

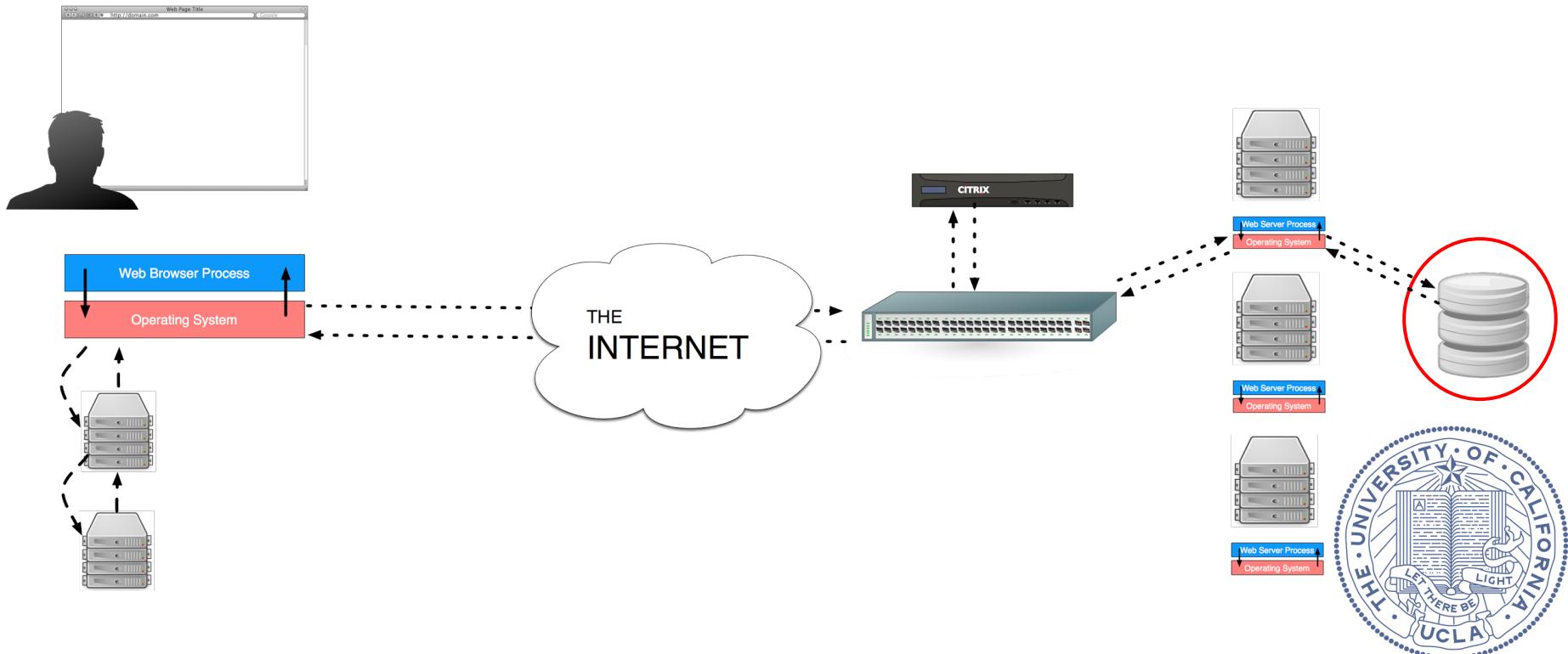
# CS 188/219

## Scalable Internet Services

Andrew Mutz  
May 6, 2015



# For Today



# Motivation

After today's lecture you will understand how NoSQL can be used to build into scalable internet services.

NoSQL data stores won't likely be part of your project, but after this lecture you should understand when you could look to them in practice.



# Motivation



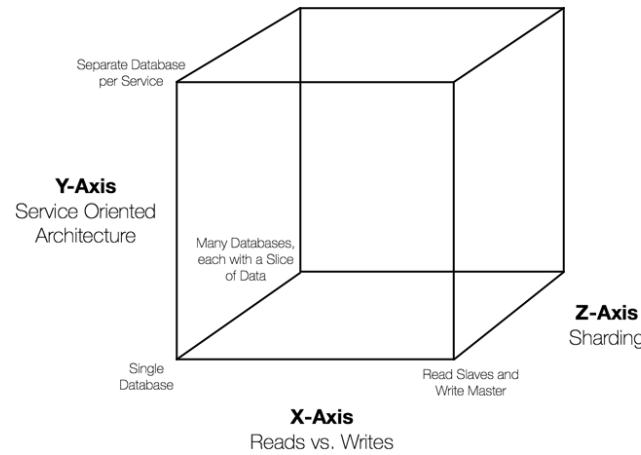
As our application has experienced greater and greater popularity, the data layer has proven difficult to scale horizontally.

Without a scaling path for our data layer, it will be a bottleneck limiting our application.



# Motivation

Relational Databases are great tools for our data layer, but we can't simply spread load across multiple RDBMSes.



We've looked at a few techniques for scaling RDBMSes:

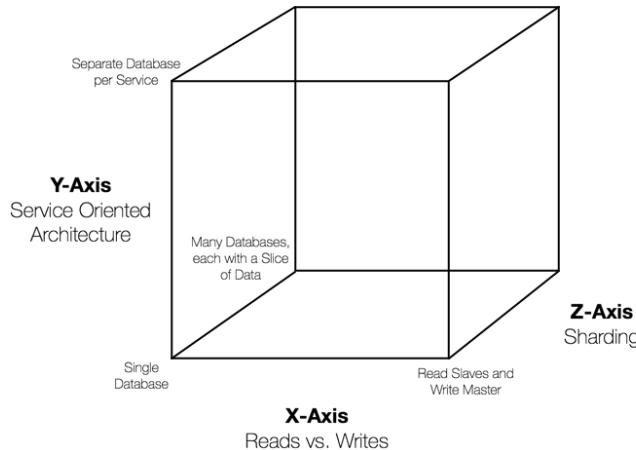
- Sharding
- Service Oriented Architectures
- Distinguishing Reads from Writes



# Motivation

What if these techniques aren't sufficient for our target application?

- There's no good way to shard our application?
- We've already broken our application out via SOA and still have load hotspots?
- We're already using read-slaves and it's not enough?



When relational databases fail to scale to our needs, we need to turn to non-relational solutions.



# NoSQL



Non-relational databases are sometimes called NoSQL databases.

