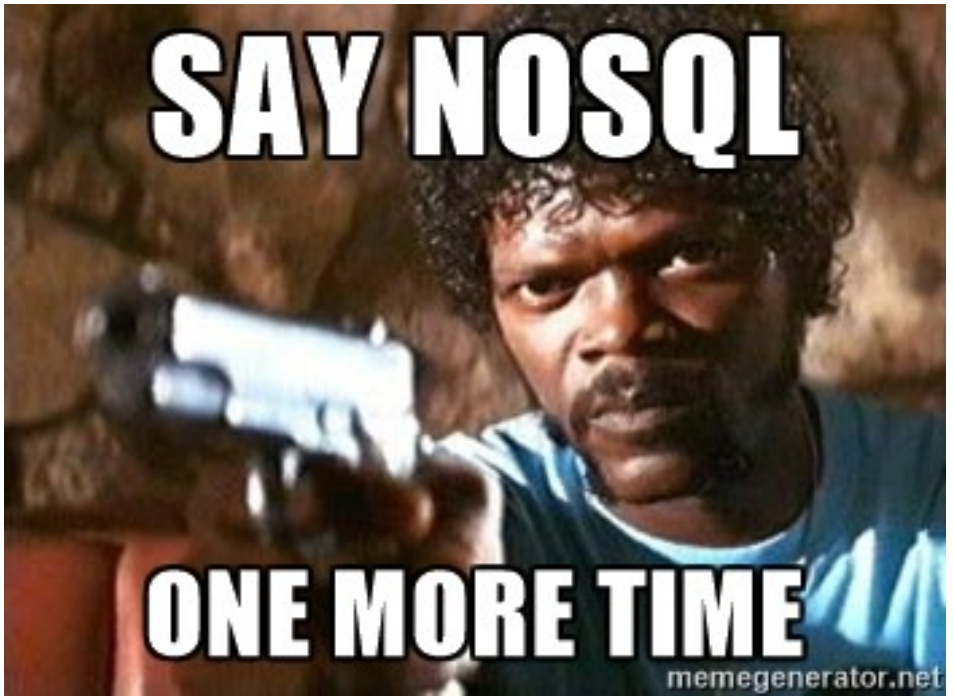
# El movimiento noSQL




**Bases de Datos No Relacionales**

Instituto de Computación, FING, UdelaR – 2021

CC-BY Lorena Etcheverry lorenae@fing.edu.uy

# ¿Por qué surge el movimiento noSQL?

# Arquitectura de las aplicaciones en la web 2.0

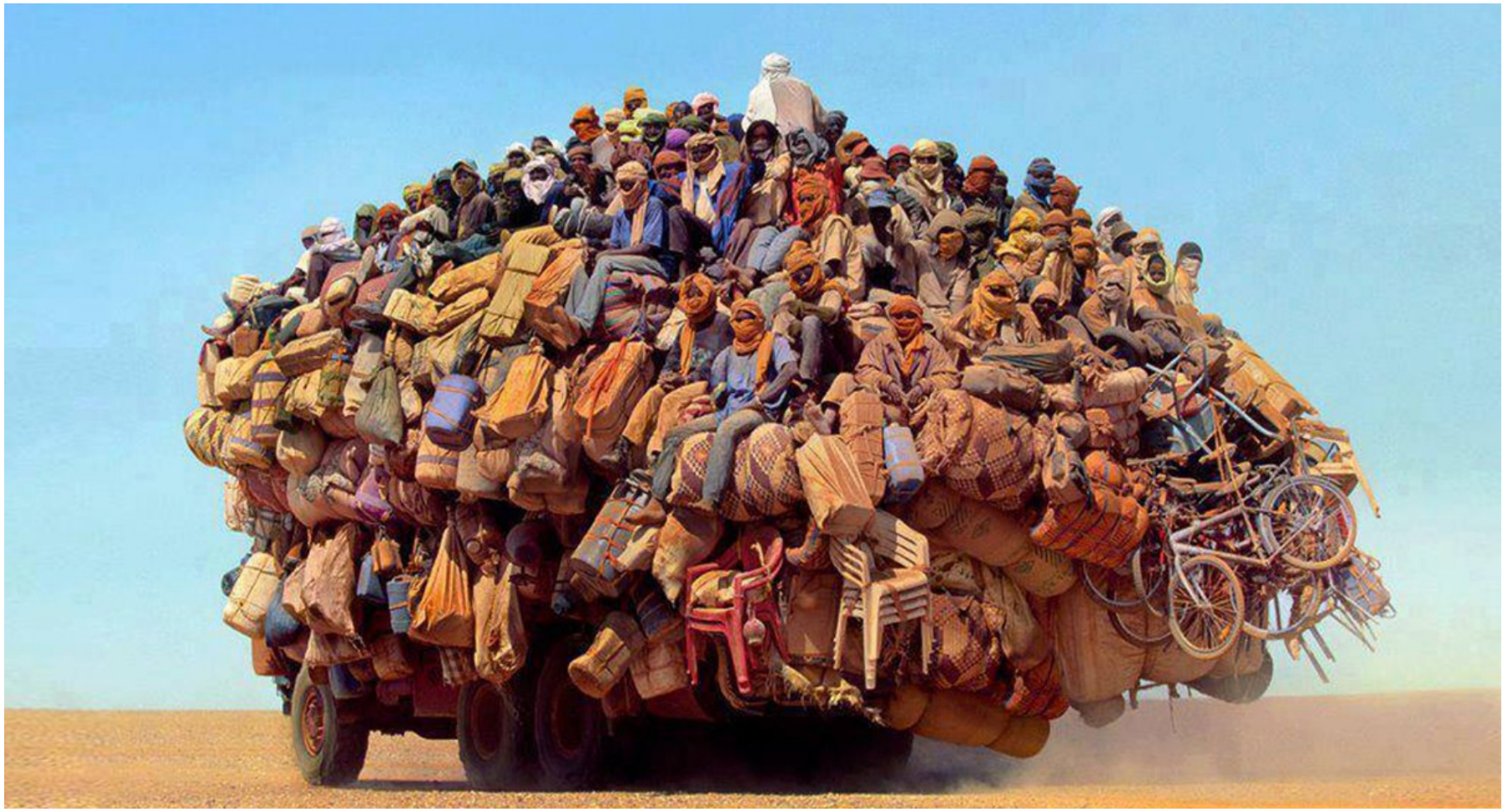
Aplicaciones cliente-servidor

Páginas dinámicas

Implementación:

en la década del 2000 era muy popular

el stack LAMP



# y cuando aparecen problemas de escala..

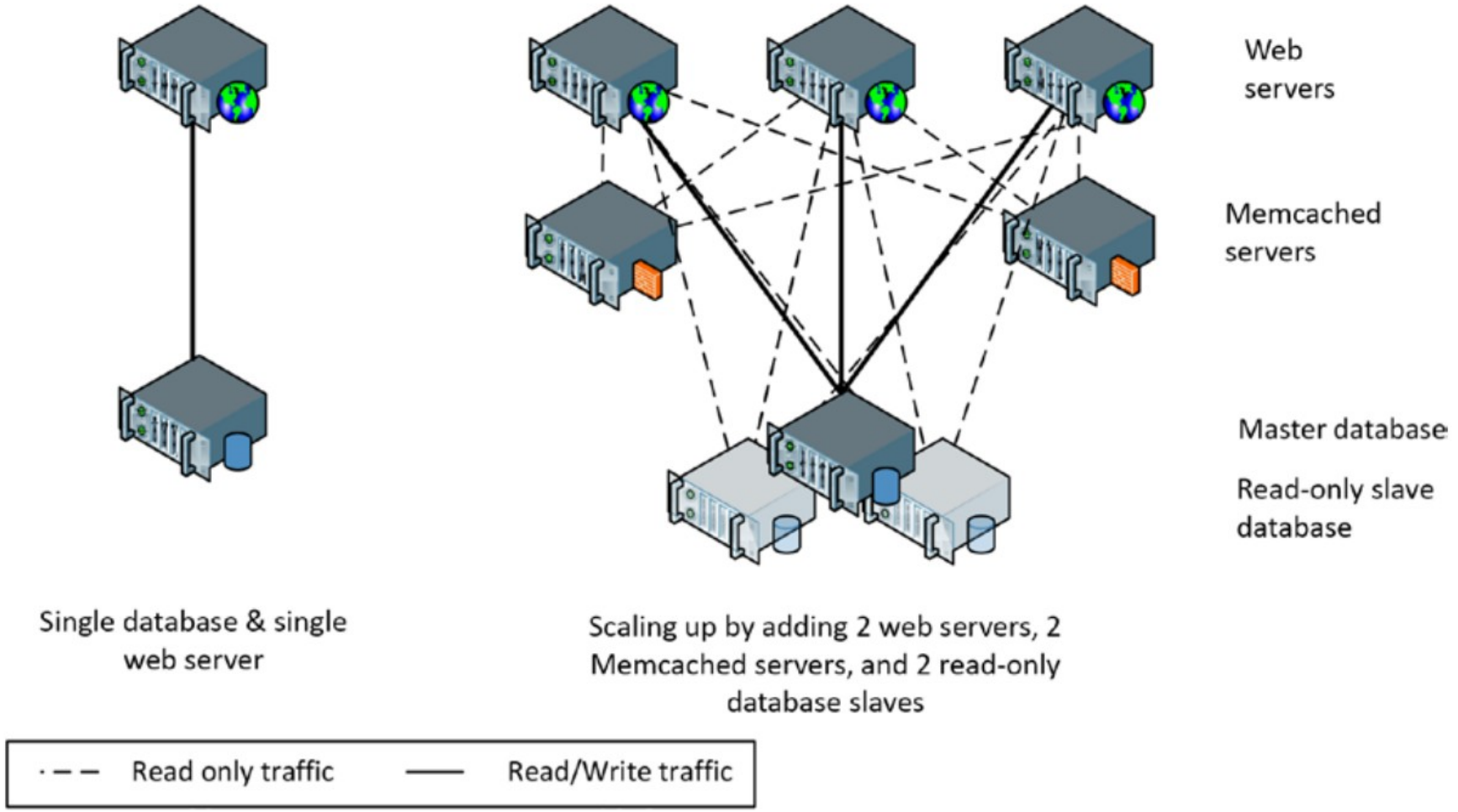



***Figure 3-1.*** *Scaling up with Memcached servers and replication*

Manteniendo réplicas de la base de datos en diferentes servidores mejoro la performance de **lectura**.

¿cómo mejoro la performance de **escritura**?

