Scalability

Dr Peadar Grant

April 8, 2024

Contents

1 Scalability

Scalability assesses how any service (electronic or otherwise) copes with increased or reduced demand:

- **cafe near a college** will be busy during term but may be quieter otherwise.
- **pub in a holiday village** will experience increased demand during the summer months.
- **takeaway near a stadium** will experience a peak demand on match days.
- **ticket sales website** experiences peak demands when major concerts are put on sale.

Many of these demand peaks and troughs are predictable, some are not:

1. Supermarkets sold out of toilet roll when the COVID-19 crisis broke.
2. Beer also sold out, but Corona was left on the shelf.

When a service load increases towards 100 %, externally-facing performance will degrade.

1.1 Cloud vs on-site / data centre scaling

Data centre environments: buy more hardware.

- Hard to justify for shorter peaks. (CapEx)

Cloud environments: can provision and de-provision resources and pay only for what you use. (OpEx)

- Responsiblity for scalability splits between us and provider and depends on the service in question.
- Can automate the scaling operation depending on a metric.
- Ease and cost dependent on the architecture of your system.

1.2 Platform services

PaaS (e.g. S3, SNS, SQS, DynamoDB, Lambda) are scaled as needed by provider.

- No natural barrier at 100 % capacity
- Generally don't see any scaling operations - the service just expands as we use it.
- Unexpected high bills possible. Must monitor.

1.3 Infrastructural services

- Some parts of IaaS like VPC, EFS also scaled as needed.
- Infrastructural services (EC2, RDS) are our own responsibility to scale.
- This means that we will have to identify opportunities for scaling, select the most suitable methods and automate if required.

There is rarely a “right answer” when it comes to scaling.

2 Patterns

There are two key scaling methods, Figure 1.

