CS188
Scalable Internet Services

John Rothfels, 12/3/20

Announcements

Final paper due **next Wednesday @ 11:59pm**.

Final presentations **next Thursday/Friday during lecture & lab**.

Extremely limited edition CS188 plant baby prize contest 🌱

How many plants + musical instruments do I have in my house?

Email answer rothfels@cs.ucla.edu for a chance to win!

Motivation

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

NoSQL data stores won't be part of your project*, but after this lecture you should understand when you could use them.

* except possibly Redis

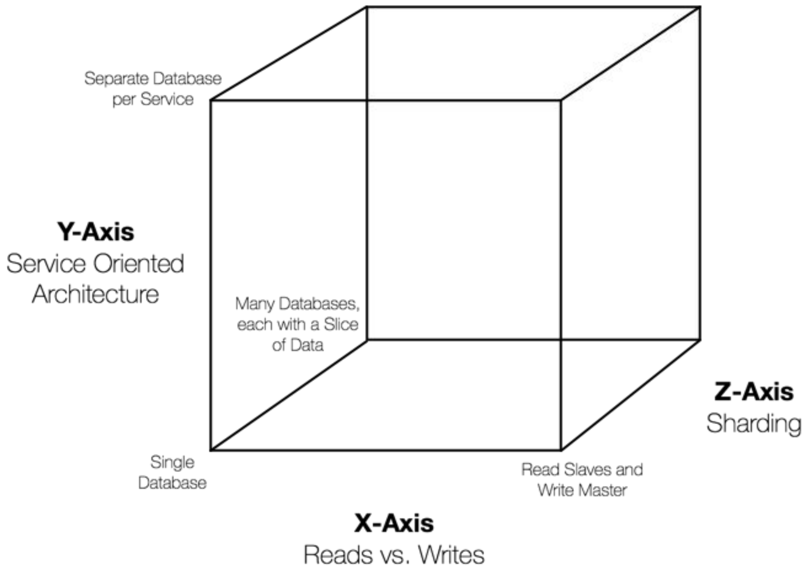
Motivation

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

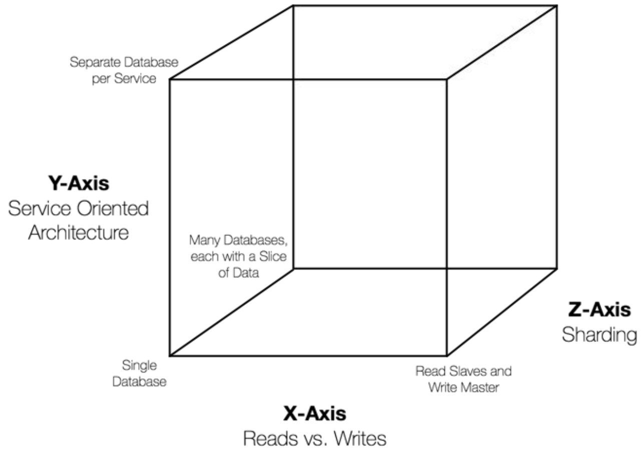
Relational Databases are great tools for our data layer, but we can't necessarily spread load across multiple RDBMSes.



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.

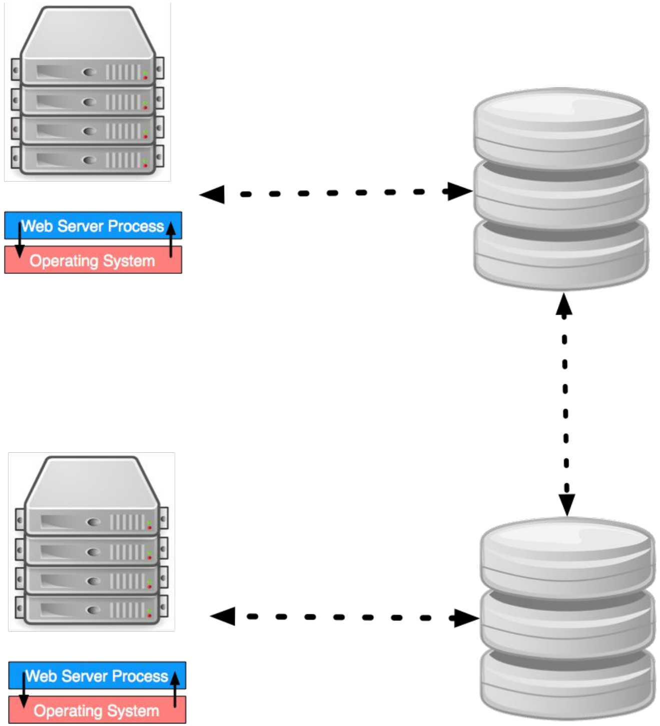
- You can easily add hardware to the database to increase throughput.

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.

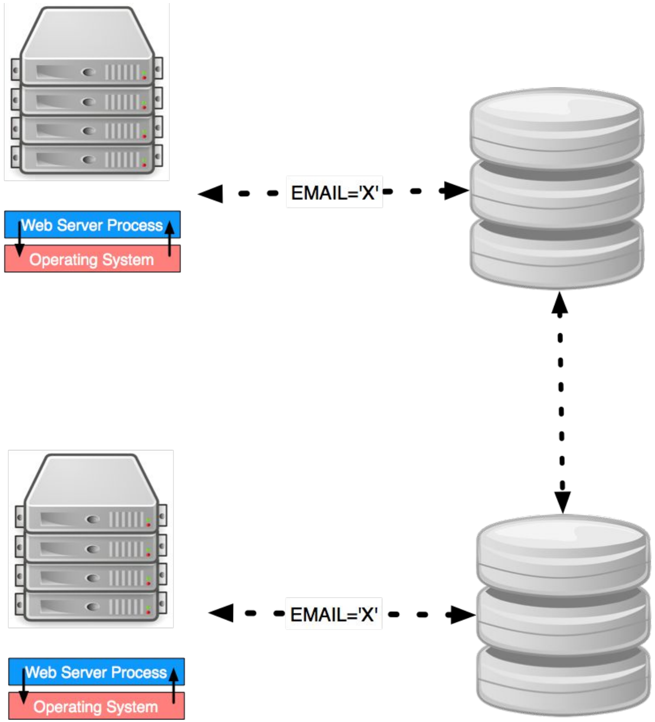
We can update on both nodes, and the databases keep each other in sync.



NoSQL

Lets say:

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

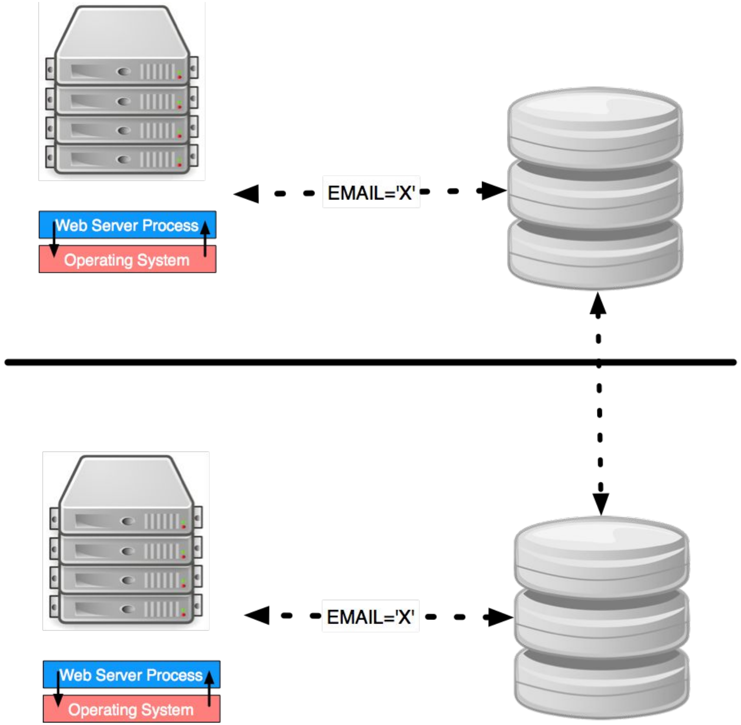


If both try to write the same email address to different rows, and the databases can communicate, they can resolve this in some manner.

- Ex: Allow one, fail the other.