This is an umbrella term for many types of databases

- Key-value stores
- Column-oriented data stores
- Document-oriented stores
- Graph databases



# NoSQL



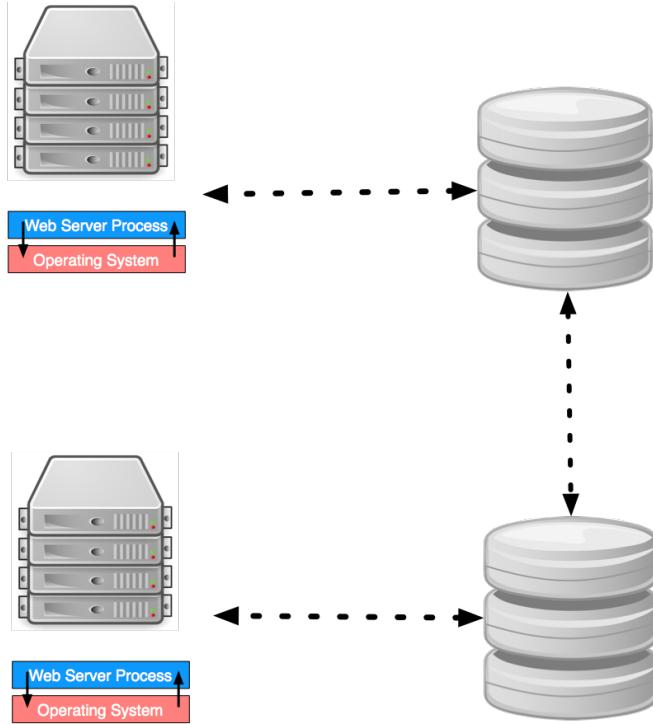
Most NoSQL solutions are good at horizontal scaling.

- You can easily add hardware to the database to increase throughput.

In exchange for better horizontal scaling, these databases provide the application layer fewer guarantees



# NoSQL

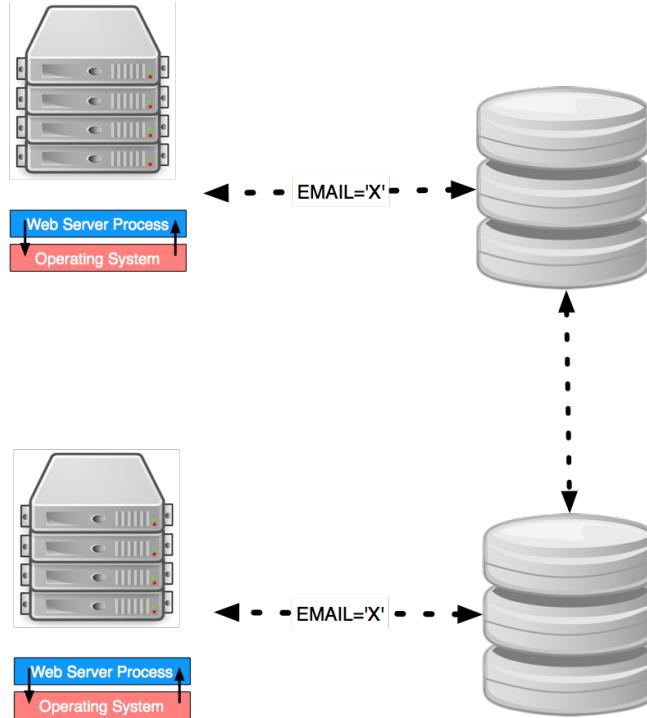


Let's say we want a database to span multiple machines.

We can update on both nodes, and the databases keep each other in synch.



# NoSQL



Lets say:

- We have two clients sending writes.
- There's a uniqueness constraint on email.

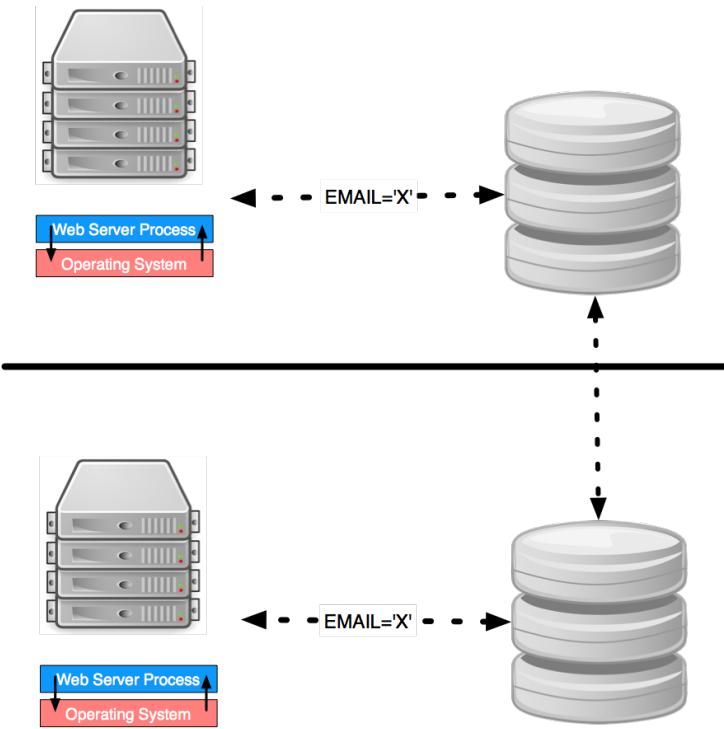
If both try to write the same email address to different columns, and the databases can communicate, they can resolve this in some manner.

- Ex: Allow one, fail the other.

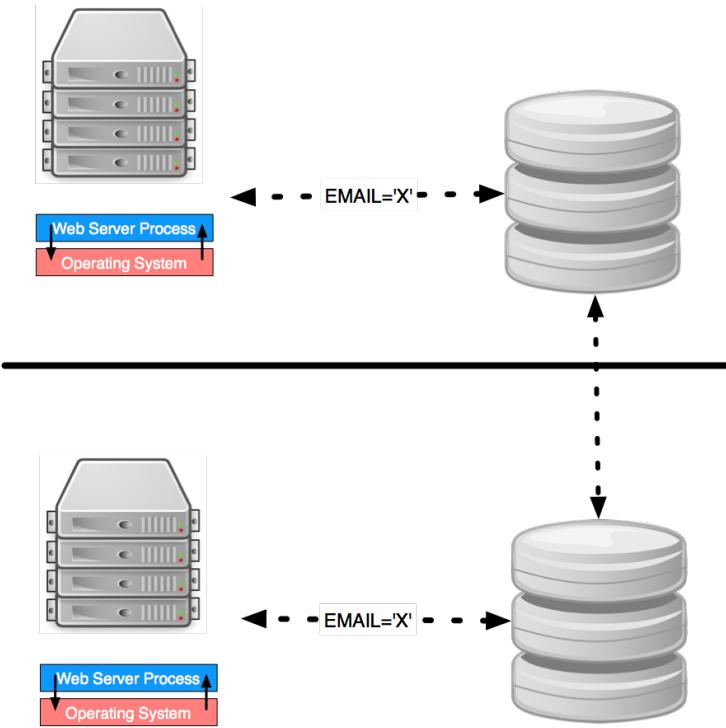


# NoSQL

How do we handle a network partition?