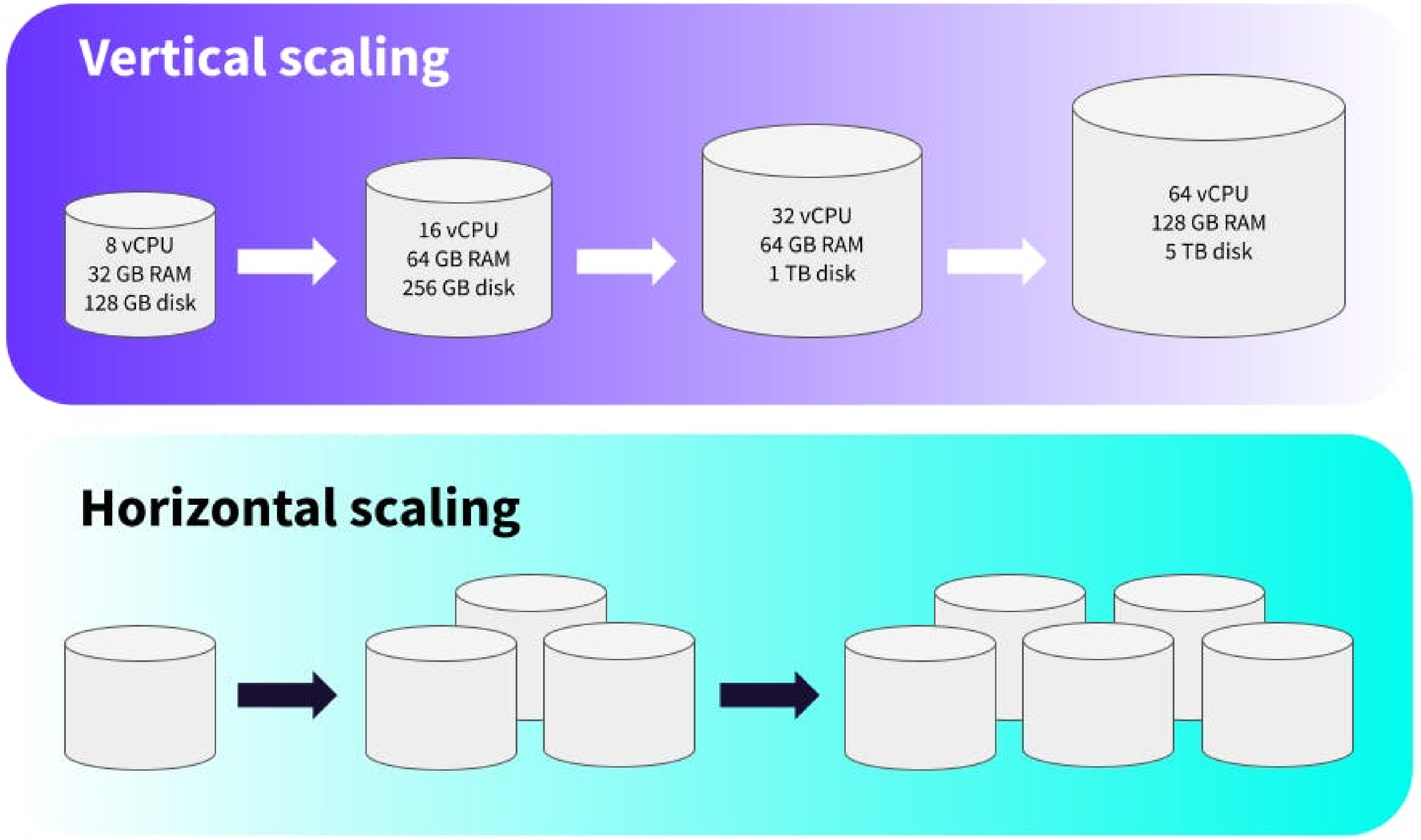

Figure 1: Horizontal vs vertical scaling

Vertical (scale up) is where individual resource units are increased / decreased in capacity to meet demand.

- Does not require architectural changes to system.
- Scaling operations normally require an interruption in service.

Horizontal (scale out) is where additional resource units are added / removed from a pool to meet demand.

- Scaling operations can take place without interrupting service.
- Requires architectural changes to the system to accomodate.

2.1 Practicalities

- System architecture choices will have a strong influence on ease of scaling.
- Horizontal scaling easier if each instance independent of others.
- Some parts of a system can't be horizontally scaled and need to be vertically scaled.
- Easiest to scale when an application is decomposed into a number of parts/services which are then scaled appropriately according to the load on that component.

3 EC2 scaling

A server machine has a number of different indicators of its performance. These include CPU load, RAM utilisation, hard disk space and network throughput. In this treatment we will focus on CPU & RAM capacity.

3.1 Before scaling

Before scaling any system / application (whether in the cloud or on a physical server), we should:

1. address inherent bottlenecks either in code (if available) or configuration.
2. determine if the application is primarily hitting the limits on CPU, RAM or disk/network throughput.
3. decompose system into consituent parts, which runn on separate EC2 instances.
4. consider replacing IaaS components with PaaS

3.2 Vertical scaling

Vertical scaling mainly involves increasing CPU/RAM capacity of an EC2 instance:

- Key advantage is that it doesn't need any changes to the application or its configuration. Almost all applications can be vertically scaled (up to some limit).

There are a number of downsides to this traditional scaling approach:

- Will involve downtime. This might be short (minutes to an hour) on a cloud service.
- Cannot be done easily continuously on-demand, may be paying usage fees for spare capacity.

3.3 Horizontal scaling

Horizontal scaling involves sharing the load among a pool of EC2 instances:

- Instances can be added/dropped from the pool as needed.
- All instances must (normally) be identical. Either clone or use `cloud-init`.
- Some form of *load balancing* will be needed to divide work:

  Implicit: where the architecture itself will provide a form of load balancing.

  Explicit: where we need to add/configure load balancing components. (Possible charges!)

3.4 Benefits of horizontal scaling

- Infinite scaling possible on a practical level.
- Instances can be added/removed without affecting other instances.
- Can span multiple AZ and regions for redundancy: combined scaling and high-availability.
- Can update software across pool of instances gradually without downtime.

4 Simple scaling example: queue processing

Imagine we have a batch process to run on data that is dropped into S3:

- This could be facial recognition, sales analytics, auto subtitling, translation etc.
- We use an S3 event to send a message to an SQS queue when a new object is added.
- None of the jobs depend on any other.

On an EC2 instance, a program runs to poll the SQS queue and process the data, which takes some time for each job.