NoSQL

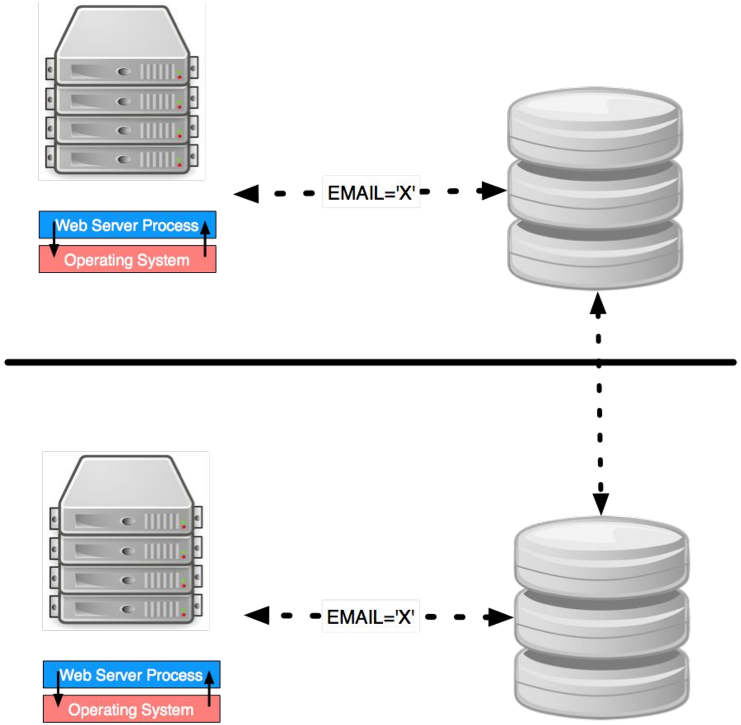
⁉️ **How do we handle a network partition?**



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?
- Fix later if needed?
- Not accept such writes during a partition?



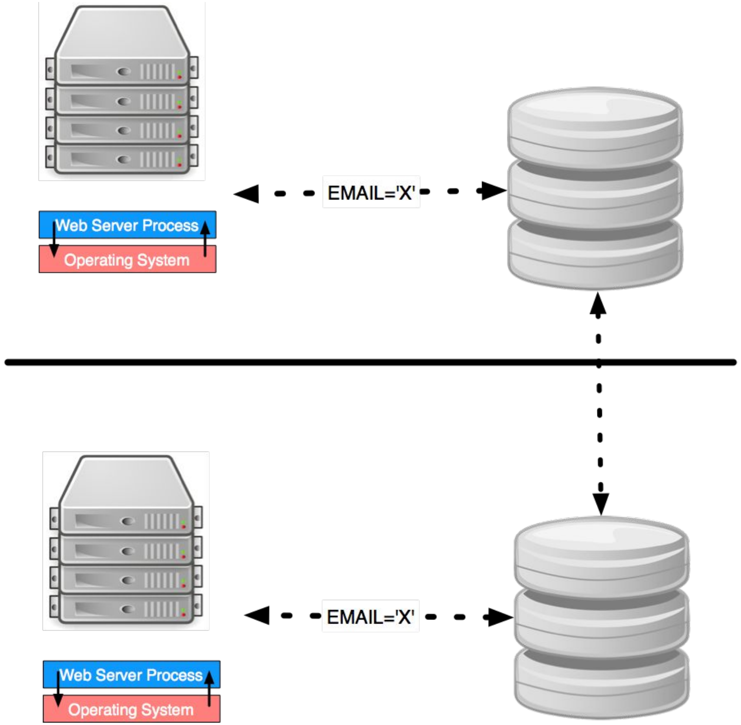
NoSQL

⁉️ **How do we handle a network partition?**

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

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

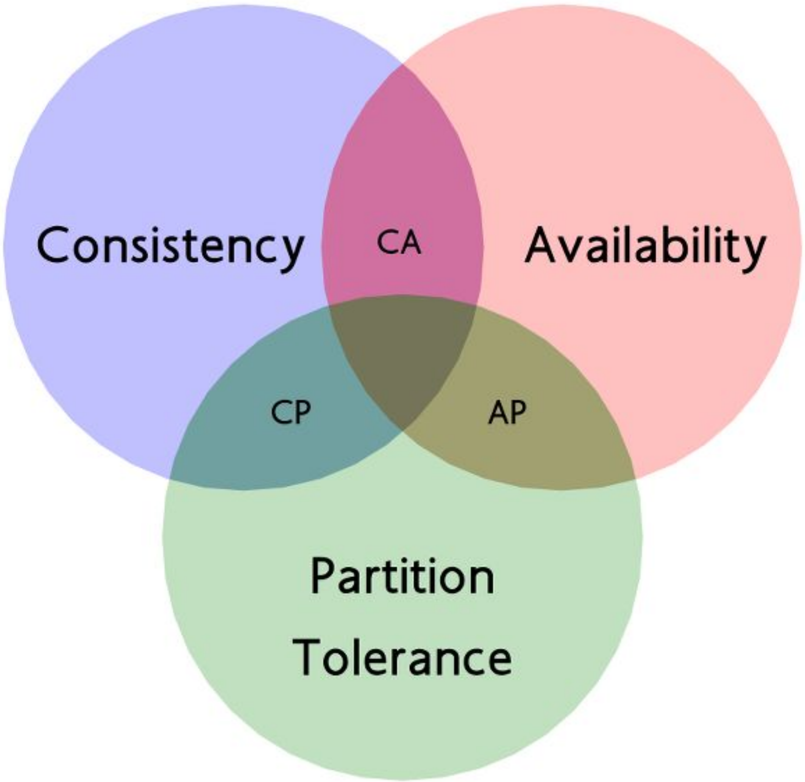
If we ignore this scenario, we can't tolerate network partitions.



NoSQL

It is situations like these that motivated the **CAP theorem** by Eric Brewer in the late 90s.

“Consistency, Availability, Partition Tolerance choose any two.”

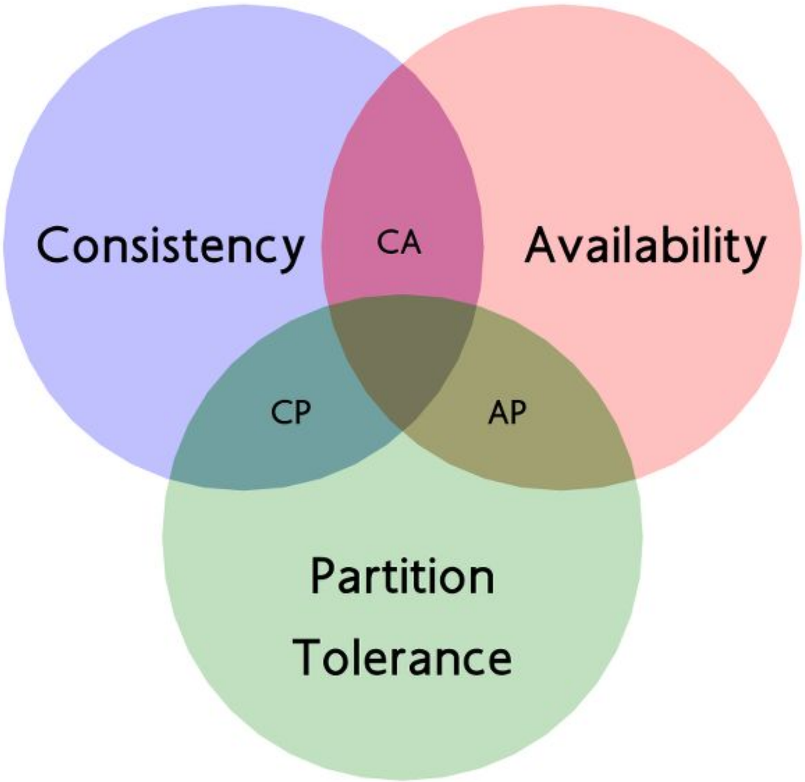


NoSQL

AP systems:

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

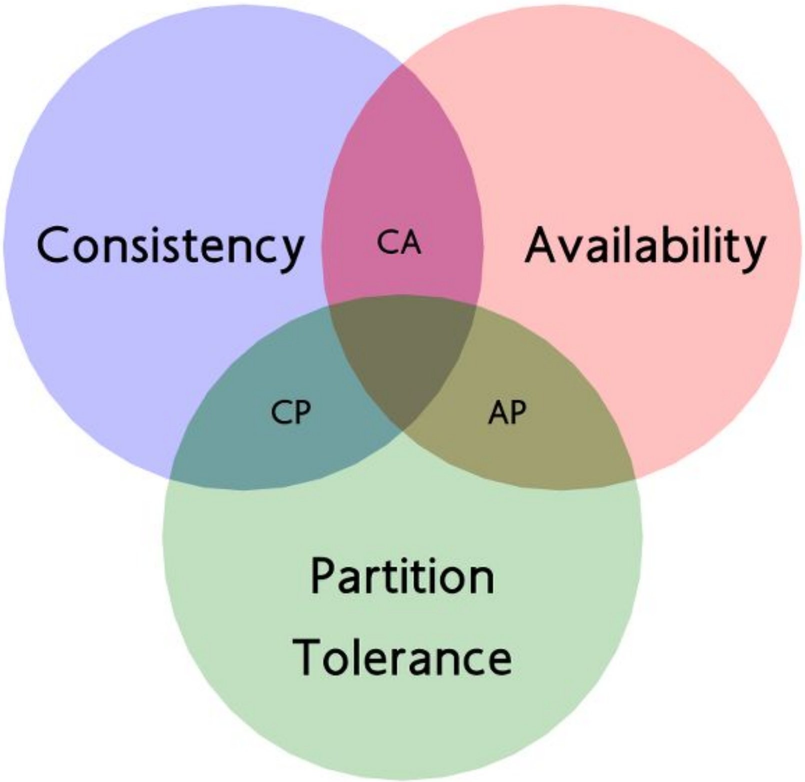
In the earlier example, an AP solution would accept the writes.



NoSQL

CP systems:

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

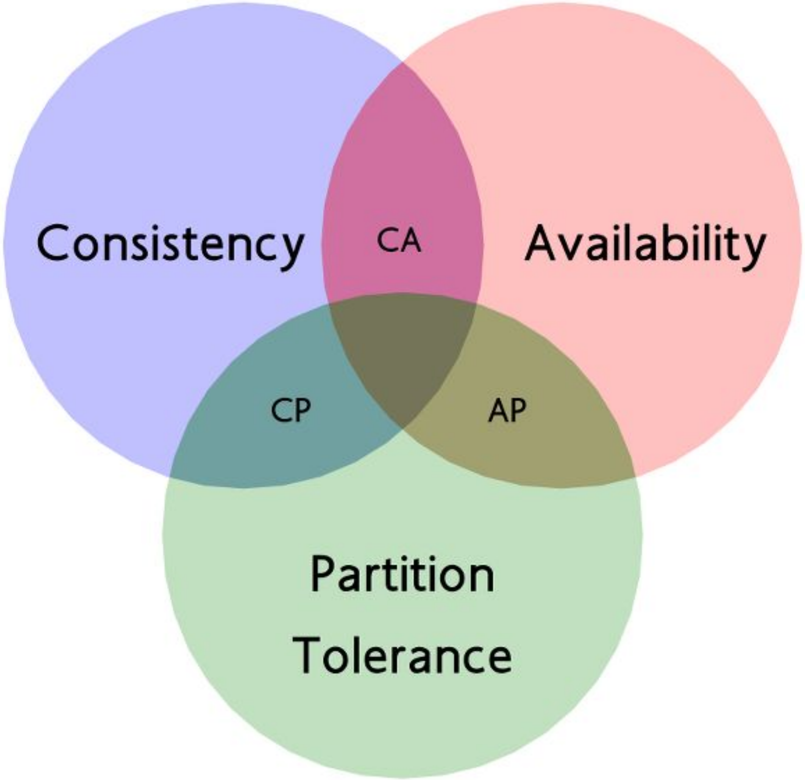


In the earlier example, a CP solution would not allow any writes.

NoSQL

CA systems:

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



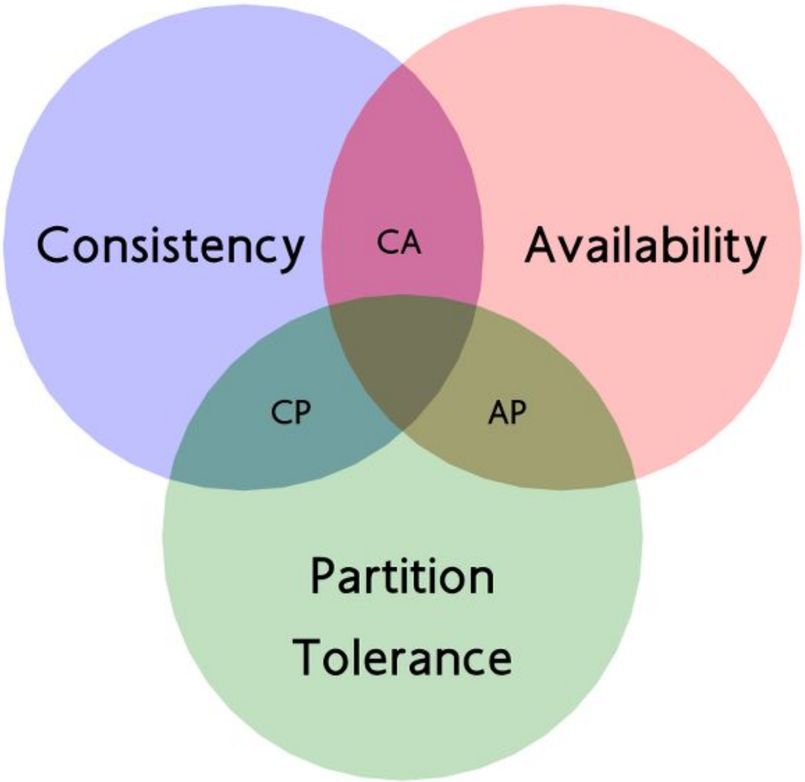
A CA solution would never get into the earlier scenario because it wouldn't be deployed where partitions could happen.

NoSQL

Assuming no partitions is very limiting:

- For HA and latency reasons, we'd like to have multi-site
- Even within a datacenter, we could have a partition

As a result, trade-offs tend to be more C vs A



ACID vs. BASE

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

ACID	**BASE**
Atomicity	**B**asically
Consistency	**A**vailable
Isolation	**S**oft State
Durability	**E**ventually Consistent

Consistency

Consistency comes in many forms:

Strong Consistency

- After update, everyone sees new value

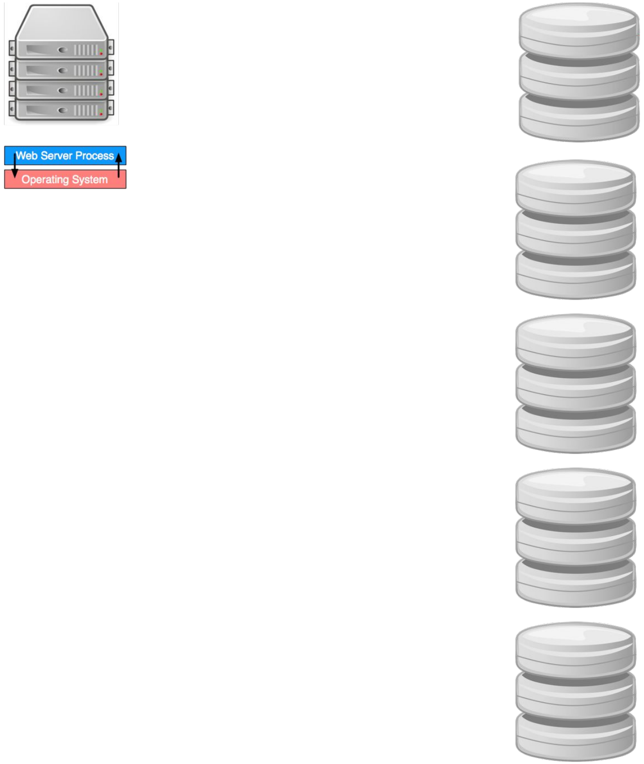
Consistency

Consistency comes in many forms:

Eventual Consistency

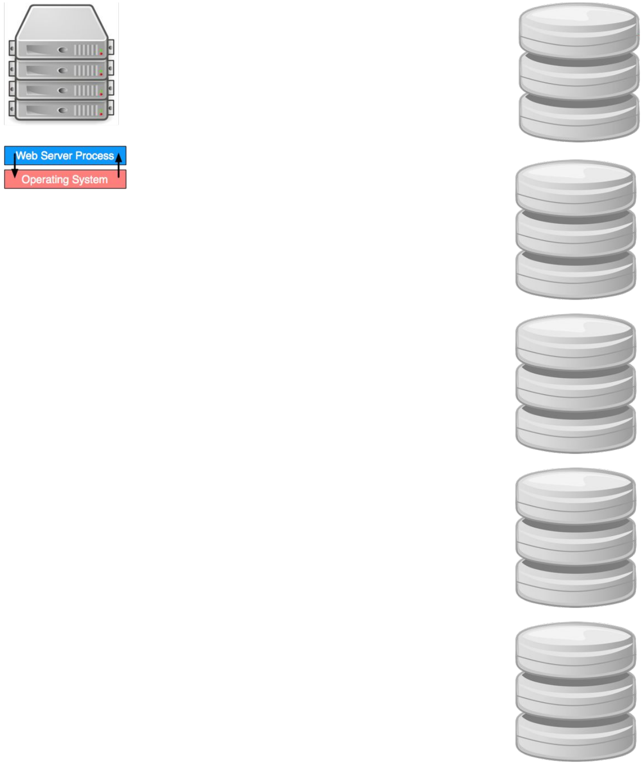
- Eventually the system will converge on the new value
- Read-your-writes Consistency
    - You immediately see any data you have written
- Causal Consistency
    - You see your own writes, and anyone you communicate with sees your writes
- Session Consistency
    - Within a session, you see your own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

NWR



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

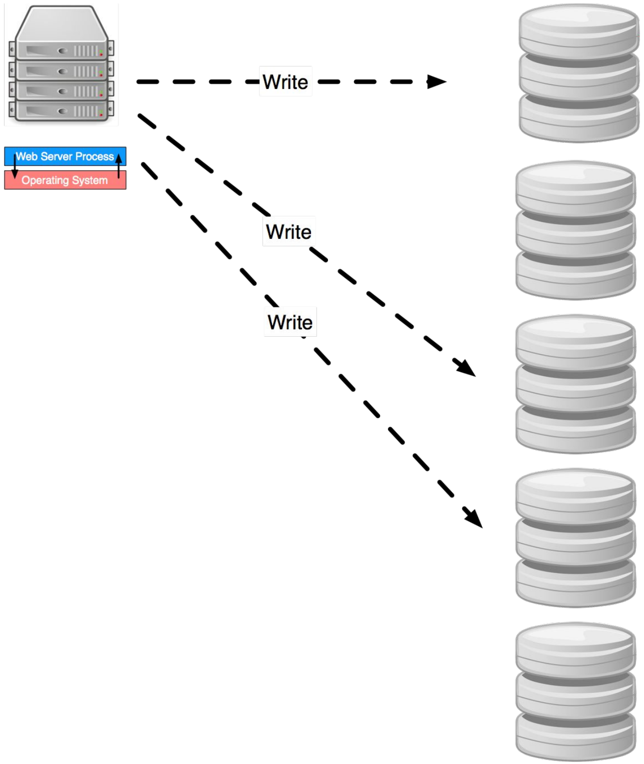
NWR



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

In this example, N is 5.

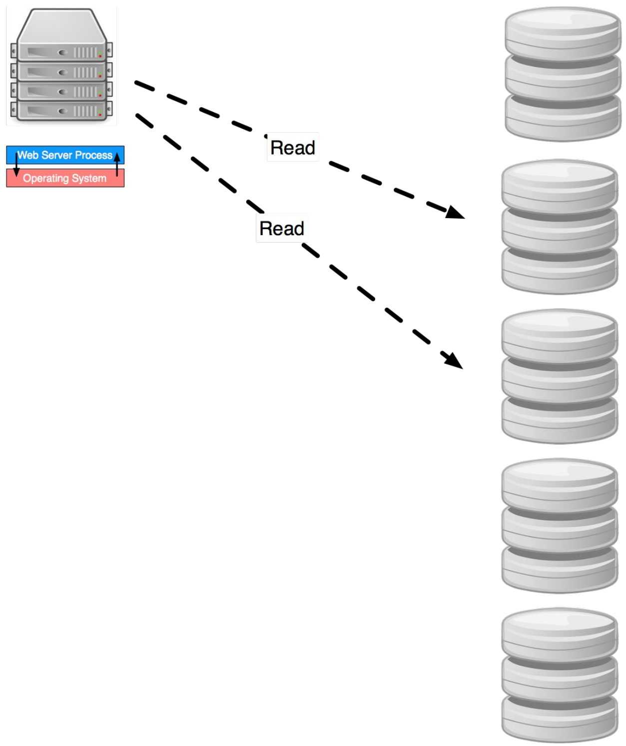
NWR



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

In this example, W is 3.

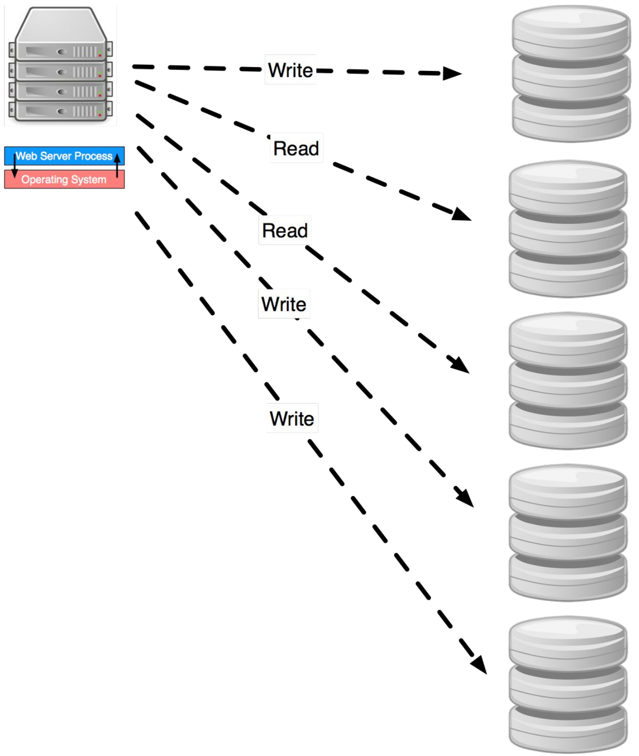
NWR



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

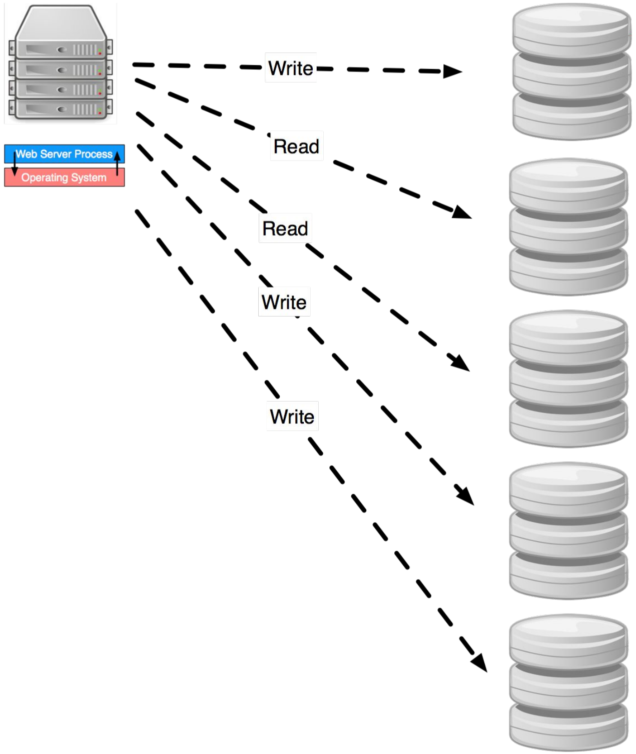
In this example, R is 2.