# ¿y si particionamos los datos?

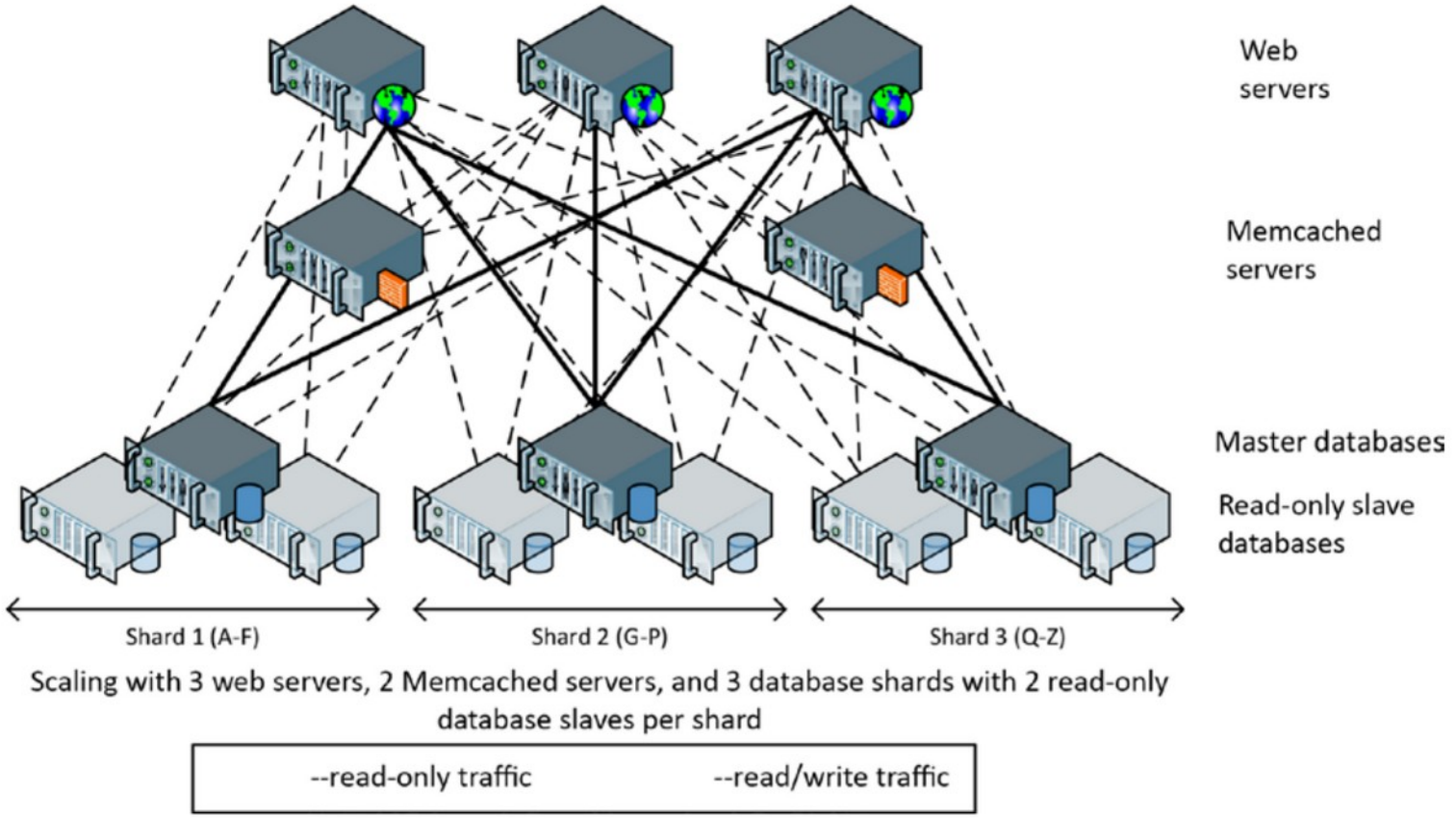



***Figure 3-2.*** *Memcached/replication architecture from Figure 3-1, with sharding added*

Particionamiento *(sharding)* horizontal de los datos según la clave.

# Un ejemplo de uso: Facebook (circa 2011)

- más de 4,000 shards MySQL
- más de 9,000 servidores Memcached
- 1.4k millones de lecturas por segundo
- 3.5 millones de cambios en filas por segundo
- 8.1 millones de operaciones de acceso a disco por segundo


¿pero cómo se gestiona esto?

# Algunas desventajas del uso de *sharding*

- Complejidad de la aplicación
  - Particionamiento elástico!
- SQL mutilado :)
  - ¿cómo hago consultas que involucren más de una partición?
- Pérdida de la integridad transaccional
  - Las operaciones que afectan más de una partición dejan de cumplir ACID
- Complejidad operacional
  - ¿balance de carga? ¿cambios en el esquema?

# el teorema CAP (Brewer,2000)

"En un sistema de bases de datos distribuído puedo tener a lo sumo dos de las siguientes propiedades:

*Consistency*, *Availability* y *Partition tolerance*"

**CONSISTENCY:** todos los usuarios de la base de datos ven los mismos datos en cierto instante

**AVAILABILITY:** ante una falla, la base de datos sigue operacional

**PARTITION TOLERANCE:** el sistema sigue siendo operacional ante una falla de la red que comunica los segmentos del sistema distribuido



Notas sobre la demostración: http://julianbrowne.com/article/viewer/brewers-cap-theorem