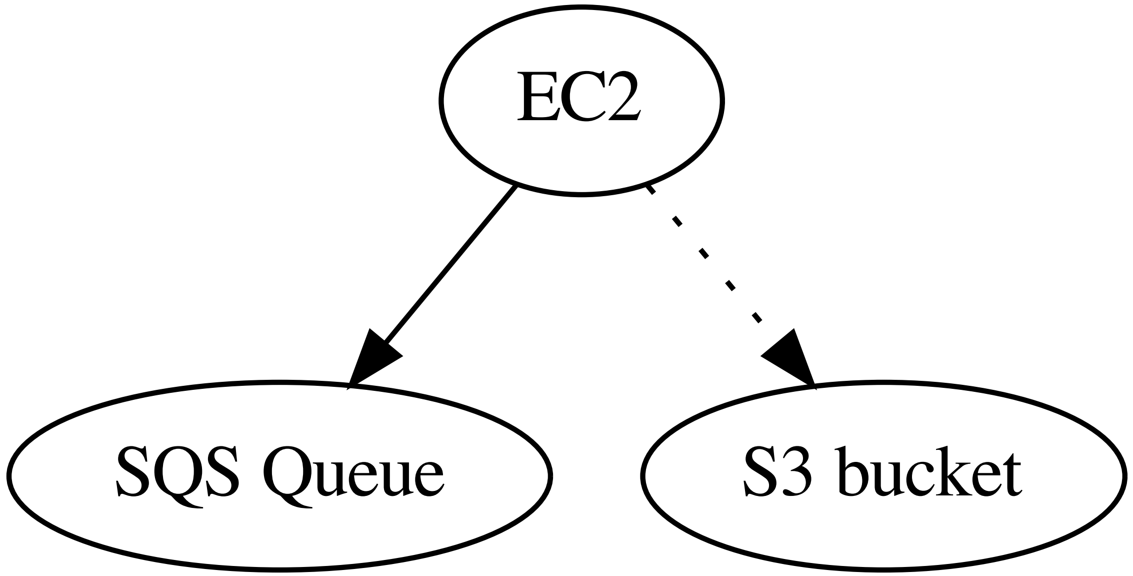

Figure 2: Single worker processing jobs from SQS queue

We could easily make identical EC2 instances to share the workload, Figure 3.

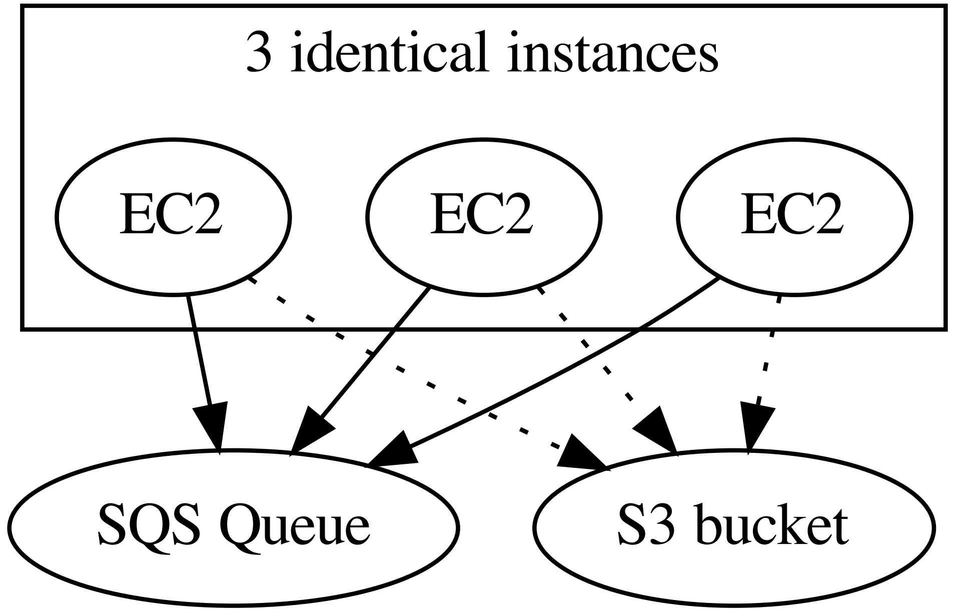

Figure 3: EC2 instances processing jobs from SQS queue

5 Auto scaling

Horizontal scaling can easily be automated.

