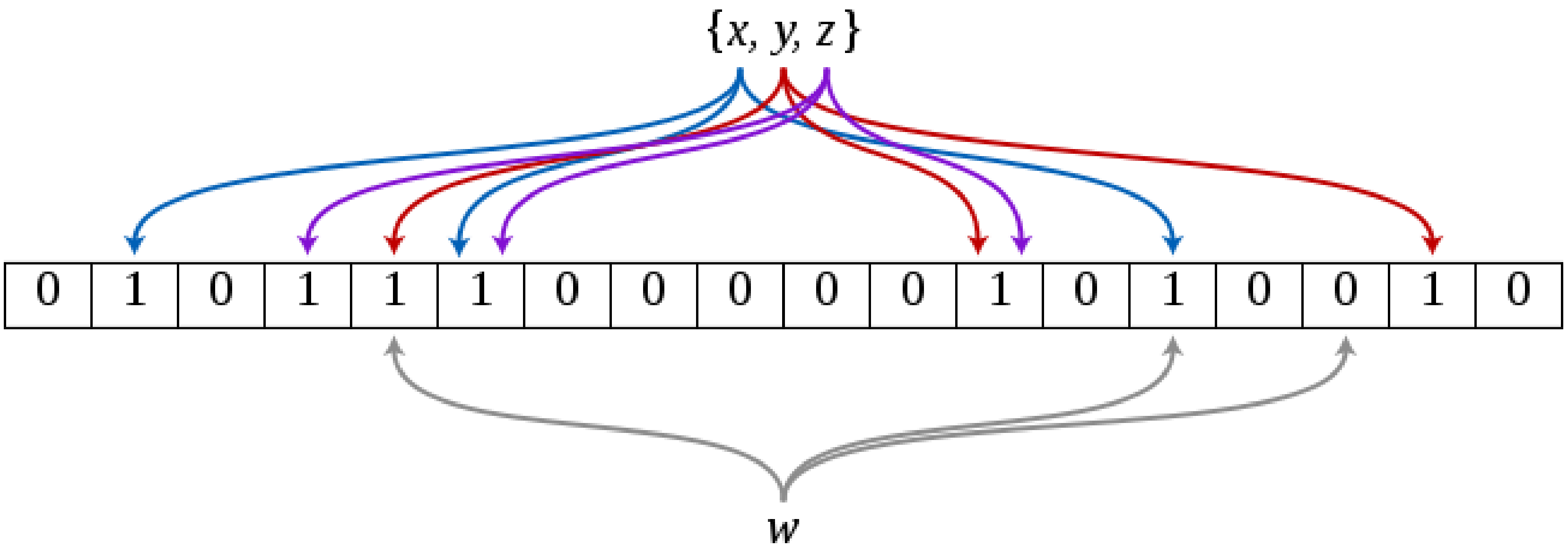


- Query w:
  - any $m[\text{hash}_1(w)]$, …, $m[\text{hash}_k(w)]$ set to **0** $\Rightarrow$
  - all $m[\text{hash}_1(w)]$, …, $m[\text{hash}_k(w)]$ set to **1** $\Rightarrow$

Aside: Bloom Filter

Reject "empty" queries using very little memory!

- Create a bit array of length m (init to 0's)
- Create k hash functions that map an object to an index of m
- Index x: set $m[\text{hash}_1(x)]$, …, $m[\text{hash}_k(x)]$ to 1