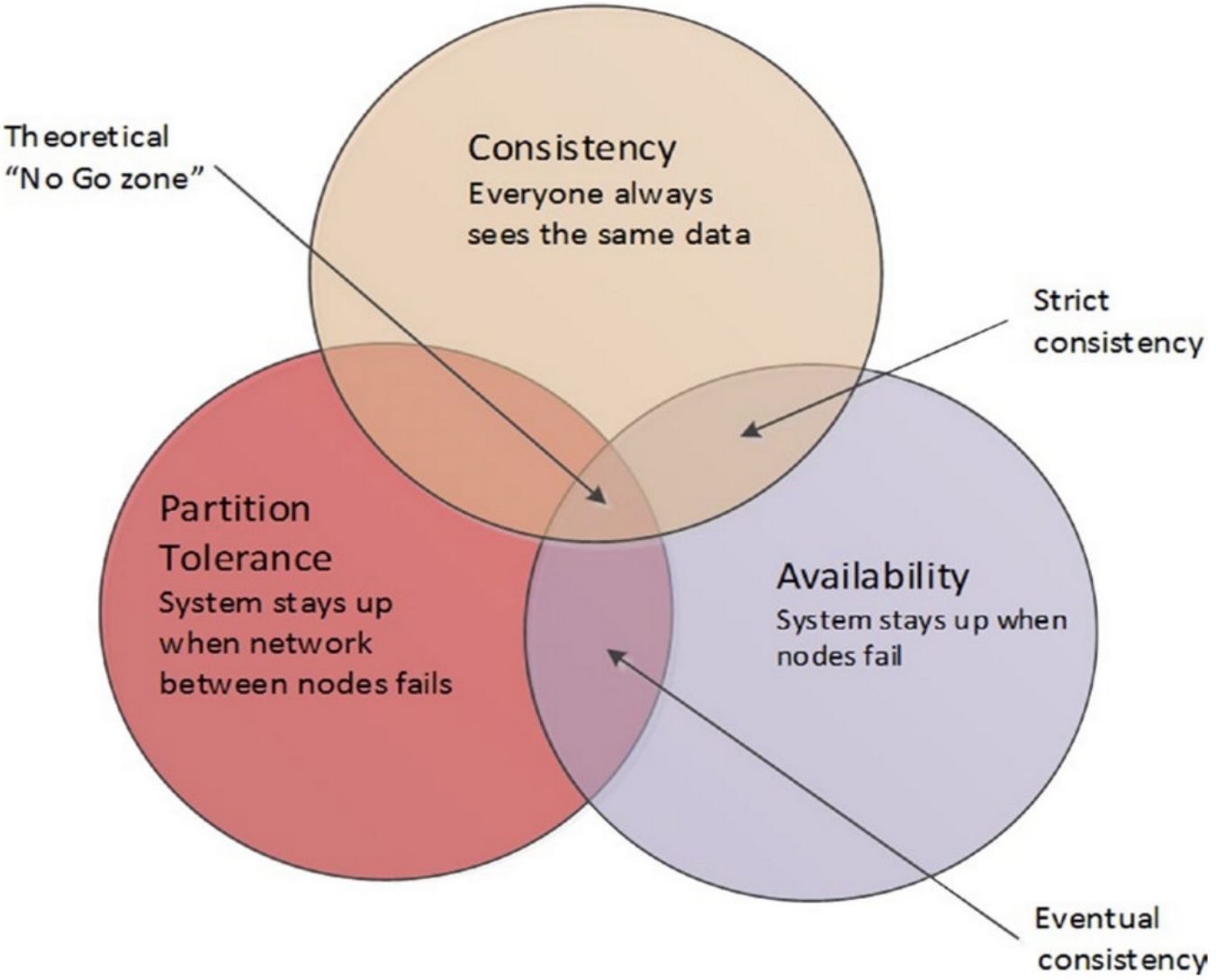
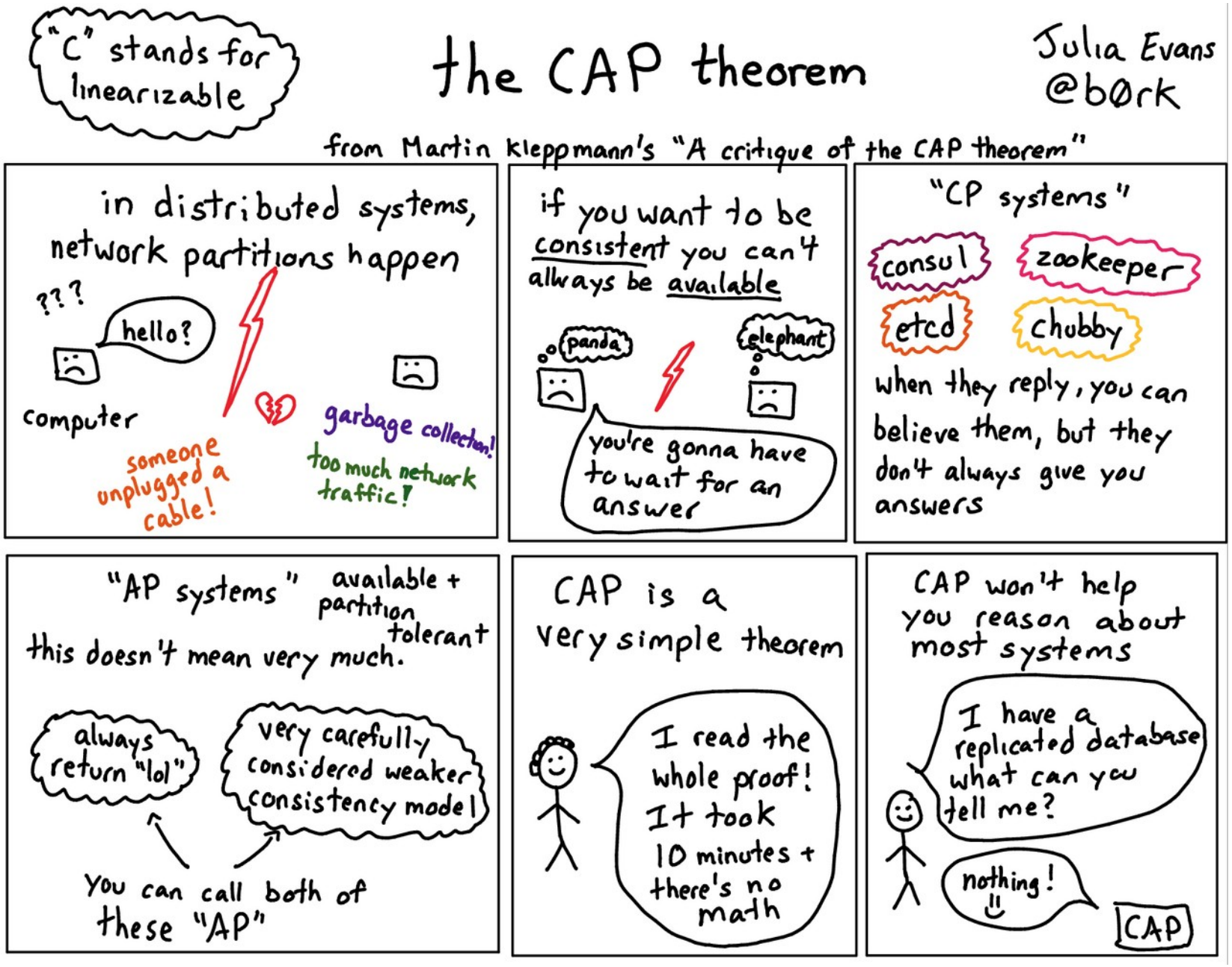



***Figure 3-4.*** *Dynamo and ACID RDBMS mapped to CAP theorem limits*






https://jvns.ca/blog/2016/11/19/a-critique-of-the-cap-theorem/

# ACIDos vs BASEs

- Más allá del teorema CAP implementar transacciones ACID en un sistema distribuído es costoso en difentes sentidos
  - Disponibilidad implica réplicas (típicamente distribuídas físicamente)
  - Para asegurar consistencia **estricta** cada transacción implica propagar cambios en forma síncrona!

**BASICALLY AVAILABLE, SOFT-STATE, EVENTUALLY CONSISTENT**

# Evolución de la discusión sobre CAP

- Keynote de Eric Brewer en PODC (2000)
  https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
- Demostración del teorema CAP (Gilbert and Lynch, 2002)
  https://www.comp.nus.edu.sg/~gilbert/pubs/BrewersConjecture-SigAct.pdf
- Problems with CAP, and Yahoo's little known NoSQL system, Daniel Abadi (2010)
  http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html
- CAP Twelve Years Later: How the "Rules" Have Changed, Eric Brewer (2012)
  https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/
- A Critique of the CAP Theorem, Martin Kleppmann (2015) https://arxiv.org/abs/1509.05393

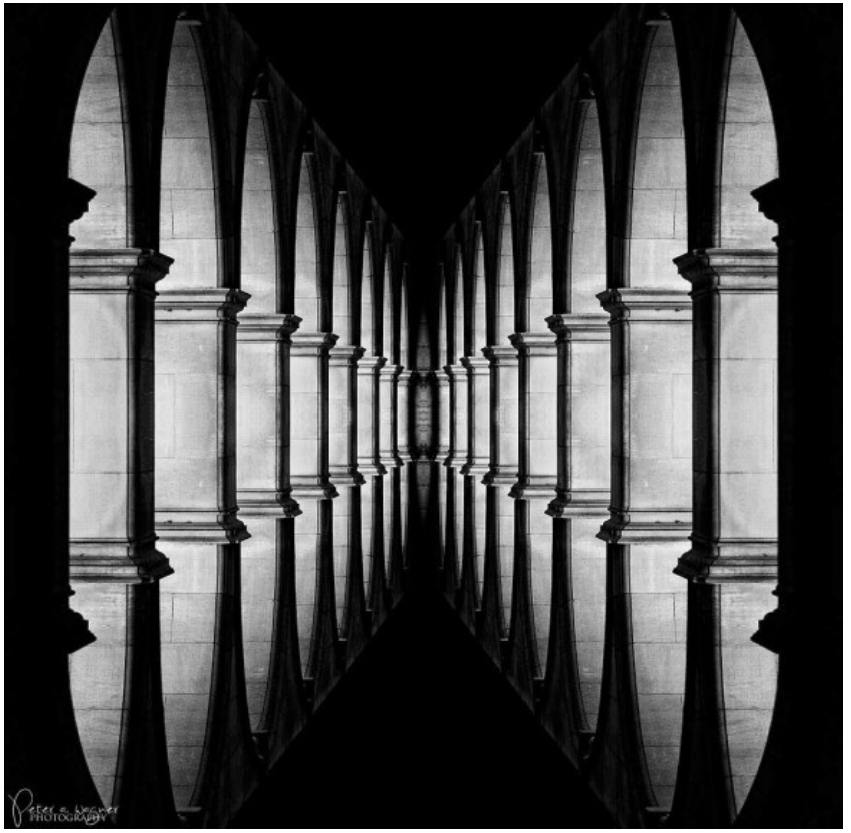
# Key-value y Column-family stores

- Emulan la funcionalidad de los *shards* relacionales
  - Múchos nodos
  - Datos distribuidos por clave
  - Datos accesibles por clave (similar a BigTable