Auto-scaling group: A group of EC2 VMs sharing a workload.

Auto-scaler: component to manage auto-scaling group

5.1 EC2 instances

Each EC2 instance is assumed to be identical:

- Each EC2 VM is cloned from a master image or can be setup with cloud-init.
- It is critical when using auto-scaling that the machine setup is entirely automated.
- There must be **no** manual setup needed on any instance.

5.2 Metrics

The auto scaling group will use a **metric** to decide if it needs to add or drop instances:

- Usually average CPU utilisation,
- Could be any other metric, like number of items in a queue,

These metrics are sent to Amazon CloudWatch, which generates auto-scaling events to add/destr
EC2 instances as required.

5.3 Capacity adjustments

5.3.1 Adding additional capacity

When the load exceeds a threshold for a defined period of time, the autoscaler will start up a new EC2 instance by cloning an AMI and/oror using cloud-init.

There will normally be a delay before the autoscaler adds any new machines.

5.3.2 Removing excess capacity

When the average CPU loads drops below a lower threshold for a defined period of time, the autoscaler will shutdown one EC2 in the group.

This will normally send a shutdown signal to the machine, similar to pressing the power button on a PC.

The EC2 is destroyed once terminated.

5.4 Spot instances

Depending on the workload, scaling can use EC2 Spot Instances.

This is like NightSaver electricity, when the EC2 spot price drops as overall demand falls.

For example, we may be running analytics that don't have to be realtime.

6 Load balancing

EC2 instances are often used to provide network services. To share an incoming load among a number of instances, we need to **load balance**.

6.1 Network load balancer

Load balancing can be done in on-site scenarios using a hardware load balancer. This obviously isn't an option in cloud environments.

AWS provide load balancers which are actually implemented as part of AWS software-defined networking that underlies VPCs.

Although the load balancer looks like a single-point-of-failure, it's actually implemented in a distributed way through the AWS infrastructure.

6.2 DNS-based

A second form of load-balancing can be implemented by having multiple IP addresses for a given hostname in DNS.

This is a form of *implicit* load balancing, as no additional hardware or software is needed.

6.3 Web application example

Simple web applications are relatively easy to scale. We simply add more EC2 instances and load balance the traffic among them, Figure 4.

In fact, this pattern will extend to many network service applications, not just web servers.