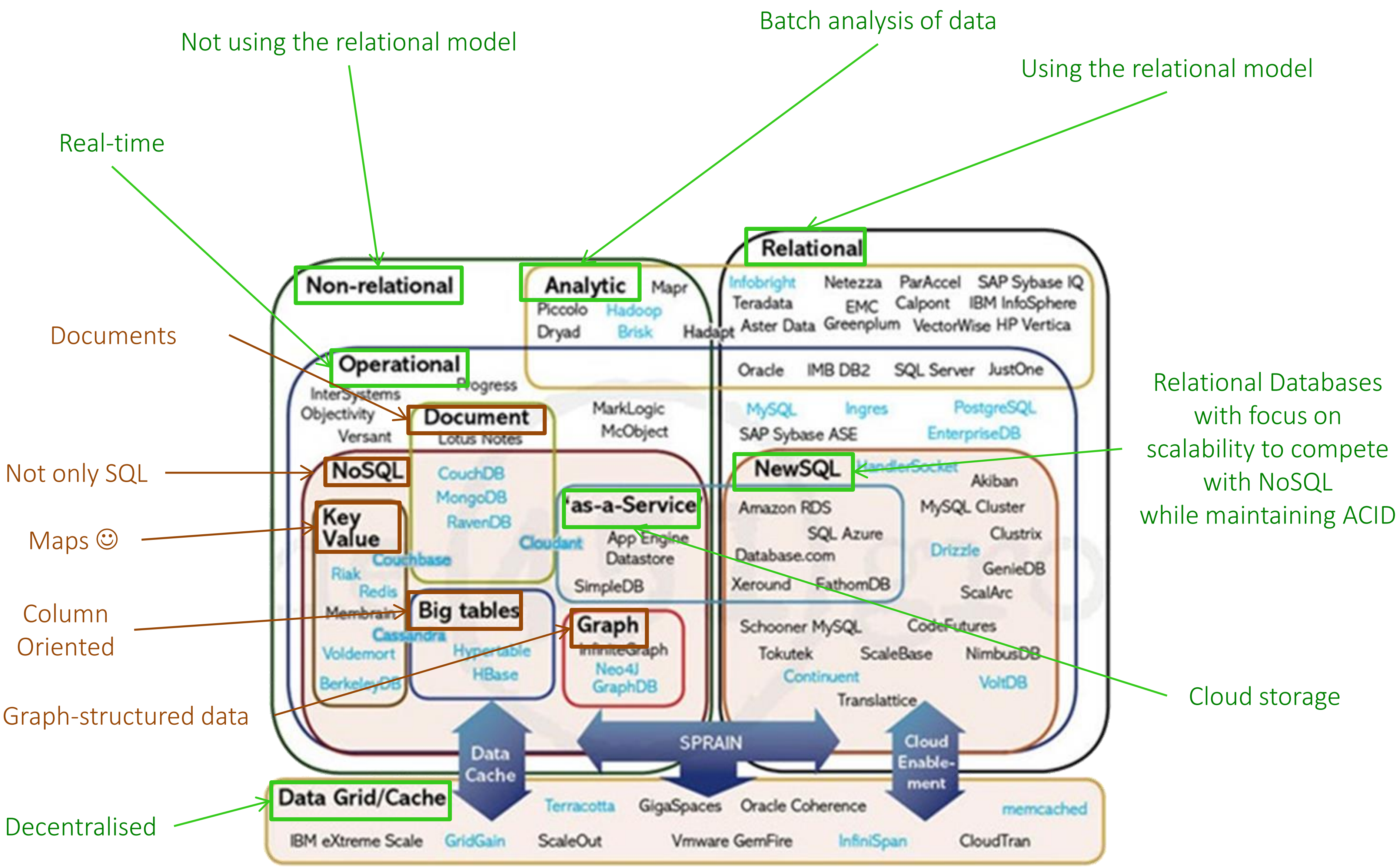
- Query w:
  - any $m[\text{hash}_1(w)]$, …, $m[\text{hash}_k(w)]$ set to **0** ⇒ not indexed
  - all $m[\text{hash}_1(w)]$, …, $m[\text{hash}_k(w)]$ set to **1** ⇒ **might** be indexed