- Relajan ACID (la solución con *shards* tmb)
  - Definen diferentes niveles de consistencia

# Column-family stores:

## Amazon Dynamo y Apache Cassandra como ejemplos



**Bases de Datos No Relacionales**

Instituto de Computación, FING, UdelaR – 2021

# Agenda

- ¿Qué problema se pretende resolver?
- Particionamiento de datos
  - Consistent hashing
- Modelos de consistencia variable
- Resolución de conflictos

# El problema original [1]

**QUE**

- Almacenar parejas (clave,valor)
- Ofrecer operaciones get() y put()
- Update del valor por completo
  - Las column family stores relajan esto luego

**COMO**

- Garantizar alta disponibilidad
- Tolerancia a particiones de la red
- Diferentes niveles de consistencia
- Escalar horizontalmente (más máquinas)
- *Always write*

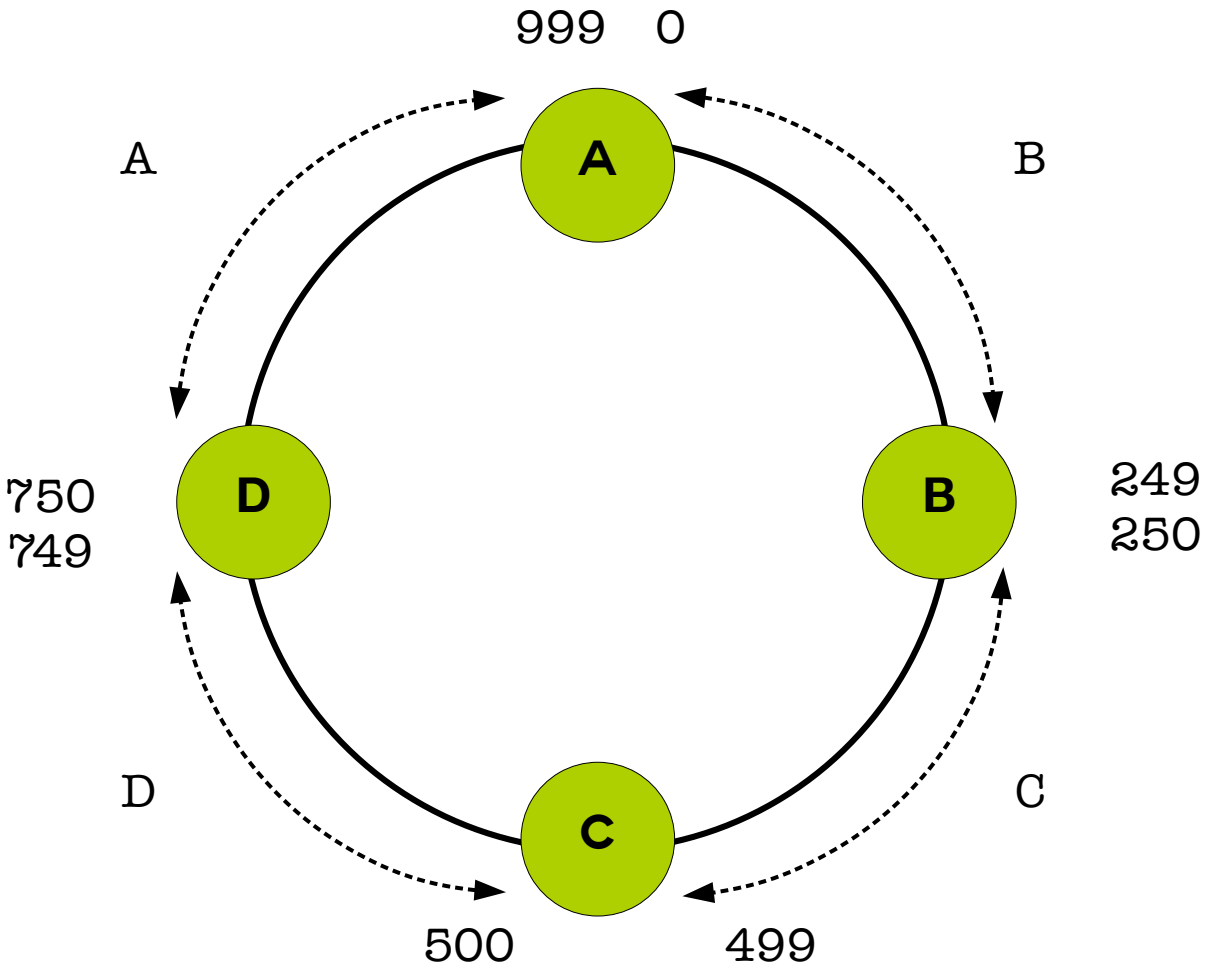
[1] Dynamo: Amazon's Highly Available Key-value Store, DeCandia et al 2007