# NoSQL

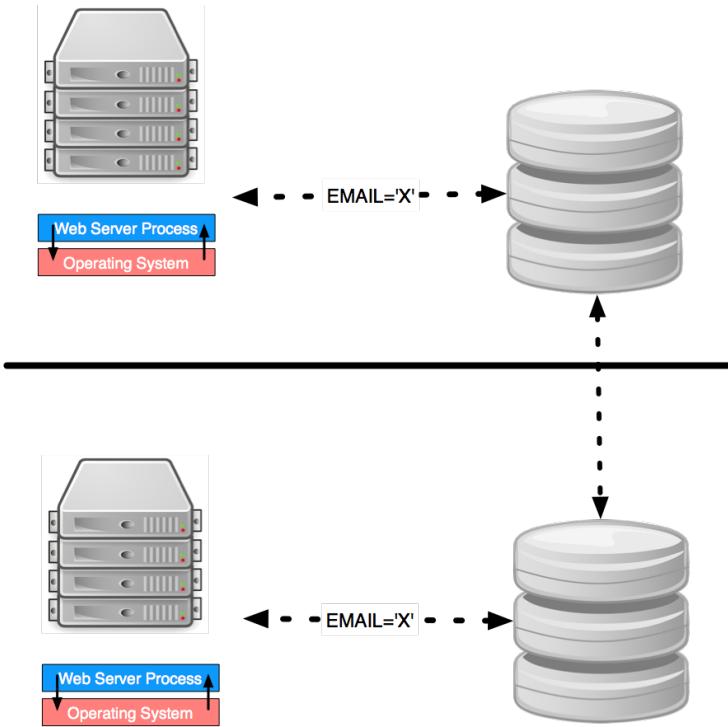


## How do we handle a network partition?

- If the databases can't communicate, they don't know if this update violates database consistency.
- Allow the write and hope for the best?
  - And fix it later if needed?
- Not accept such writes during a partition?



# NoSQL



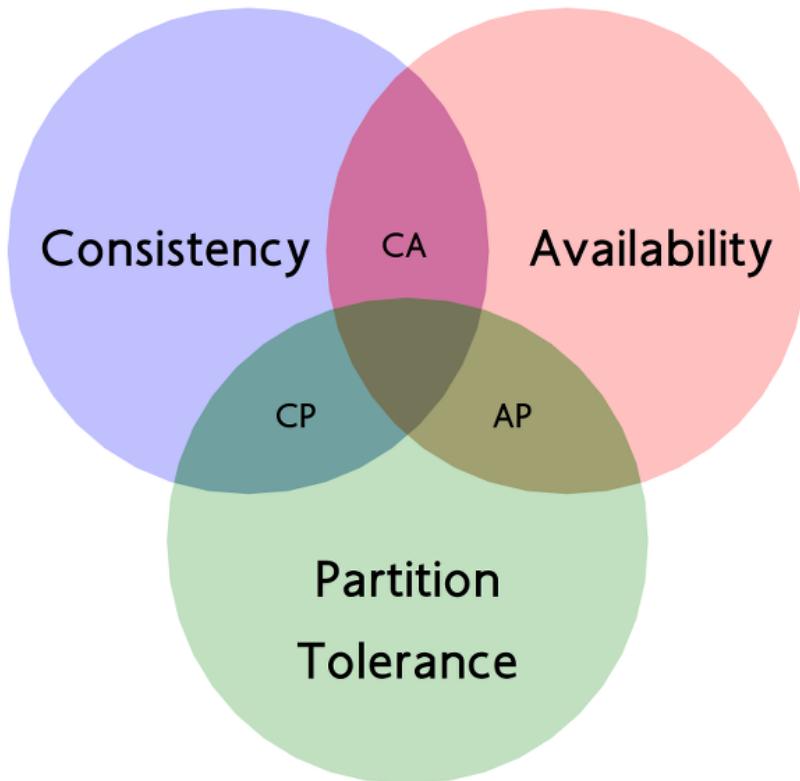
If we allow the write, our database is not consistent.

If we don't allow the write, our database is unavailable.

If we ignore this scenario, we can't tolerate network partitions.



# NoSQL

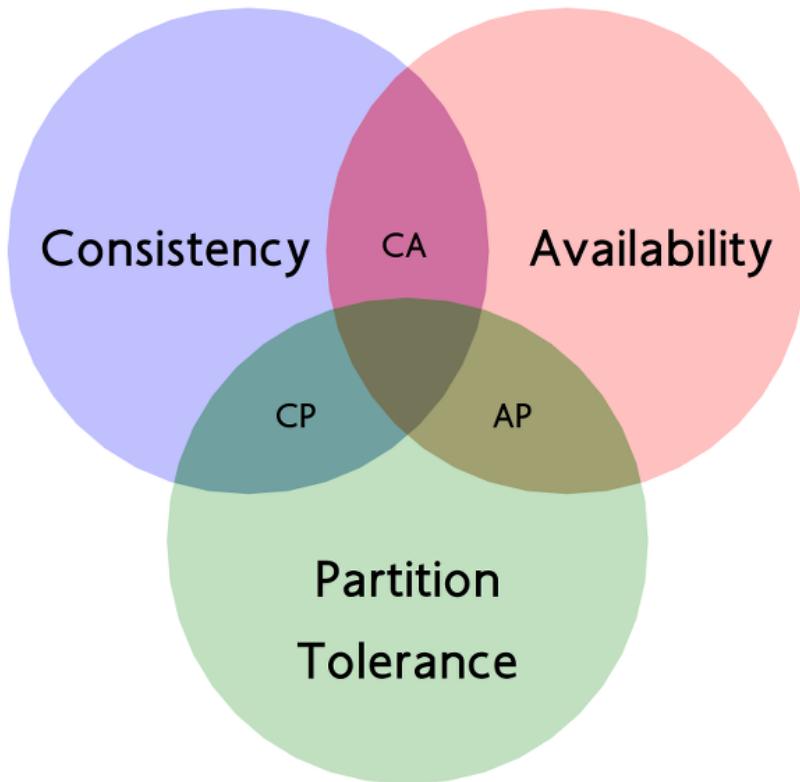


It is situations like these that motivated the CAP theorem.