Tabular Store: Apache HBase

Tabular Store: Cassandra

cassandra

Database Landscape



PROJECTS

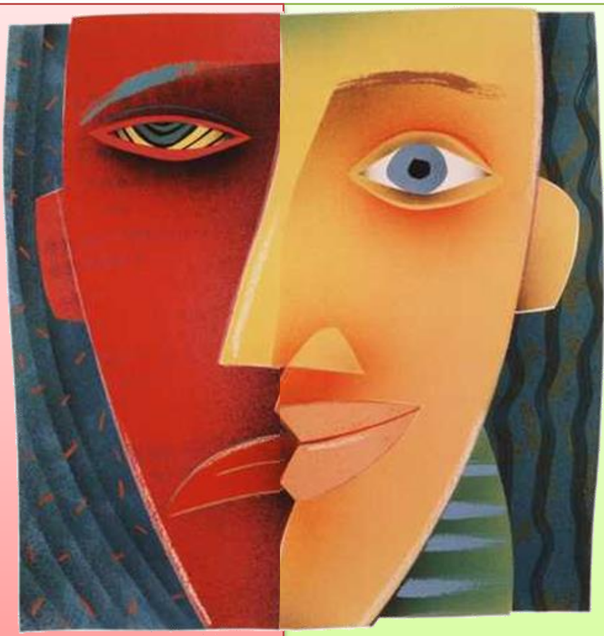
Course Marking

- 55% for Weekly Labs (~5% a lab!)
- 15% for Class Project
- 30% for 2x Controls

Assignments each week

Controls

Working in groups



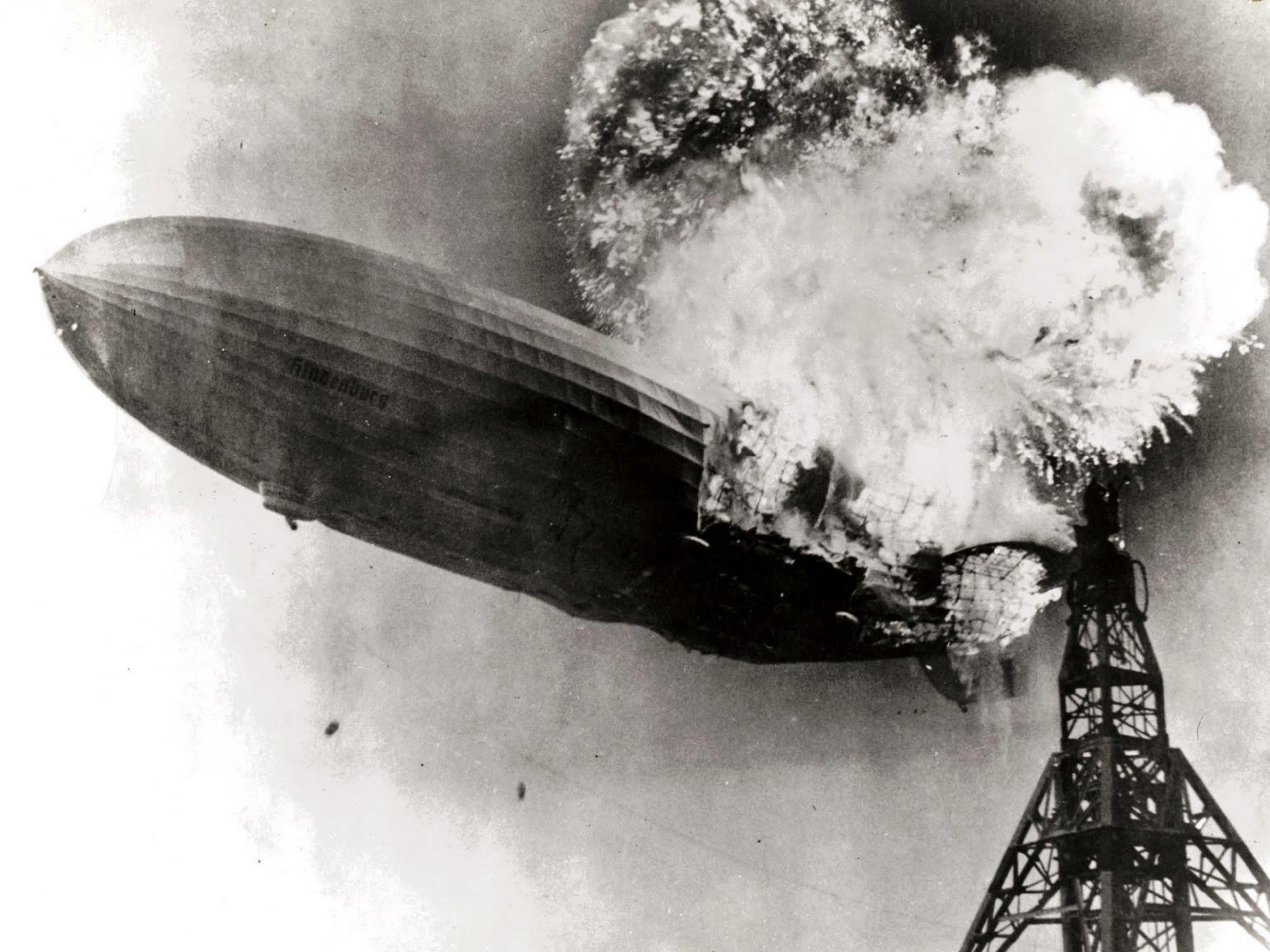
Only need to pass overall!

No final exam!

Working in groups!

Class Project

- Done in threes
- Goal: Use what you've learned to do something cool/fun (hopefully)
- Expected difficulty: A bit more than a lab's worth
  - But without guidance (can extend lab code)
- Marked on: Difficulty, appropriateness, scale, good use of techniques, presentation, coolness, creativity, value
  - Ambition is appreciated, even if you don't succeed
- Process:
  - Start thinking up topics / find interesting datasets!
- Deliverables: 4 minute presentation & short report



Questions?