# Arquitectura: aspectos claves

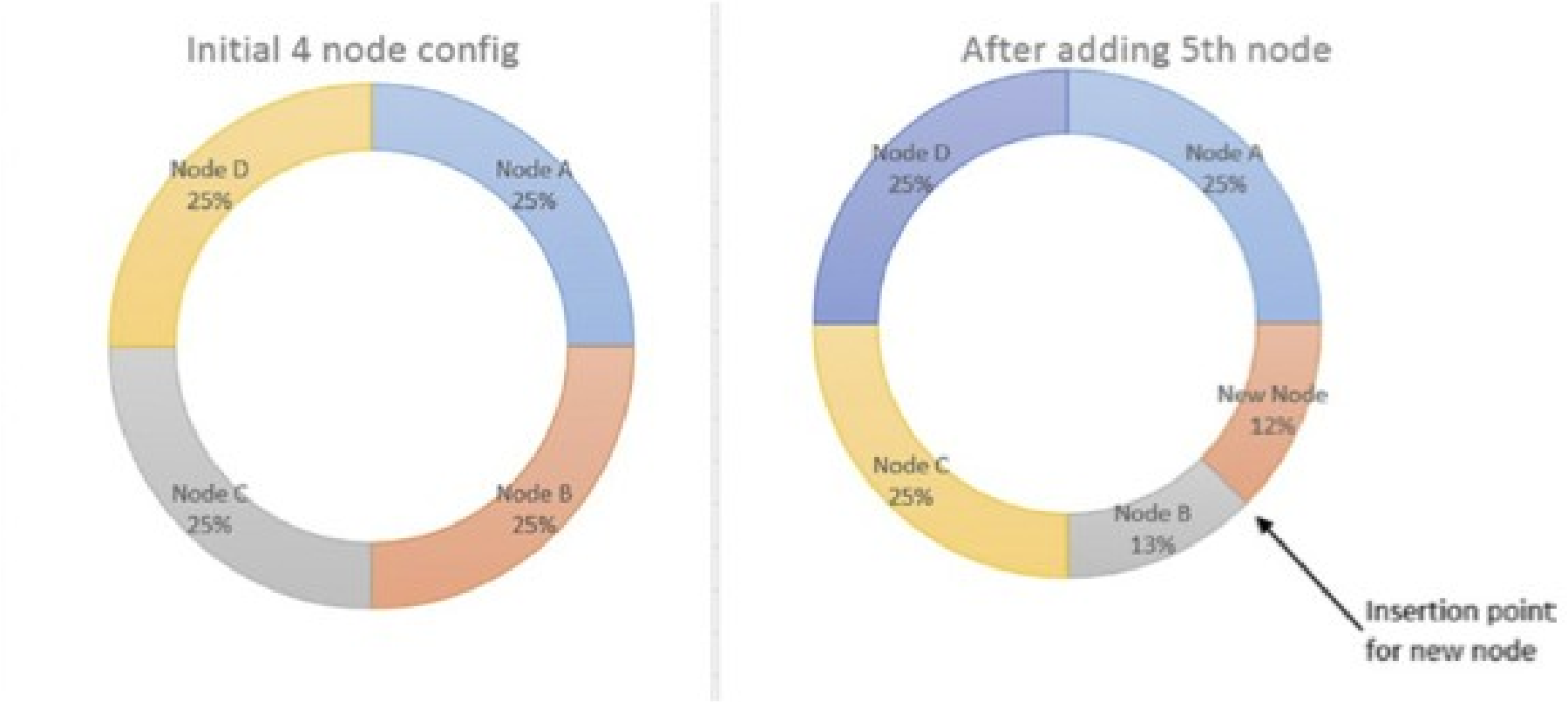
- Todos los nodos tienen iguales responsabilidades (no master)
- Particionamiento basado en hashing de las claves
  - Distributed Hash Table (DHT)
  - Cantidad variable de nodos: *consistent hashing*
- Nivel de consistencia ajustable
- Versionado de datos
  - Resolución de conflictos “on read”

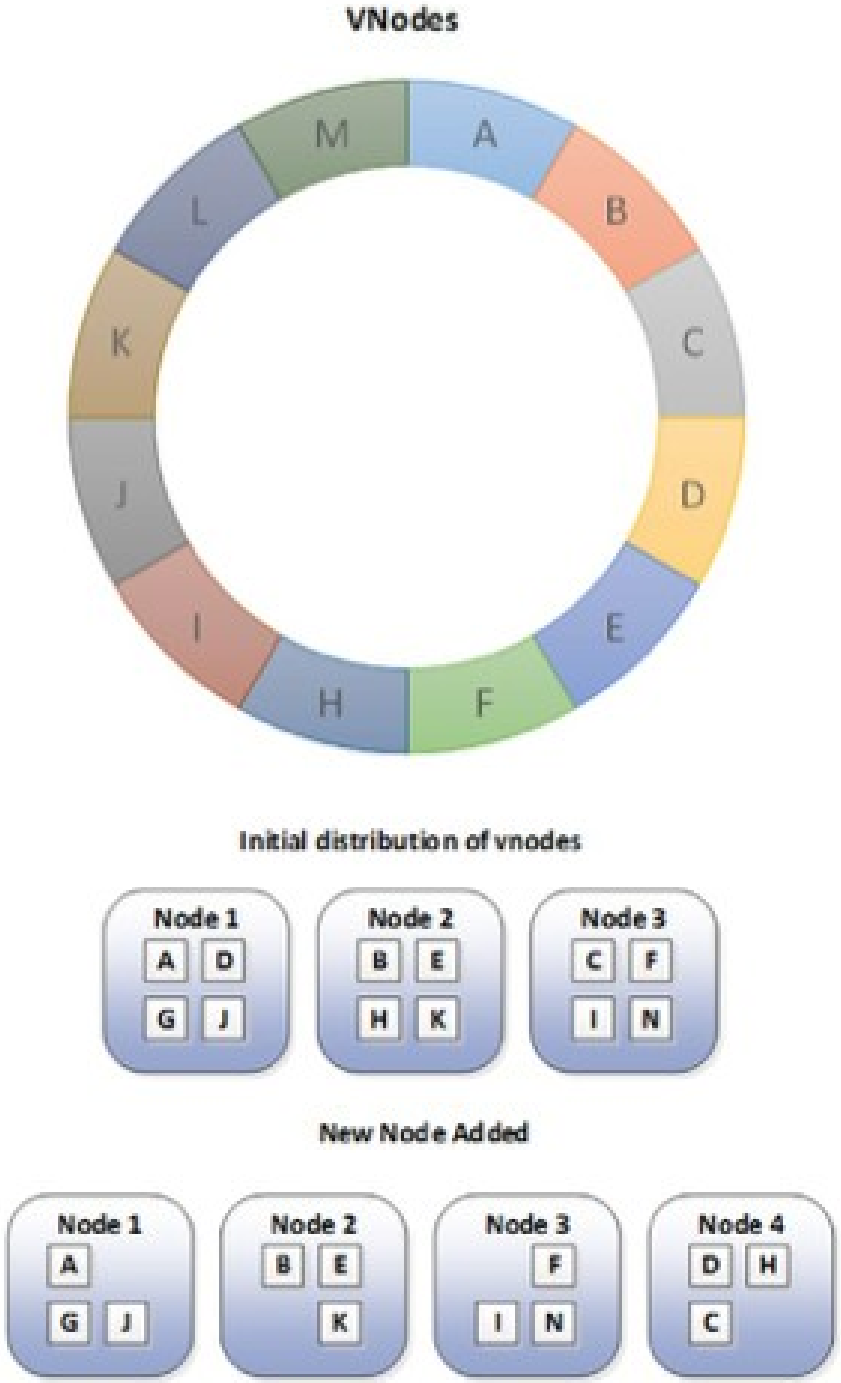
# Consistent hashing

- Se aplica una función de hash a la clave
  - Ej: MD5 y obtiene un valor 128 bits
- Cada nodo maneja un rango de valores
- Cada nodo sabe:
  - Qué nodos están en el sistema (topología de anillo)
  - Qué rango de claves maneja cada nodo