- Eric Brewer, late 90s.
- “Consistency, Availability, Partition Tolerance: choose any two.”



# NoSQL



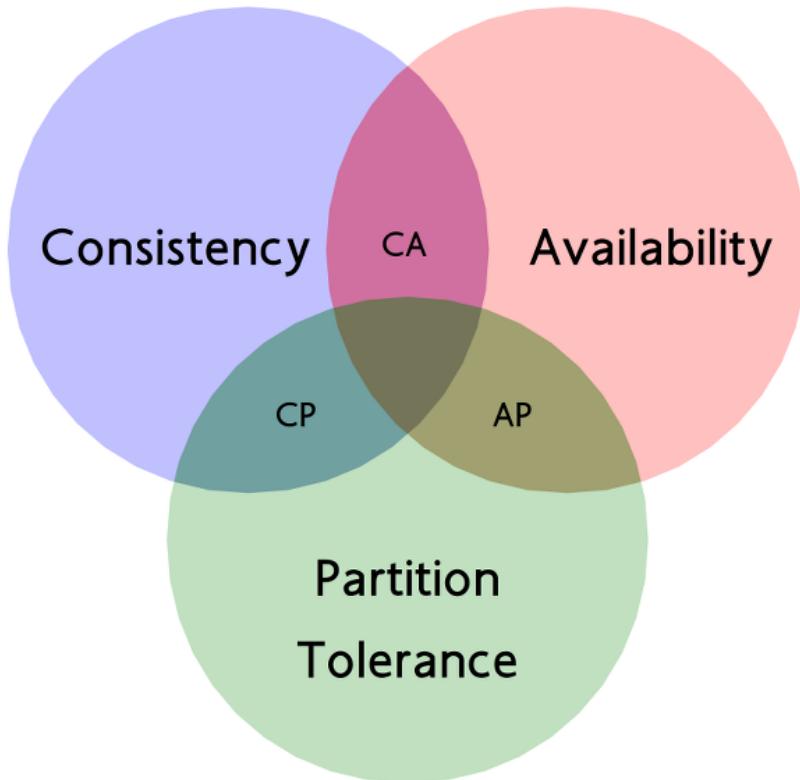
AP systems:

- Always up
- Can handle network partitions
- Not always consistent.

In the earlier example, an AP solution would accept the writes.



# NoSQL



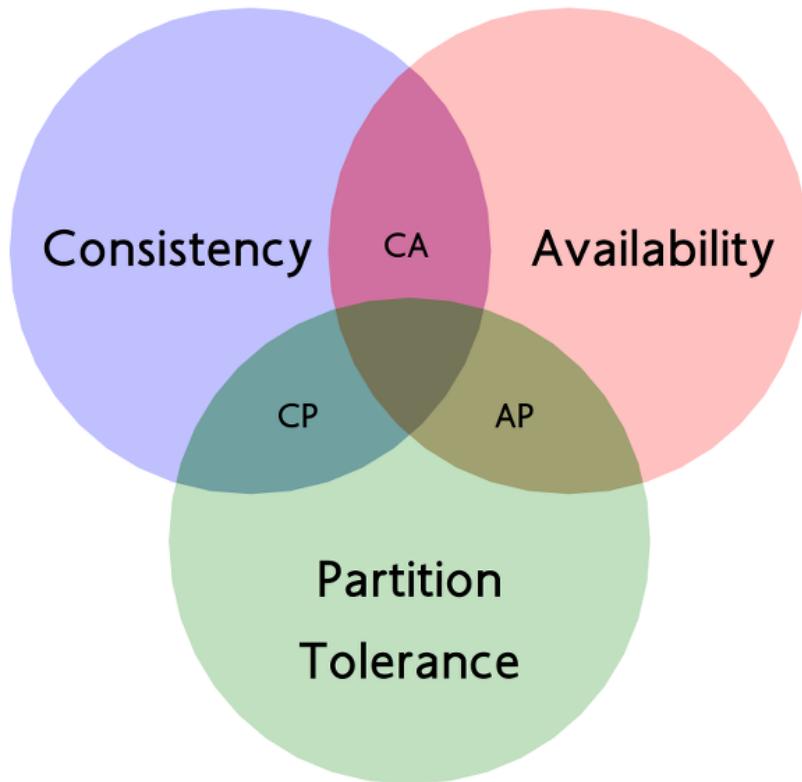
CP systems:

- Always consistent.
- Handle network partitions.
- Sometimes will be unavailable to clients

In the earlier example,  
a CP solution would not  
allow any writes



# NoSQL



CA systems:

- Always up.
- Always consistent.
- Assume no network partitions.

A CA solution would never get into the earlier scenario because it wouldn't be deployed where partitions could happen.



# ACID vs. BASE

The **BASE** acronym was created to describe these NoSQL solutions that make tradeoffs between Availability and Consistency

## ACID

Atomicity  
Consistency  
Isolation  
Durability

## BASE

Basically Available  
Soft State  
Eventually Consistent



# Consistency

Consistency comes in many forms:

- **Strong Consistency**
  - After update, everyone sees new value
- **Eventual Consistency**
  - Eventually the system will converge on the new value
  - Read-your-writes Consistency
    - You immediately see any data you have written
  - Causal Consistency
    - You see your own writes, and anyone you communicate with sees your writes
  - Session Consistency
    - Within a session, you see your own writes.



# NWR



N, W & R are a useful shorthand for describing the read/write strategy of a data store.



# NWR

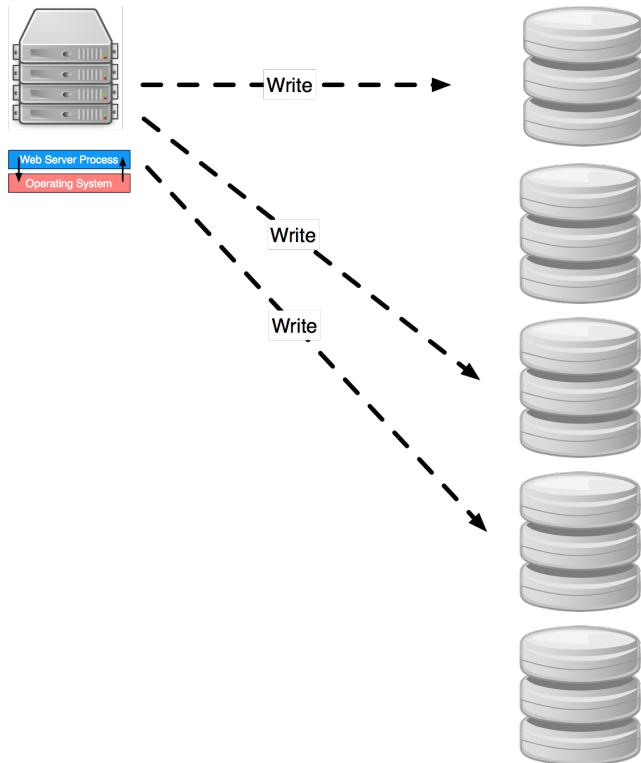


N refers to the number of separate nodes that each retain a copy of the data.