NWR



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

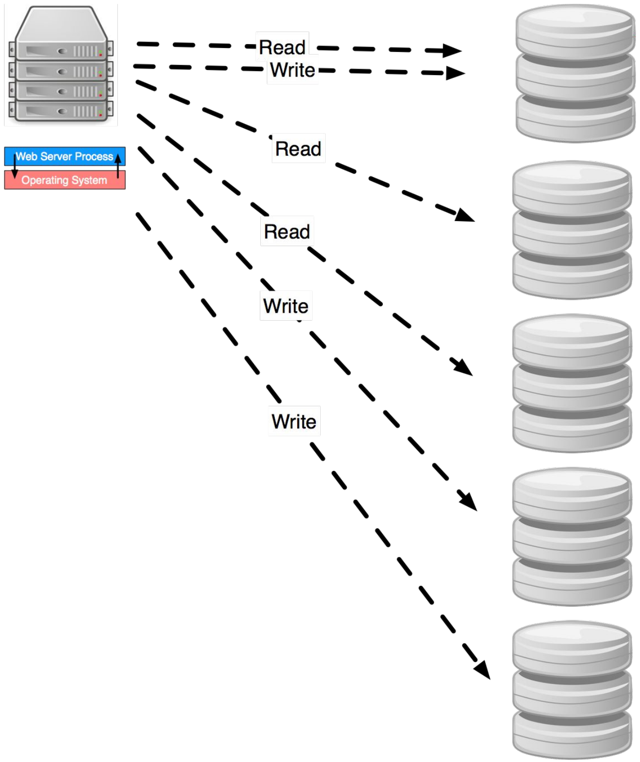
NWR



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

3 + 2 <= 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.

NWR

For strong consistency, many combinations can work.

- R=N, W=1: Write to any one server, consult all server 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.

NWR

For weaker notions of consistency, we choose W + R <= 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

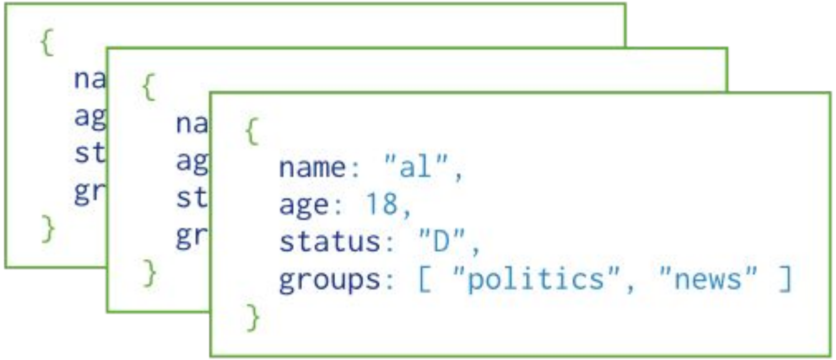
There are different types of NoSQL stores.

- 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

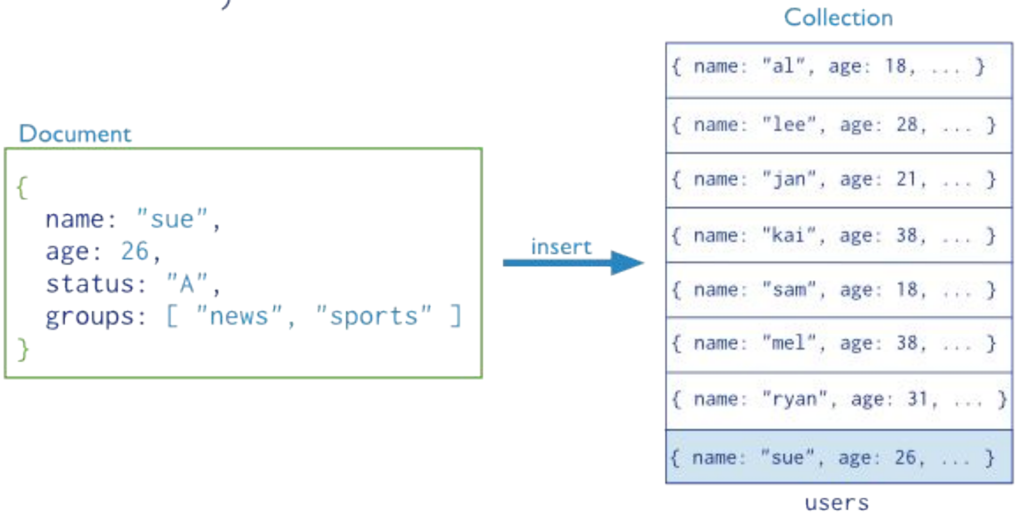


Collection

MongoDB

Collection Document

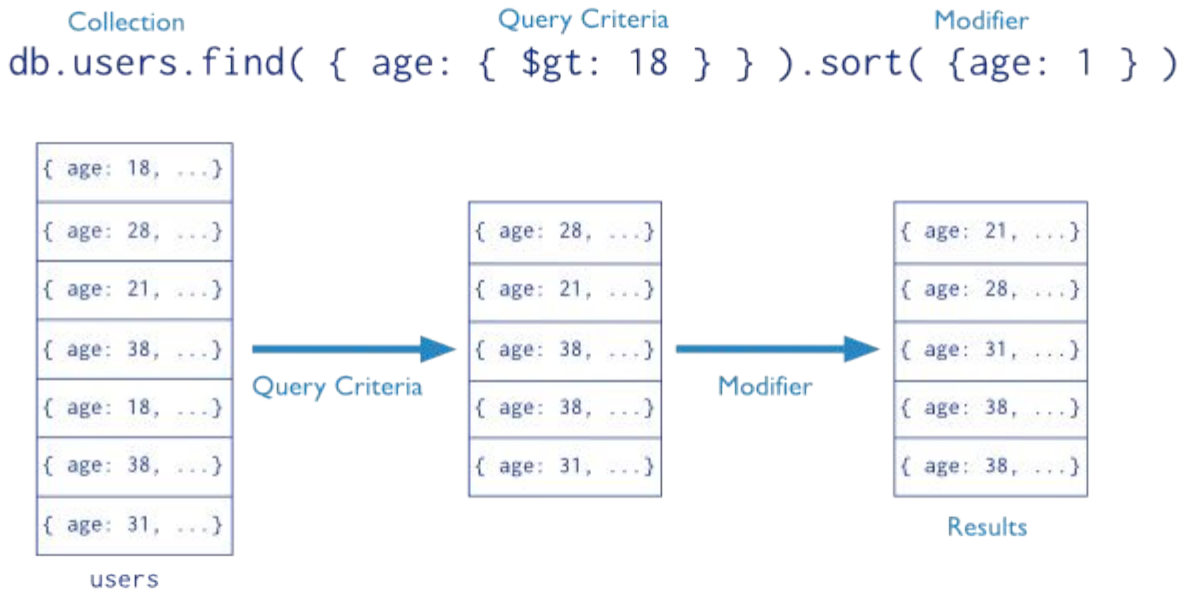
```
db.users.insert(
                 {
                   name: "sue",
                    age: 26,
                 status: "A",
                 groups: [ "news", "sports" ]
                 }
               )
```



Query language is not SQL!

MongoDB

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

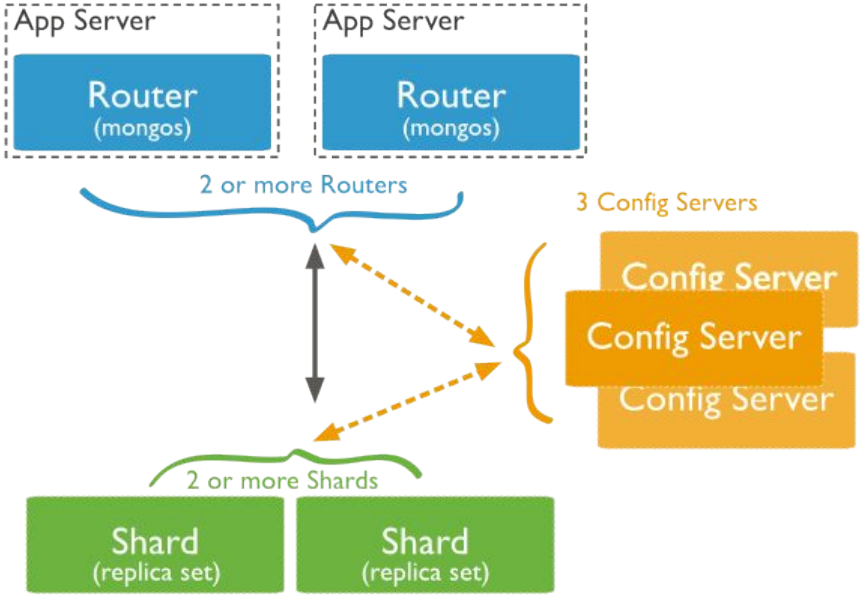
MongoDB

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, like in SQL.