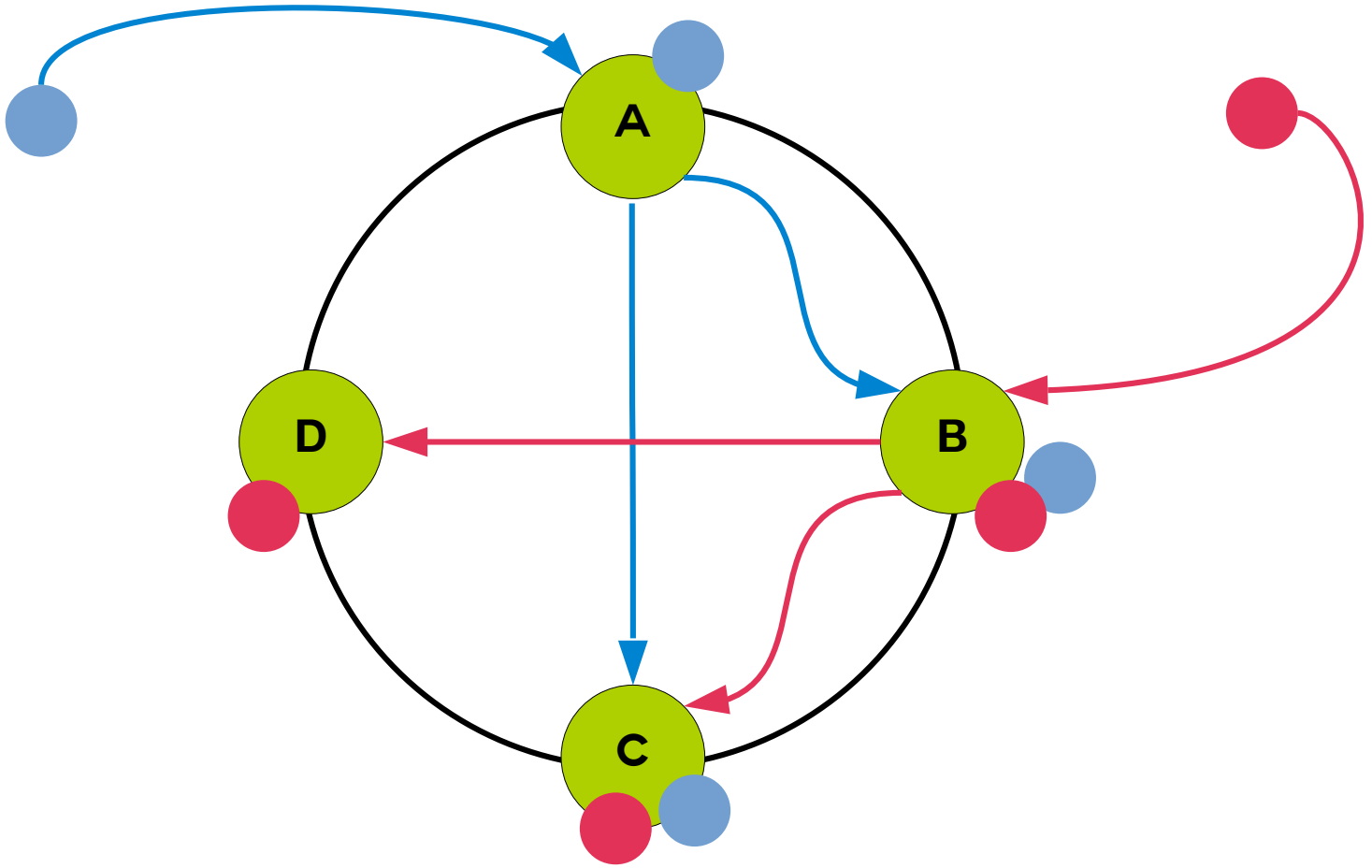



Agregar nuevos nodos implica redefinir los rangos, transmitir esta info, y copiar datos.


Initial 4 node config
Node A
25%
Node B
25%
Node C
25%
Node D
25%
After adding 5th node
Node A
25%
New Node
12%
Node B
13%
Node C
25%
Node D
25%
Insertion point
for new node