In this example, N is 5.



# NWR

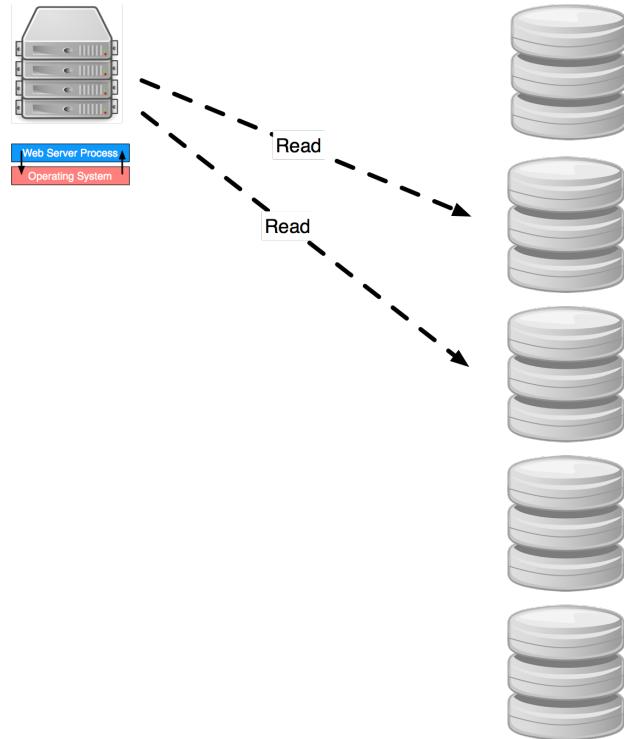


W refers to the number of nodes that we persist to before considering a write written.

In this example,  
W is 3.



# NWR

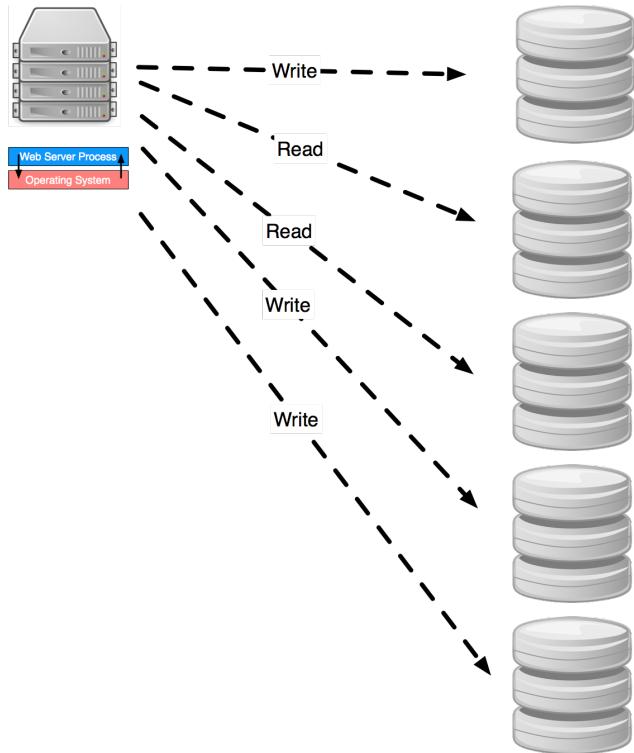


R refers to the number of nodes that we consult when reading.

In this example, R is 2.



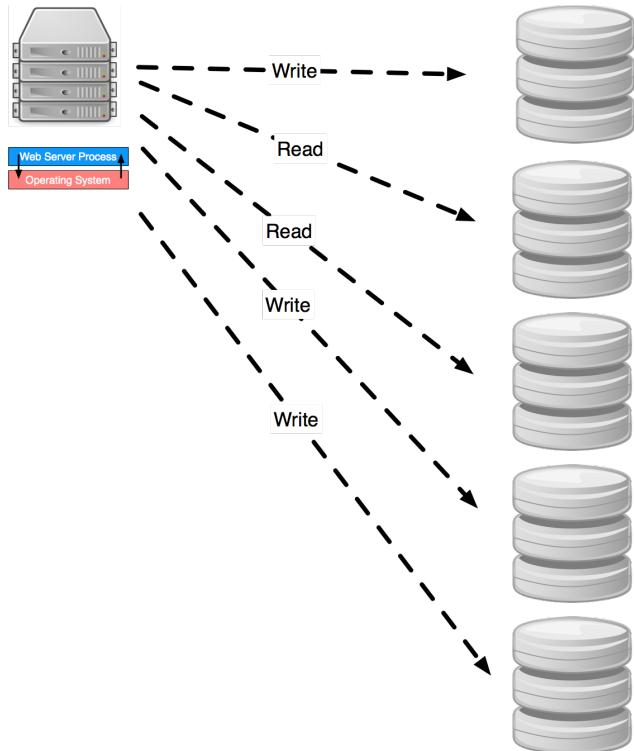
# NWR



If  $W + R \leq N$ , then you can't be sure that a read has seen all previous writes



# NWR

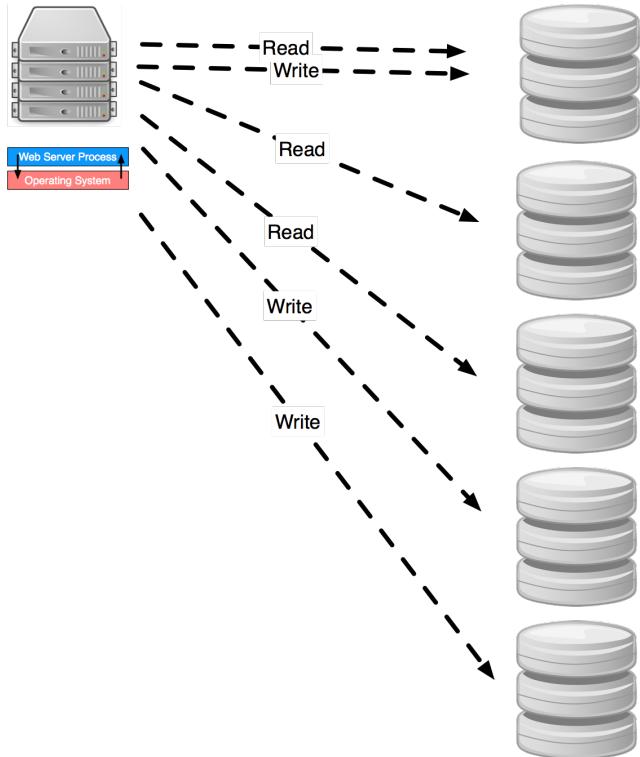


If  $W + R \leq N$ , then you can't be sure that a read has seen all previous writes.

$$3 + 2 \leq 5.$$



# NWR



If  $W + R > N$ , then you can be sure that a read has seen all previous writes.