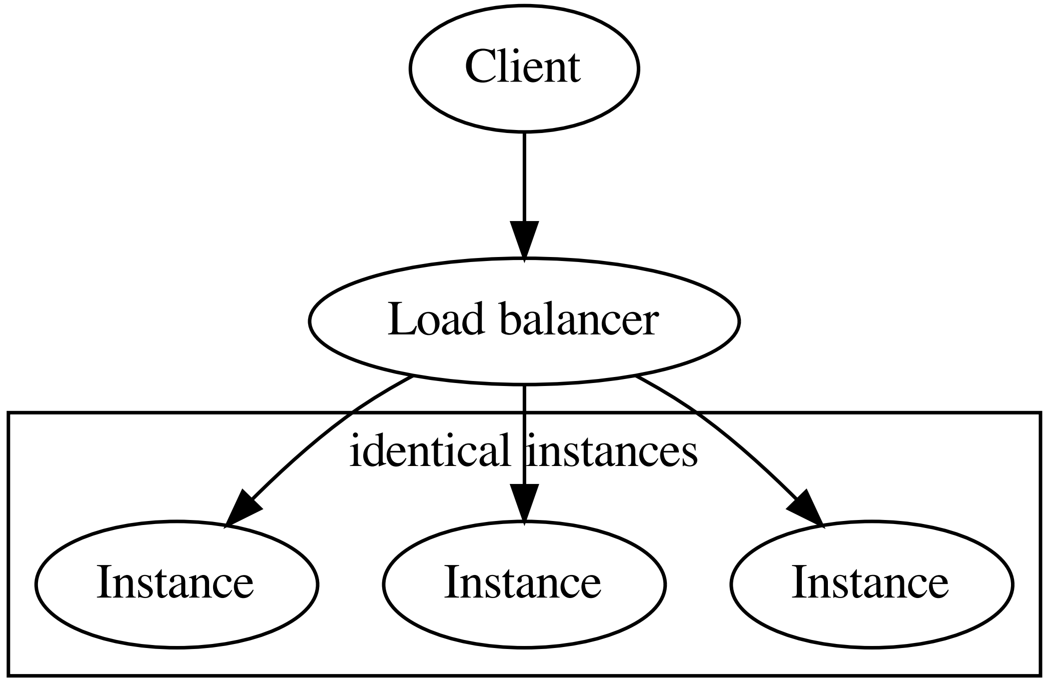

Figure 4: Scaling with load balancer

7 Service scalability

Many network services need to access other backend services, giving the pattern shown in Figure 5.

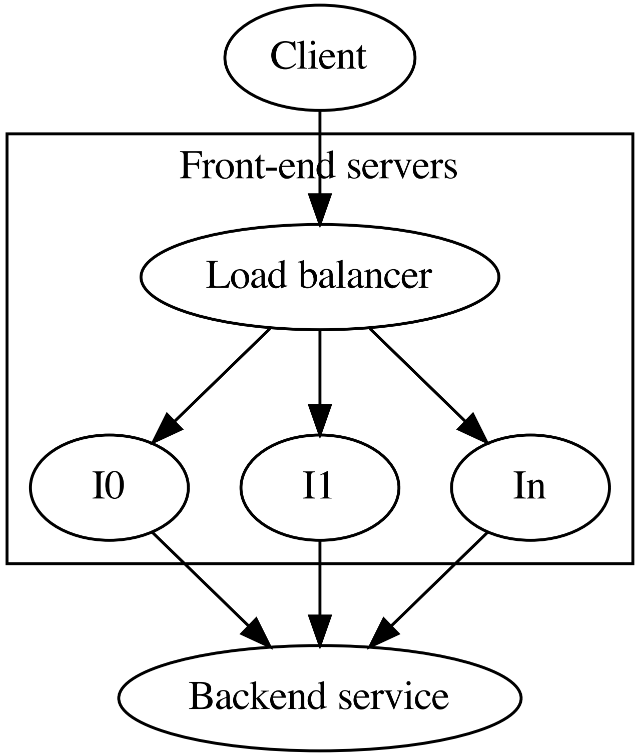

Figure 5: Load-balanced EC2 accessing database

7.1 Front-end and Back-end service examples

Back-end services might include Database, storage (S3 or EFS), message brokers, SMTP servers. Front-ends often seen using this pattern might include:

- Web application servers (e.g. PHP / Java)
- EC2-based SSH login servers for interactive usage (such as for data analytics, legacy UNIX terminal applications, teaching programming, business transaction processing)
- HTTP proxy caching servers accessing single HTTP server (very common pattern)

Obviously the front-end instances are likely to be accessing several different backend services.

7.2 Horizontal backend service scalability

The backend service itself needs to be able to handle the increased load:

- AWS platform services like SNS, S3, SQS, DynamoDB, Lambda will auto-scale on demand. You don't have to scale them.
- Some backend services, like SMTP servers can be simply scaled horizontally.
- Other backend services, particularly relational databases, need special consideration to make them scale horizontally.

7.3 Backend service scaling

Many backend services can themselves be horizontally scaled, Figure 6.