MongoDB



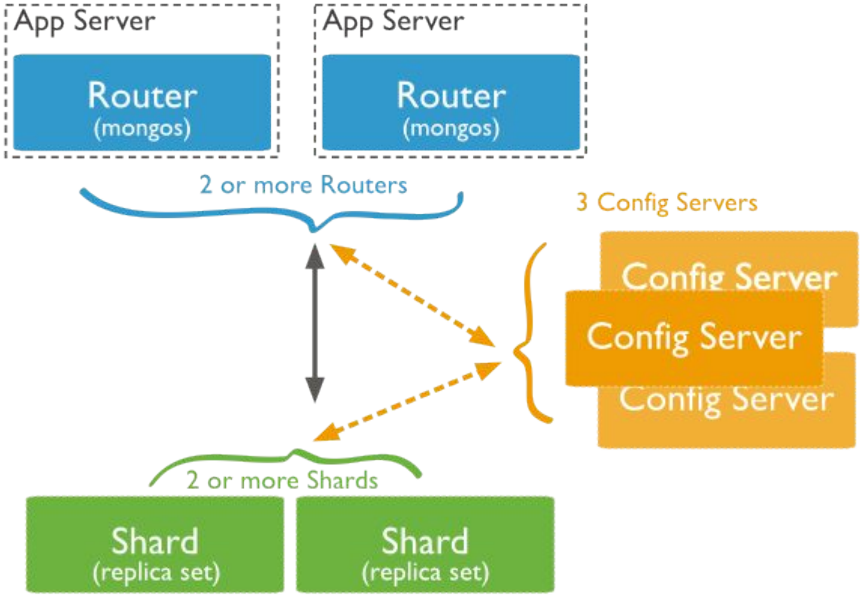
Collections can be easily sharded!

Each shard can have a replica set.

Config Servers manage the mapping between shards and data.

Mongo routes queries to the appropriate shard.

MongoDB



Replica sets use asynchronous replication

You can configure your driver to read from the primary only, or to read from read-replicas

- Reading from primary
  - (R=1, W=1, N=1)
- Reading from replicas
  - (R=1, W=1, N=3)

MongoDB

Pros:

- easy to store schema-less JSON
- don’t have to learn SQL, joins
- manages horizontal scaling for you

Cons:

- no schema is *very* hard to manage at scale without principled engineering
- no joins, no transactions
- de-normalizing data for query performance makes your database bigger, and also harder to keep consistent (you have to update every redundant copy)

Aside: (de-)normalization

De-normalization is a strategy we can use on a database to increase read performance (at the expense of losing some write performance and increased storage) by adding redundant copies of data or by grouping data.

A normalized database is one in which there is no redundant data. Relations are stored with (id) references.

userId	authToken
1	a6sg96nz

id	name	email
1	John	john@me

userId	playlist	genre
1	Kinda Blue	jazz

Aside: (de-)normalization

De-normalization is a strategy we can use on a database to increase read performance (at the expense of losing some write performance and increased storage) by adding redundant copies of data or by grouping data.

A **de-normalized** database is one in which there is redundant data. Instead of joining, you can read all the data you need from one table.

userId	authToken
1	a6sg96nz

id	name	email	authToken
1	John	john@me	a6sg96nz

userId	playlist	genre
1	Kinda Blue	jazz

MongoDB vs. (My)SQL, 2020

In 2020, many SQL databases (MySQL, PostgreSQL, etc) have support for JSONB columns.

- You can put JSON strings into SQL!
- You can tell SQL to generate (virtual) columns from data embedded in your JSON!
- You can put indices & constraints on data embedded in your JSON!

By mixing SQL and JSON, you get combined benefits of “schemaless” document storage, with structured SQL database.

MongoDB vs. (My)SQL, 2020

```
CREATE TABLE email {
  id bigint(20) NOT NULL AUTO_INCREMENT,
  json JSON NOT NULL,
  fromAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.fromAddress'))) VIRTUAL NOT NULL,
  toAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.toAddress'))) VIRTUAL NOT NULL,
  gmailId varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.gmailId'))) VIRTUAL,
  UNIQUE KEY gmailId (gmailId)
}
```

MongoDB vs. (My)SQL, 2020

```
CREATE TABLE email {
  id bigint(20) NOT NULL AUTO_INCREMENT, // 2 data columns
  json JSON NOT NULL,
  fromAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.fromAddress'))) VIRTUAL NOT NULL,
  toAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.toAddress'))) VIRTUAL NOT NULL,
  gmailId varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.gmailId'))) VIRTUAL,
  UNIQUE KEY gmailId (gmailId)
}
```

MongoDB vs. (My)SQL, 2020

```
CREATE TABLE email {
  id bigint(20) NOT NULL AUTO_INCREMENT,
  json JSON NOT NULL,
  fromAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.fromAddress'))) VIRTUAL NOT NULL,
  toAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.toAddress'))) VIRTUAL NOT NULL,
  gmailId varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.gmailId'))) VIRTUAL,
  // 3 generated columns from data inside the `json` column
  UNIQUE KEY gmailId (gmailId)
}
```

MongoDB vs. (My)SQL, 2020

```
CREATE TABLE email {
  id bigint(20) NOT NULL AUTO_INCREMENT,
  json JSON NOT NULL,
  fromAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.fromAddress'))) VIRTUAL NOT NULL,
  toAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.toAddress'))) VIRTUAL NOT NULL,
  gmailId varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.gmailId'))) VIRTUAL,
  UNIQUE KEY gmailId (gmailId) // constraint on generated column
}
```

MongoDB vs. (My)SQL, 2020

```
CREATE TABLE email {
  id bigint(20) NOT NULL AUTO_INCREMENT,
  json JSON NOT NULL,
  fromAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.fromAddress'))) VIRTUAL NOT NULL,
  toAddress varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.toAddress'))) VIRTUAL NOT NULL,
  gmailId varchar(255) GENERATED ALWAYS AS
     (json_unquote(json_extract(`info`,_utf8mb4'$.gmailId'))) VIRTUAL,
  UNIQUE KEY gmailId (gmailId)
}

insert into email (json) values ('{...}') // json must pass SQL constraints
```

Redis

Redis is a key-value data store.

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

Primarily keeps data structures in memory.

Persistence to disk is optional.

Redis

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
- Etc.

Here’s a full list of commands. Here’s a cheatsheet.

Redis

Redis *sort of* has transactions

- Redis operations are simple and atomic / thread safe*
- You can batch up a series of commands into a "transaction"
- When a command fails, the previous do not roll back

* Your application code might not be thread safe though!

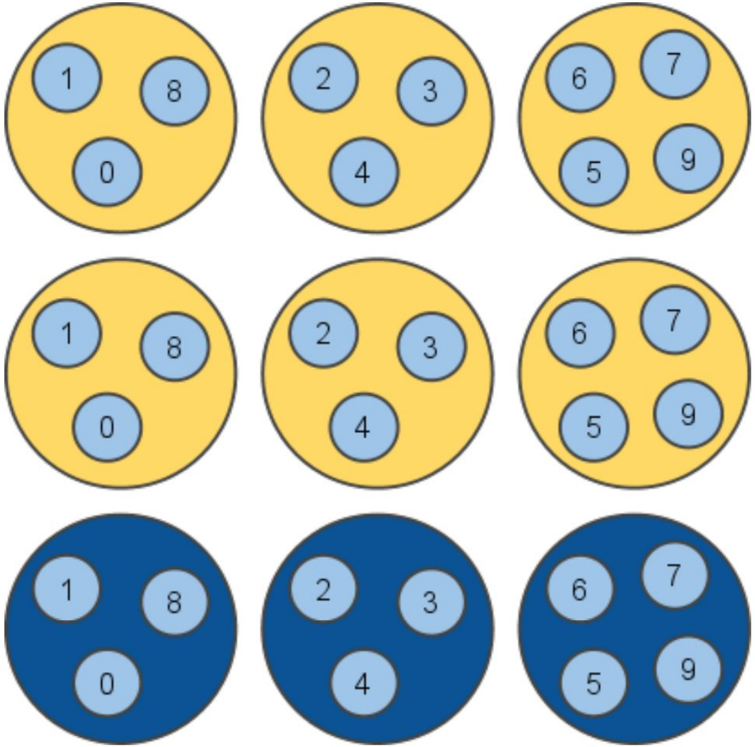
Redis

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

Most often, I’ve seen Redis used as a memory-only cache. No persistence!