Any two size-3 subsets of 5 servers must have overlap

$$3 + 3 > 5.$$



For strong consistency, many combinations can work.

- $R=N, W=1$ : Write to any one server, consult all servers on reads. Use the newest value.
- $R=1, W=N$ : Write to all servers, consult any server on reads.
- $R = N/2 + 1, W = N/2 + 1$ : Write to a quorum, read from a quorum.



For weaker notions of consistency, we choose  $W + R \leq N$ .

Exactly which type of consistency we see will depend on “session stickiness”

- I can write to multiple nodes, but if User A’s reads and writes to the same node, we can more easily implement “see your own writes” consistency.
- Similar use of stickiness to achieve Session Consistency.



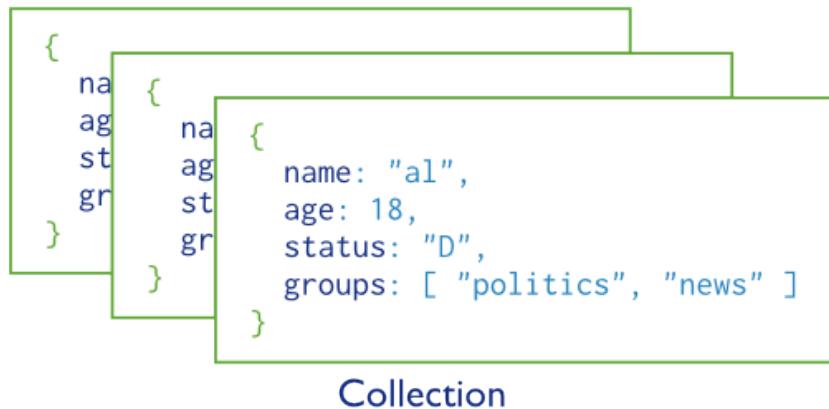
# NoSQL

There are different types of NoSQL stores

- Document-oriented stores
  - We will look at MongoDB
- Key-value stores
  - We will look at Redis
- Column-oriented data stores
  - We will look at Cassandra
- Graph databases
  - We won't be looking at these today
  - Specialized data stores, not always horizontally scalable.



# MongoDB

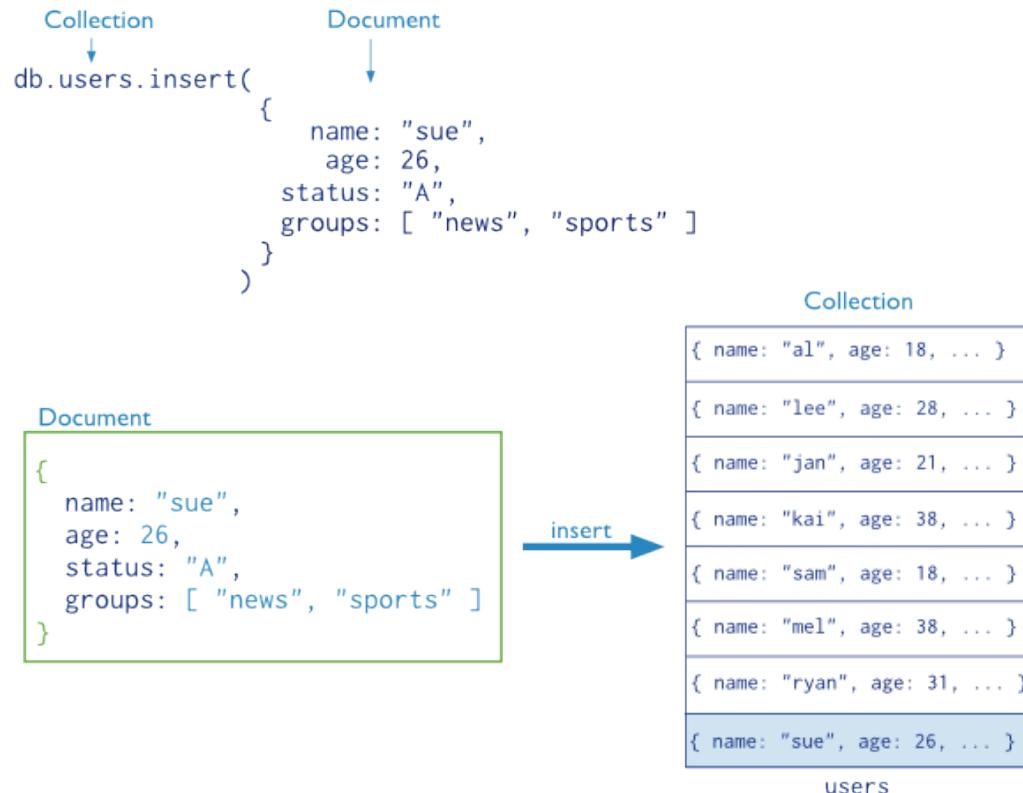


MongoDB is a Document-oriented data store.