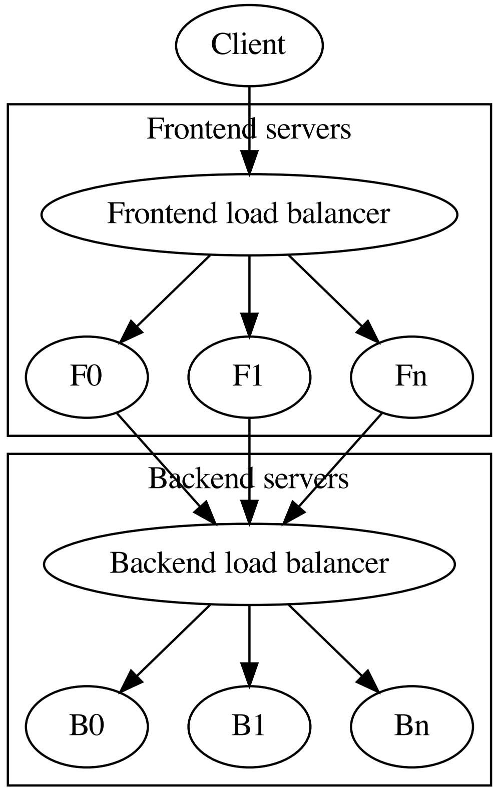

Figure 6: Backend scaling

8 Relational database scaling

In general, relational databases cannot be scaled by simply adding more instances. Data must be available on all nodes maintaining ACID-compliance. This limits our options to:

- Vertically scale the relational database. This will eventually hit a limit, and may incur large cost due to over-provisioning.
- Split the relational database workload between one master server and multiple slave servers:
  - Master database takes all write traffic and scales vertically.
  - Slaves handle read-traffic and scale horizontally.
  - Utilise built-in database replication to provide read slave replicas.
  - This method will work best when the read query load greatly exceeds the write load. Luckily, this is the case for many applications.

8.1 Replication

Replication is a feature available on most common relational database engines (MySQL, Oracle, PostgreSQL). A *slave* follows the *master* database.

8.2 Usage

We can connect to the slave database and use it as we would the master, except that:

- Read-only queries will proceed as normal.
- Writes can be made only to the master. Slaves will reject them.
  - Our application / system must be able to maintain two separate database connections and direct queries appropriately.

8.3 Synchronicity

Synchronous replication will not return from a `COMMIT` until the transaction has been replicated to the slave DB.

Asynchronous replication will return immediately from a `COMMIT` once the transaction has been committed on the master DB as normal. The transaction will be replicated to the slave some time later. (This may be very short!)

8.4 Multi-master replication

Some database engines do in in limited circumstances permit multi-master replication. Most will not.

For those that do, it usually breaks one of the ACID principles, and only makes sense in limited circumstances.

8.5 Control

Control of replication depends on the database engine we're using.

Some databases have the slave initiate the replication connection to the master.

In others, the master initiates the connection to the slave.

8.6 Disconnection

Normally a slave can “catch up” if disconnected from the master for some time.

For example, if a new slave is brought online it will be able to go from no data up to being consistent with the mater.

Often however we use other methods to “seed” a slave with a copy of the master DB.

8.7 Scaling with read replicas

Figure 7 shows a horizontally-scaled front-end connecting to a master/slave backend service.