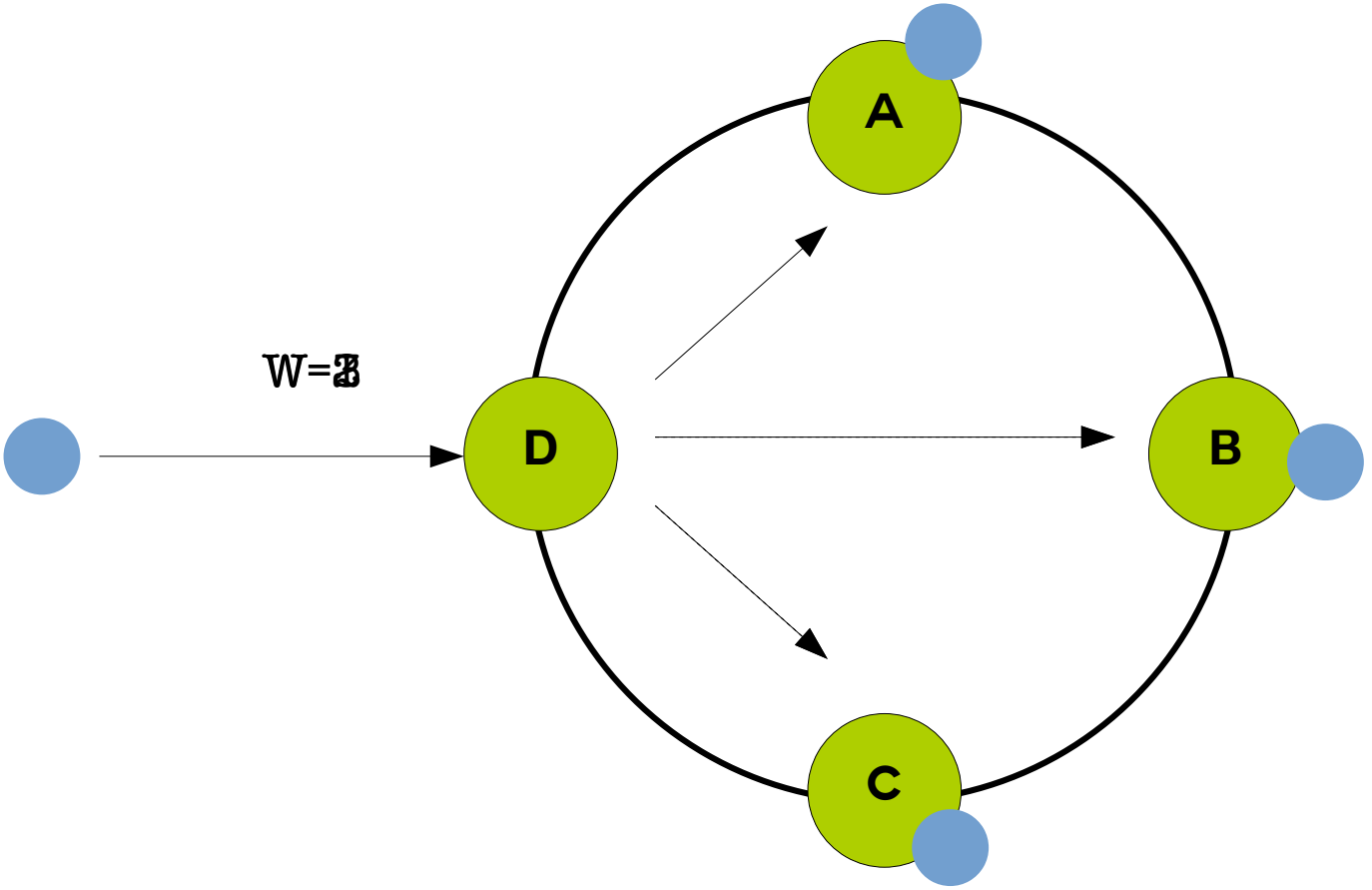

*Virtual nodes* para reducir el costo de mover datos físicamente

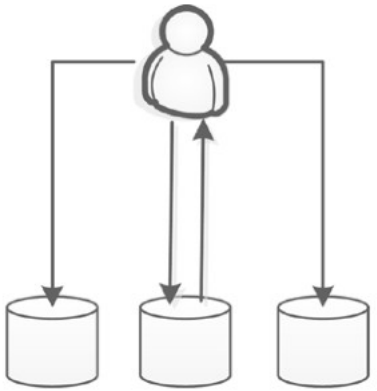



La cantidad de réplicas y su ubicación es ajustable.

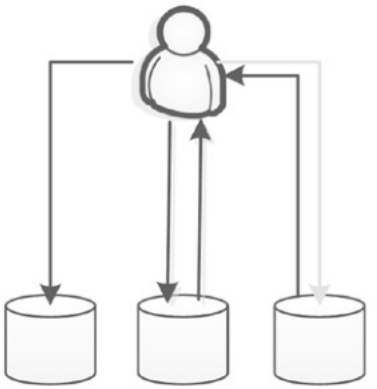
# Consistencia ajustable

- Notación NWR
  - N: cantidad de copias a mantener (réplicas)
  - W: cantidad de copias a escribir para completar un *write*
  - R: cantidad de copias a leer en una operación de *read*
- Variando estos valores obtenemos diferentes niveles de consistencia

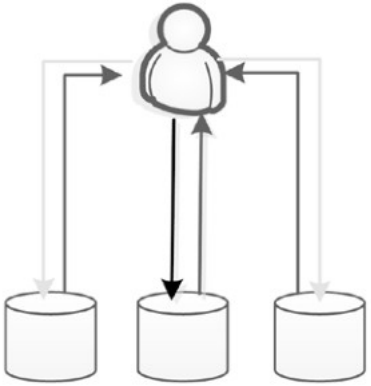

A
W=3
D
B
C



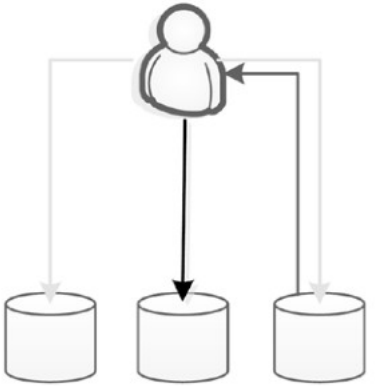
N=3 W=3 R=1
Escritura lenta, lectura rápida
CONSISTENTE



N=3 W=2 R=2
Escritura más rápida, lectura requiere resolver cual es el valor más reciente
CONSISTENTE



N=3 W=1 R=3
Escritura rápida (puede perderse datos),
Lectura lenta y resolver el valor más reciente
CONSISTENTE



N=3 W=1 R=1
Rápido, pero no consistente (pueden haber diferentes versiones y no me entero)

# Resolución de conflictos

- Resolución de conflictos al **leer**
  - Los RDBMS resuelven conflictos al **escribir** via locks
- Mecanismos:
  - *"Last Write Wins"* (timestamps)
  - Vector clocks

# Last Write Wins

Cliente A


Cliente B


Cliente C


Se sobreescribe todo el valor.

En column family stores sobreescribo cada columna

Puedo perder datos!!!

# Vector Clocks

A cada dato se le adiciona un vector de parejas (nodo, contador), donde el contador indica la cantidad de veces que ese nodo escribió ese dato.

Esto permite determinar cuando dos versiones de un dato pueden potencialmente presentar conflictos.

# Vector Clocks (ejemplo)

Cliente A

{A:1} key:user22 | name:Ann | age:24 | mail:ann@mail.com

T:1

Cliente B

{A:1,B:1} key:user22 | name:Ann | age:26 | mail:ann@mail.com

T:2

Cliente C

{A:1,C:1} key:user22 | name:Ann | age:24 | mail:ann@new.com

T:3

Hay un conflicto entre los últimos dos registros

Diferentes estrategias para resolverlo

# Resumen

- Amazon Dynamo fue la primer key-value store
- Persistencia eventualmente consistente de parejas clave valor, escalable y tolerante a fallas en la red
- Sentó las bases para varios productos:
  - Apache Cassandra, Riak, DynamoDB

# Referencias adicionales

- Dynamo: Amazon's Highly Available Key-value Store, DeCandia et al 2007
  - www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- DynamoDB - The Paper That Changed The Database World
  - https://vimeo.com/144994937
- Deep Dive: Amazon DynamoDB
  - https://www.youtube.com/watch?v=VuKu23oZp9Q