- Stores “Documents” that are nested hash-like structures.
- These Documents are stored in “Collections” (similar to a table in RDBMS).
- Has no fixed Schema.
- Docs can have references to other docs
  -



# MongoDB



Query language is not  
SQL



# MongoDB

Collection  
Query Criteria  
Modifier  
`db.users.find( { age: { $gt: 18 } } ).sort( {age: 1} )`

{ age: 18, ... }
{ age: 28, ... }
{ age: 21, ... }
{ age: 38, ... }
{ age: 18, ... }
{ age: 38, ... }
{ age: 31, ... }

Query Criteria

{ age: 28, ... }
{ age: 21, ... }
{ age: 38, ... }
{ age: 38, ... }
{ age: 31, ... }

Modifier

{ age: 21, ... }
{ age: 28, ... }
{ age: 31, ... }
{ age: 38, ... }
{ age: 38, ... }

Results

Query language is  
not SQL



# MongoDB

Documents are stored in JSONB

- Binary version of JSON
- Can nest other JSON documents



No notion of transactions

- Unit of atomicity is the Document.
- Inserting multiple documents can fail individually



# MongoDB

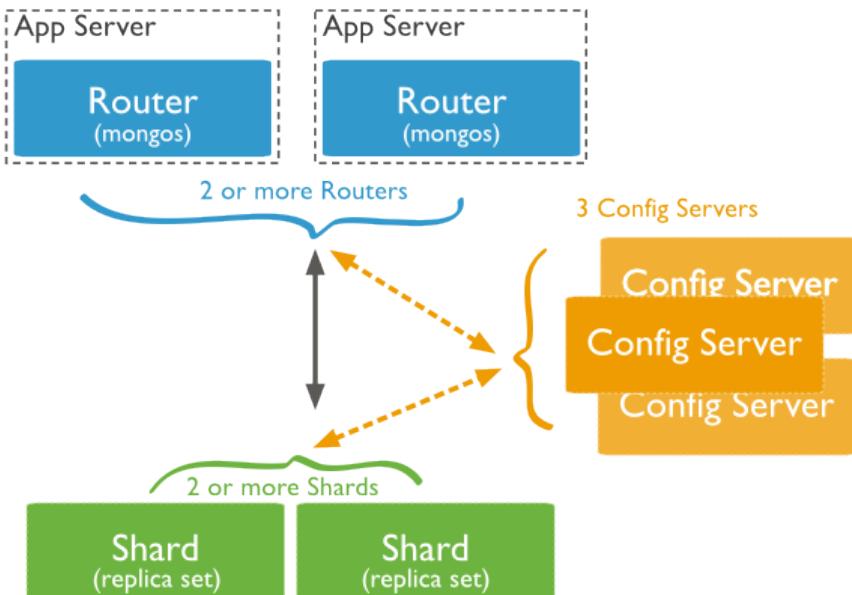
No notion of Joins.

- If you want to do computation based on relations between documents, read them into memory and do them at the application layer.

Can have secondary indexes based on document values



# MongoDB



## Collections can be sharded

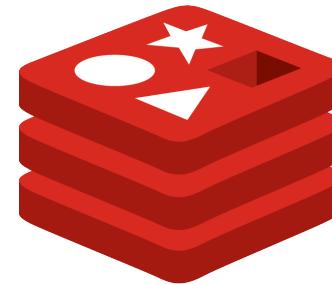
- Each shard can have a replica set
- Config Servers manage the mapping between shards and data.
- Mongos routes queries to the appropriate shard



# Redis

**Redis is a key-value data store.**

- Also called a data structure store
- Supports many data structures
  - Lists
  - Sorted Sets
  - Hashes
  - Bitmaps



redis

Primarily keeps data structures  
in memory

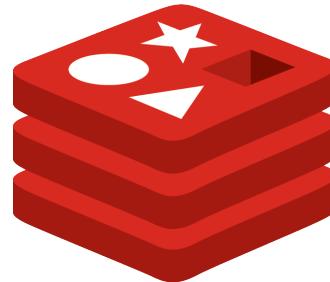
- Persistence to disk is optional



# Redis

## Redis interface

- Each data type allows similar mechanisms to what you would do in memory
- Access hashes by key
- Access lists by index
- Sorted sets can return top-K
- Push/pop on lists



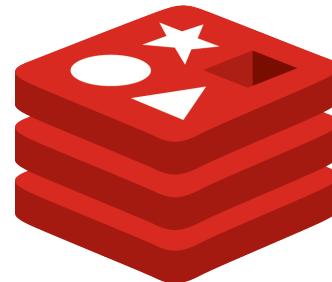
redis



# Redis

## Redis sort of has transactions

- Redis operations are simple
- You can batch up a series of commands into a transaction
- When a command fails, the previous do not roll back



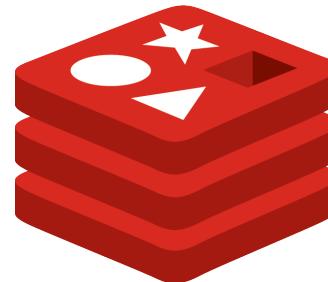
redis



# Redis

Two options for disk persistence

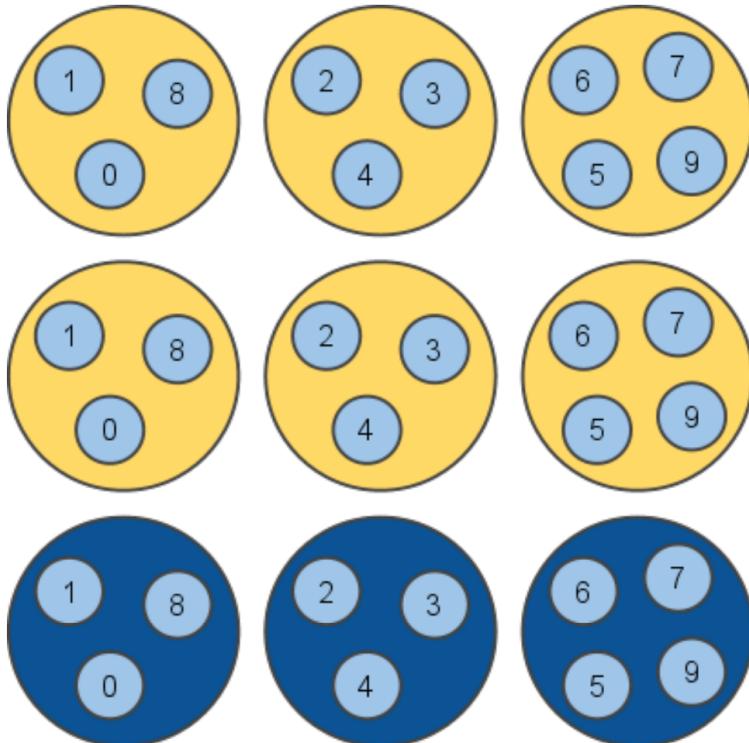
- RDF: Redis Database File
  - Forks process and saves a dump
- AOF: Append Only File
  - Saves updates to a log
  - Log is replayed upon start