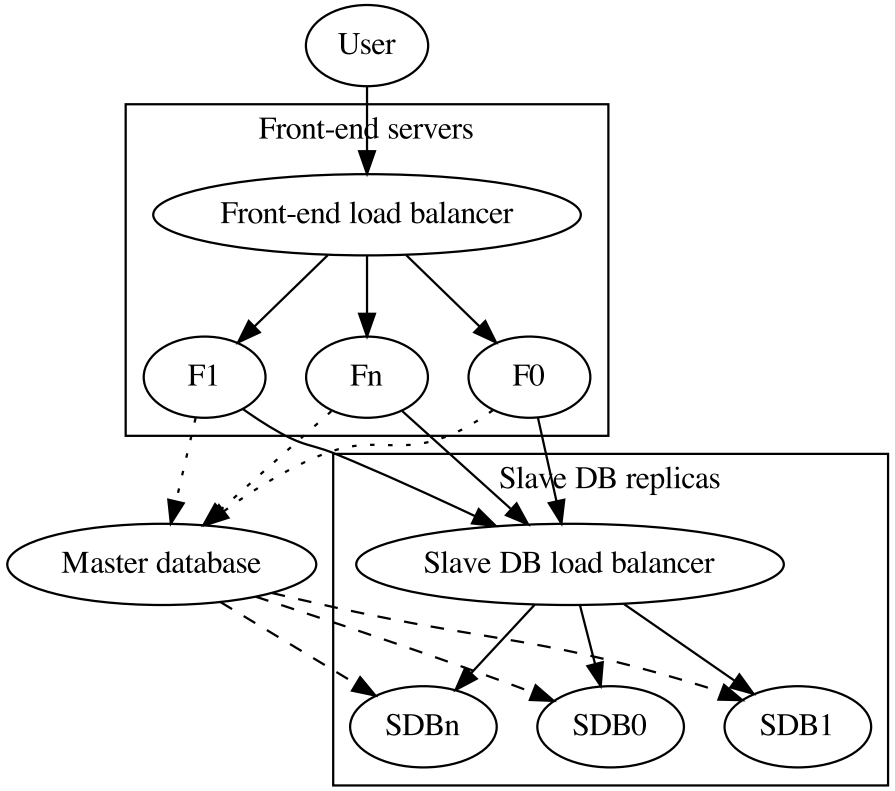

Figure 7: Horizontal scaling with read replicas

Looking at the diagram:

- Front-end servers maintain *two* database connections each: to both slave (solid line) and master DB (dotted line).
- Traffic from front-end servers to slave DB servers is load balanced separately, but in the same way as front-end.
- Replication traffic (dashed line) from master to slaves.
- Master DB is scaled vertically to accomodate write traffic.

8.8 Cascading replication

Can also have cascading replication schemes, where the master is replicated onto a slave which in turn is replicated onto other slaves.

This reduces replication traffic burden on the master.

9 Stateful web applications

HTTP is primarily a stateless protocol. Each web page (or other resource like a javascript, image or CSS) file is separately obtained by the browser:

Request is sent from the browser to the web server for the given resource.

Response returns the resource (and accompanying header metadata) when requested by the client.

In this simple stateless model, horizontal scaling works perfectly.

9.1 Maintaining state

Many web applications now require a server to be able to maintain client state across multiple requests (e.g. shopping basket):

- This involves issuing the client a *cookie* that the client then presents during subsequent requests.
- The web server then maintains a mapping from the cookie to a server-side data structure.
- The exact implementation of this varies depending on the server-side programming language and framework.

Once horizontal scaling is employed, an immediate problem has the potential to develop:
How do we ensure that a client's session is maintained across multiple requests when a load balancer is employed?

9.2 Sticky load balancing

We can configure the load balancer to “stick” a particular client to a server once the initial connection has been made:

- This requires no application or server changes.
- Need to tune the load balancer so that load is distributed effectively, and clients not left hanging if server fails.

In general, sticky load balancing is a very easy to setup solution.

9.3 Shared session state

The alternative to sticky load balancing is maintaing a shared session state across all instances. There are many ways we can do this, but two common methods are:

9.4 In-database session

where an application already uses a database, the session data can instead be placed there.

- This will normally cause a large load on the DB server.
- Requires application or server configuration change (difficulty depends on specific language, framework and application server).

9.5 Shared session store

As shared session store is a simple, high-performance, key-value storage server shared across all instances:

- This data doesn't need to be backed up and can be often stored in RAM only.
- Examples include: Redis, Memcached.
- Requires application or server configuration change (difficulty depends on specific language, framework and application server).
- Represents a potential single-point-of-failure
- Itself needs to be scaled, ideally as a platform service so we don't deal with the scaling ourselves.

Figure 8: Shared session store

9.6 Comparison

In general, I recommend using sticky load balancing first, and then trying these approaches if that doesn’t suit:

For example, clients remaining logged in for a long time.

10 Auto scaling with load balancing

10.1 Adding instances

When the load exceeds a threshold for a defined period of time, the autoscaler will:

1. Start up a new EC2 instance by cloning an AMI and/oror using cloud-init.
2. Wait for the new machine to start up and the required services to start. This may be time-based or function by sending requests to the machine.
3. The load balancer will be then configured to send a portion of traffic to the machine.
4. The auto scaler will wait for the average CPU load to drop as a result of the new machine for a fixed time delay period before adding more instances.

10.2 Removing instances

When the average CPU loads drops below a lower threshold for a defined period of time, the autoscaler will:

1. Instruct the load balancer to stop sending requests to the VM that will be terminated.
2. Optinally wait until all active connections to the machine have been closed, or more usually wait for a short time delay.
3. Shutdown the VM and remove it.
4. The autoscaler will wait for the average CPU load to rise a result of removing the VM for a fixed time delay period before removing more machines.