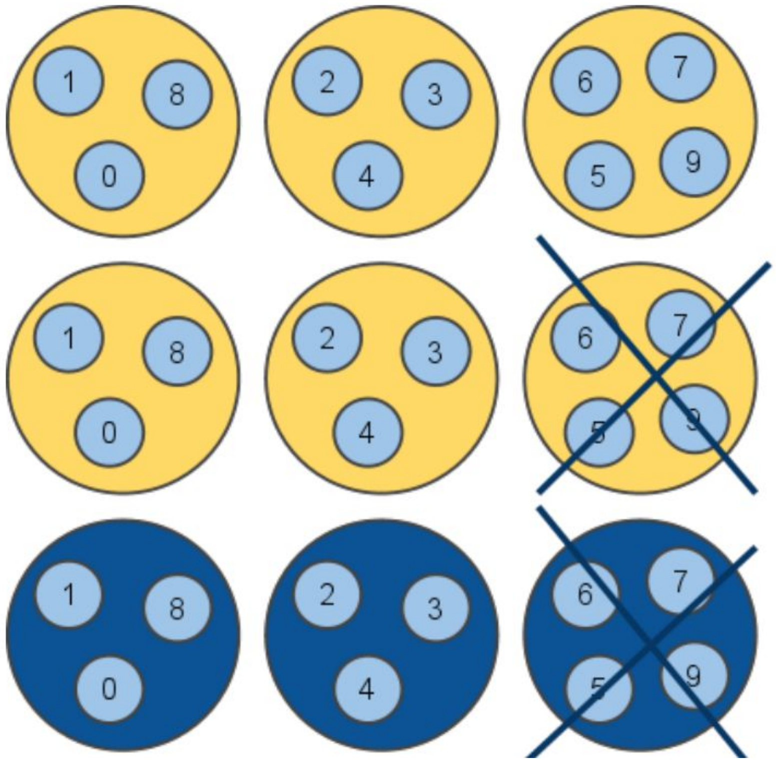
Redis



Redis cluster supports sharding.

Single master for writes, replicas for failover.

Redis

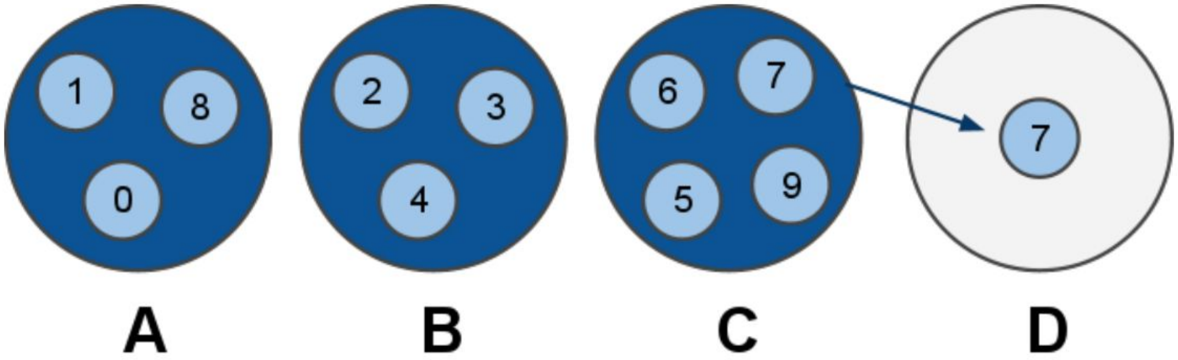


Cluster can handle all reads if up to two nodes are down.

It's possible to read from slaves, but default sends all read and write operations to master.

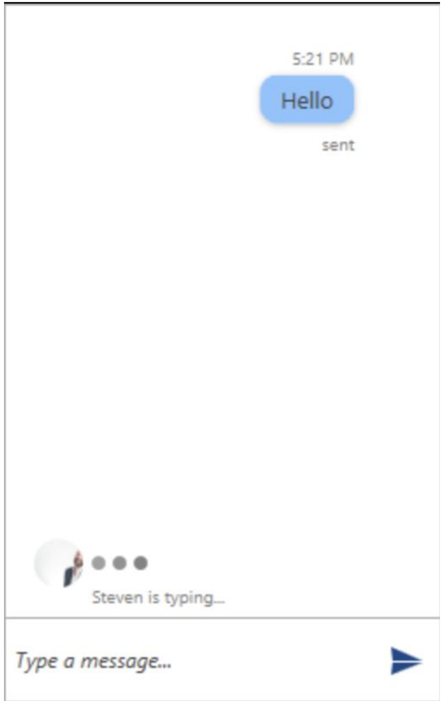
Redis

Cluster can also dynamically rebalance after adding hardware.



Redis

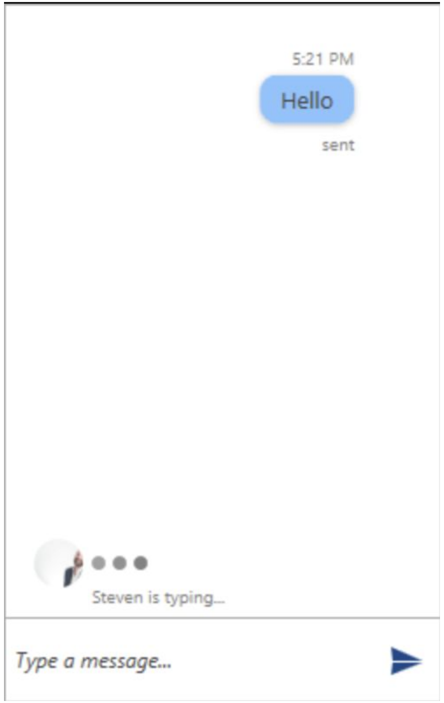
⁉️ **How/why might we use Redis to implement typing indicators?**



Redis

⁉️ **How/why might we use Redis to implement typing indicators?**

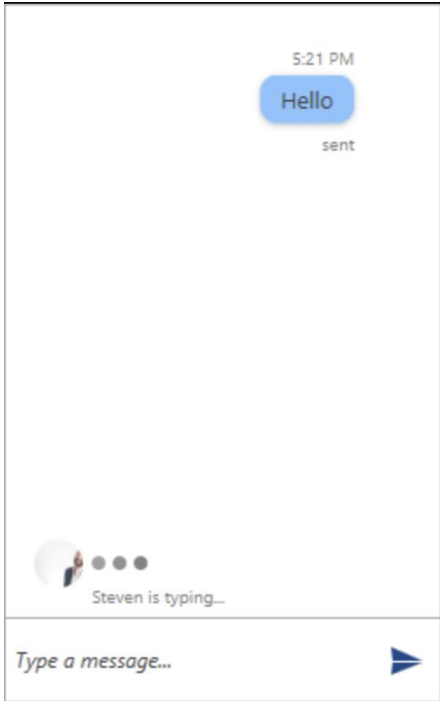
- Client (browser/phone) knows when the user is typing
- Client tells the server (via API) the user is typing
- Client polls “is typing” (via API) whenever viewing thread
    - Server should expect frequent/chatty client communication
    - You ideally show the indicator as soon as the user starts typing



Redis

⁉️ **How/why might we use Redis to implement typing indicators?**

- Polling the server can be costly if we have to talk to SQL for every request
- The server could keep “who is typing” state in memory to keep up with the polling requests, but this would break if we horizontally scale our appserver
    - Instead, the (horizontally deployed) server can talk to a shared Redis instance that keeps state in memory and can scale to handle the polling load

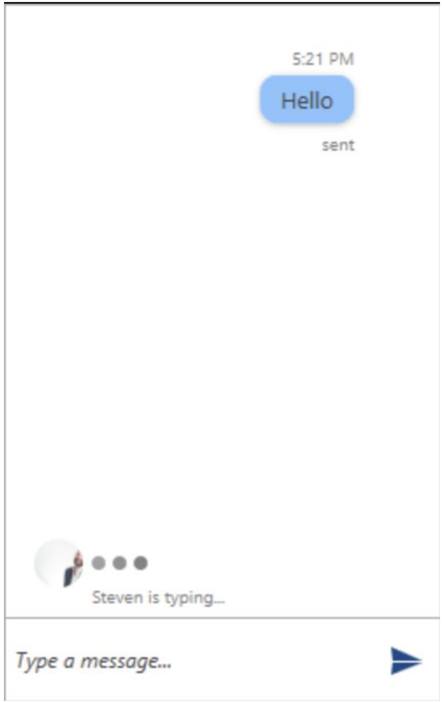


Redis

⁉️ **How/why might we use Redis to implement typing indicators?**

Key requirements:

- Fast lookup: “is there somebody typing on this thread”
- Timeouts: if somebody stops typing, the typing indicator should disappear after a few seconds

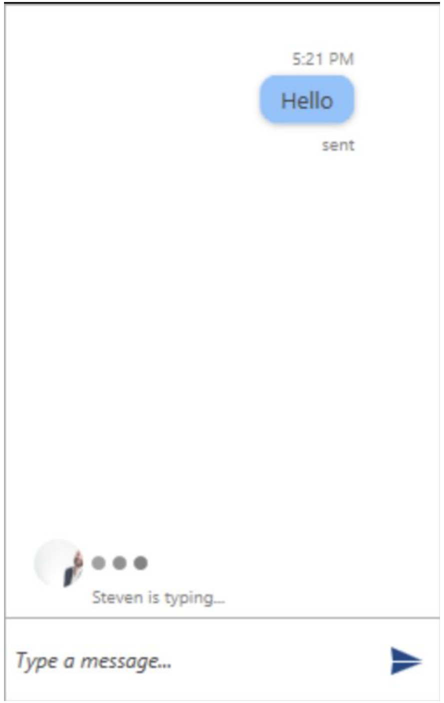


Redis

⁉️ **How/why might we use Redis to implement typing indicators?**

Data model:

```
// user 3 typing on thread 1234
thread-1234-user-3: "1"
```

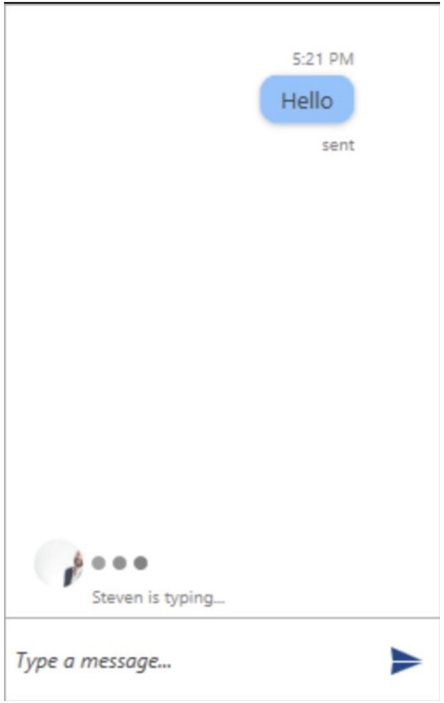


Redis

⁉️ **How/why might we use Redis to implement typing indicators?**

Set typing indicator:

```
if (isTyping) {
  // SET thread-1234-user-3 "1"
  // EXPIRE thread-1234-user-3 5
} else {
  // DELETE thread-1234-user-3
}
```

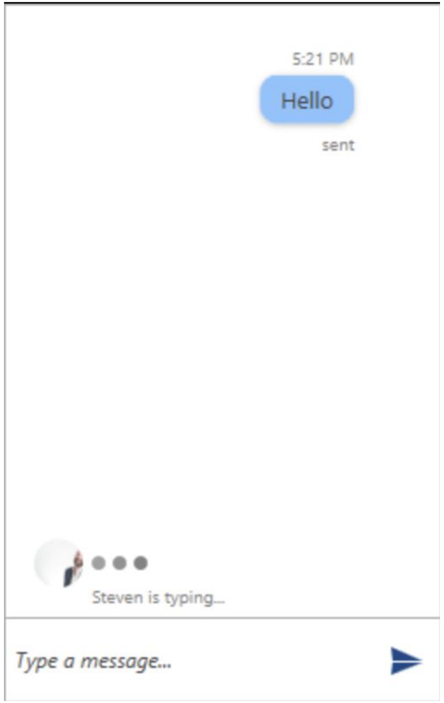


Redis

⁉️ **How/why might we use Redis to implement typing indicators?**

Get typing indicator for thread 1234:

```
KEYS thread-1234-* // users typing on 1234
```



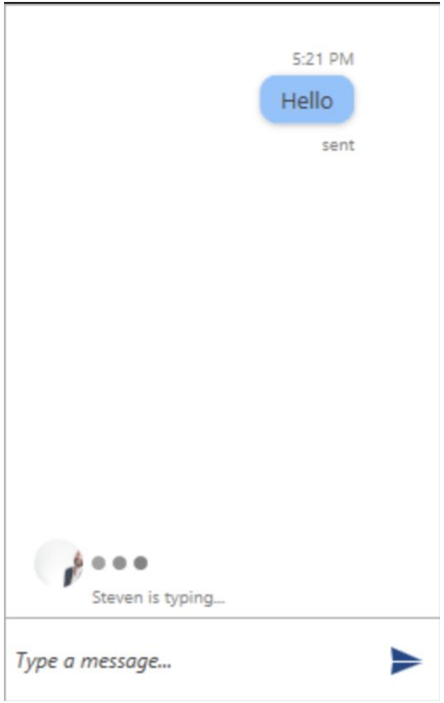
Redis

⁉️ **How/why might we use Redis to implement typing indicators?**

Data model (alt):

```
// users typing on thread 1234
thread-1234: Set(3, …)
```

Pros/cons?



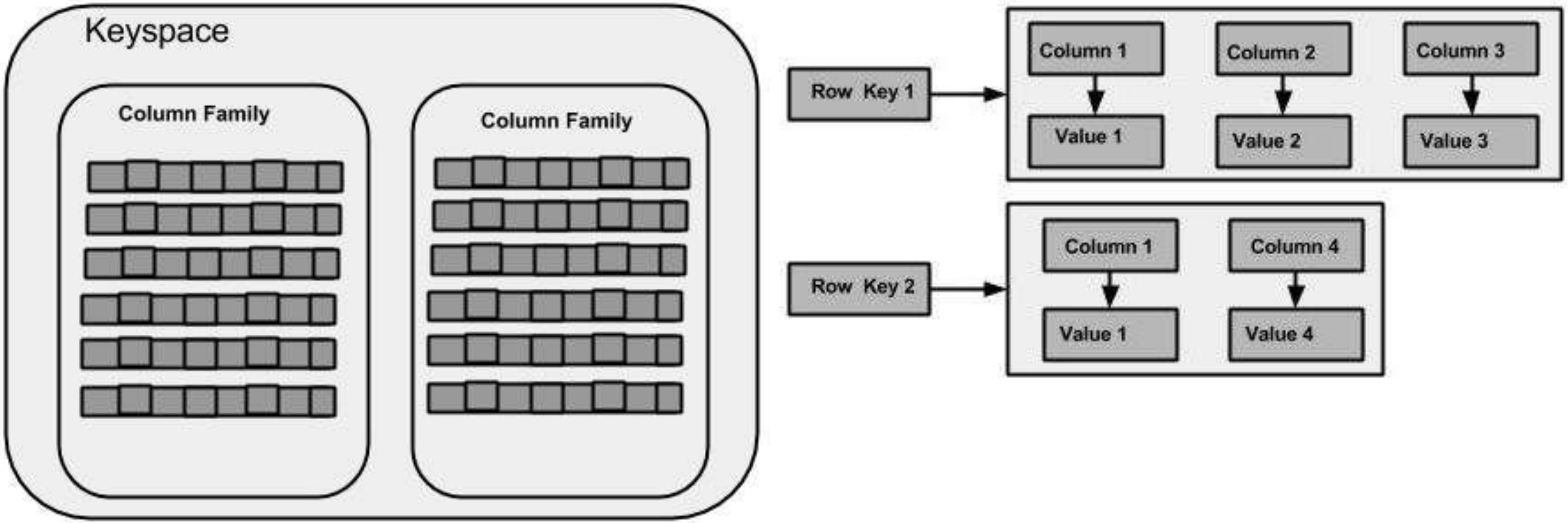
Cassandra

Most similar to a RDBMS

Cassandra has table-like structures called **ColumnFamilies**

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

Cassandra



Keyspace
Column Family
Column Family
Row Key 1
Row Key 2
Column 1
Column 2
Column 3
Value 1
Value 2
Value 3
Column 1
Column 4
Value 1
Value 4

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)
```