redis



# Redis

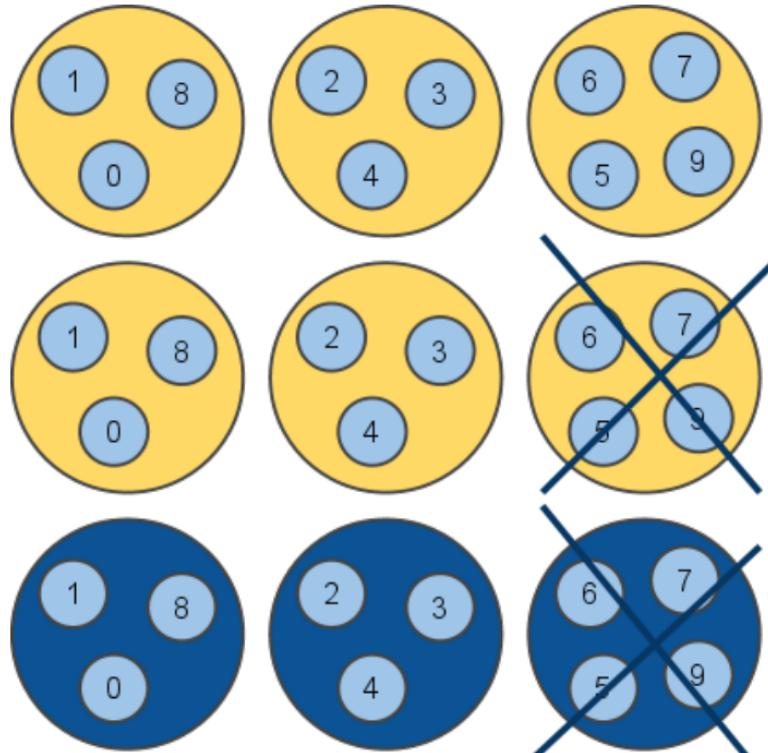


Redis cluster supports  
sharding

Single master for writes,  
replicas for reads.



# Redis



Redis cluster supports sharding

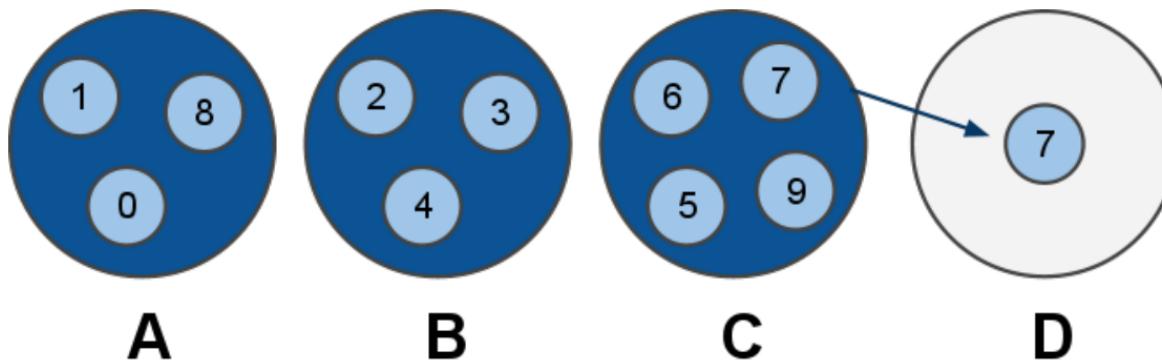
Single master for writes,  
replicas for reads.

Cluster can handle all  
reads if up to two  
nodes are down



# Redis

Cluster can also dynamically rebalance after adding hardware.



# Cassandra

Most similar to a RDBMS

Cassandra has table-like structures called ColumnFamilies

- ColumnFamilies have many rows
- Rows are like a big hash, either many keys and values
- Rows are heterogeneous and can be schemaless
- Rows can be very long
  - Have many keys and values



# Cassandra

Static Column Family:

row key	columns ...			
	name	email	address	state
jbellis	jonathan	jb@ds.com	123 main	TX
	daria	dh@ds.com	45 2 <sup>nd</sup> St.	CA
egilmore	name	email		
	eric	eg@ds.com		

Dynamic Column Family

row key	columns ...			
	dhutch	egilmore	datastax	mzcassie
jbellis				
dhutch	egilmore			
egilmore	datastax	mzcassie		



# Cassandra

Interface is called CQL, similar to SQL

```
SELECT * WHERE KEY = 11194251 AND  
startdate = '2011-10-08-0500';
```

Features are very limited

- Most queries are key-value
- Secondary indices are allowed
- Sorting is very limited



# Cassandra

No transactions

- Atomic batches exist
- No isolation from other batches

No Joins.

- Do this at the application layer



# Cassandra

- Cassandra is a masterless system
- Distributed and highly available
- Data is automatically split across nodes
- Reads are eventually consistent
  - But can be made strictly consistent (per statement)



# Cassandra

- Consistency per statement

SELECT \* WHERE KEY = 11194251...

CONSISTENCY LEVEL ONE (R=1)

CONSISTENCY LEVEL ALL (R=N)

CONSISTENCY LEVEL QUORUM (R=N/2+1)

UPDATE ... WHERE KEY = 11194251...

CONSISTENCY LEVEL ONE (W=1)

CONSISTENCY LEVEL ALL (W=N)

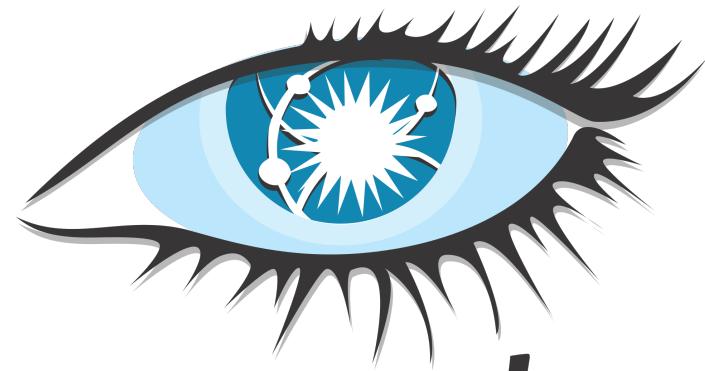
CONSISTENCY LEVEL QUORUM (W=N/2+1)



# Cassandra

Cassandra distributes keyspace across a virtual ring of nodes.

- This ring can be randomized or ordered
- Ordered allows faster range queries
- Randomized avoids hotspots



*cassandra*



# Motivation

After today's lecture you will understand how NoSQL can be used to build into scalable internet services.

NoSQL data stores won't likely be part of your project, but after this lecture you should understand when you could look to them in practice.



# For Next Time...

For Friday, finish more stories and be ready to demo.

- Also, start writing simple tsung tests for your app.

For Monday, read Chapter 4 in HPBN.

