**Part 1 - Clones**

Public servers of a scalable web service are hidden behind a load balancer.  This load balancer evenly distributes load (requests from your users) onto your group/cluster of  application servers. That means that if, for example, user Steve interacts with your service, he may be served at his first request by server 2, then with his second request by server 9 and then maybe again by server 2 on his third request.

Steve should always get the same results of his request back, independent what server he  “landed on”. That leads to the first golden rule for scalability: every server contains exactly the same codebase and does not store any user-related data, like sessions or profile pictures, on local disc or memory.   
  
Sessions need to be stored in a centralized data store which is accessible to all your application servers. It can be an external database or an external persistent cache, like Redis. An external persistent cache will have better performance than an external database. By external I mean that the data store does not reside on the application servers. Instead, it is somewhere in or near the data center of your application servers.   
  
But what about deployment? How can you make sure that a code change is sent to all your servers without one server still serving old code? This tricky problem is fortunately already solved by the great tool Capistrano. It requires some learning, especially if you are not into Ruby on Rails, but it is definitely both the effort.

After “outsourcing” your sessions and serving the same codebase from all your servers, you can now create an image file from one of these servers (AWS calls this AMI - Amazon Machine Image.) Use this AMI as a “super-clone” that all your new instances are based upon. Whenever you start a new instance/clone, just do an initial deployment of your latest code and you are ready!

**Part 2 - Database**

After following [Part 1](https://www.lecloud.net/post/7295452622/scalability-for-dummies-part-1-clones) of this series, your servers can now horizontally scale and you can already serve thousands of concurrent requests. But somewhere down the road your application gets slower and slower and finally breaks down. The reason: your database. It’s MySQL, isn’t it?

Now the required changes are more radical than just adding more cloned servers and may even require some boldness. In the end, you can choose from 2 paths:

**Path #1** is to stick with MySQL and keep the “beast” running. Hire a database administrator (DBA,) tell him to do master-slave replication (read from slaves, write to master) and upgrade your master server by adding RAM, RAM and more RAM. In some months, your DBA will come up with words like “sharding”, “denormalization” and “SQL tuning” and will look worried about the necessary overtime during the next weeks. At that point every new action to keep your database running will be more expensive and time consuming than the previous one. You might have been better off if you had chosen Path #2 while your dataset was still small and easy to migrate.

**Path #2** means to denormalize right from the beginning and include no more Joins in any database query. You can stay with MySQL, and use it like a NoSQL database, or you can switch to a better and easier to scale NoSQL database like MongoDB or CouchDB. Joins will now need to be done in your application code. The sooner you do this step the less code you will have to change in the future. But even if you successfully switch to the latest and greatest NoSQL database and let your app do the dataset-joins, soon your database requests will again be slower and slower. You will need to introduce a cache.

**Part 3 - Cache**

After following [Part 2](http://www.lecloud.net/post/7994751381/scalability-for-dummies-part-2-database) of this series, you now have a scalable database solution. You have no fear of storing terabytes anymore and the world is looking fine. But just for you. Your users still have to suffer slow page requests when a lot of data is fetched from the database. The solution is the implementation of a cache.

With “cache” I always mean in-memory caches like [Memcached](https://href.li/?http://memcached.org/) or [Redis](https://href.li/?http://redis.io/). Please **never do file-based caching**, it makes cloning and auto-scaling of your servers just a pain.   
  
But back to in-memory caches. A cache is a simple key-value store and it should reside as a buffering layer between your application and your data storage. Whenever your application has to read data it should at first try to retrieve the data from your cache. Only if it’s not in the cache should it then try to get the data from the main data source. Why should you do that? Because **a cache is lightning-fast**. It holds every dataset in RAM and requests are handled as fast as technically possible. For example, Redis can do several hundreds of thousands of read operations per second when being hosted on a standard server. Also writes, especially increments, are very, very fast. Try that with a database!  
  
  
There are 2 patterns of caching your data. An old one and a new one:  
 **#1 - Cached Database Queries**That’s still the most commonly used caching pattern. Whenever you do a query to your database, you store the result dataset in cache. A hashed version of your query is the cache key. The next time you run the query, you first check if it is already in the cache. The next time you run the query, you check at first the cache if there is already a result. This pattern has several issues. The main issue is the expiration. It is hard to delete a cached result when you cache a complex query (who has not?). When one piece of data changes (for example a table cell) you need to delete all cached queries who may include that table cell. You get the point?

**#2 - Cached Objects**  
That’s my strong recommendation and I always prefer this pattern. In general, see your data as an object like you already do in your code (classes, instances, etc.). Let your class assemble a dataset from your database and then store the complete instance of the class or the assembed dataset in the cache. Sounds theoretical, I know, but just look how you normally code. You have, for example, a class called “Product” which has a property called “data”. It is an array containing prices, texts, pictures, and customer reviews of your product. The property “data” is filled by several methods in the class doing several database requests which are hard to cache, since many things relate to each other. Now, do the following: when your class has finished the “assembling” of the data array, directly store the data array, or better yet the complete instance of the class, in the cache! This allows you to easily get rid of the object whenever something did change and makes the overall operation of your code faster and more logical.  
  
And the best part: it makes asynchronous processing possible! Just imagine an army of worker servers who assemble your objects for you! The application just consumes the latest cached object and nearly never touches the databases anymore!  
  
Some ideas of objects to cache:

* user sessions (never use the database!)
* fully rendered blog articles
* activity streams
* user<->friend relationships

As you maybe already realized, I am a huge fan of caching. It is easy to understand, very simple to implement and the result is always breathtaking. In general, I am more a friend of Redis than Memcached, because I love the extra database-features of Redis like persistence and the built-in data structures like lists and sets. With Redis and a clever key’ing there may be a chance that you even can get completly rid of a database. But if you just need to cache, take Memcached, because it scales like a charm.  
  
Happy caching!

**Part 4 - Asynchronism**

This 4th part of the [series](http://www.lecloud.net/tagged/scalability) starts with a picture: please imagine that you want to buy bread at your favorite bakery.  So you go into the bakery, ask for a loaf of bread, but there is no bread there! Instead, you are asked to come back in 2 hours when your ordered bread is ready. That’s annoying, isn’t it?  
  
To avoid such a “please wait a while” - situation, asynchronism needs to be done.  And what’s good for a bakery, is maybe also good for your web service or web app.

In general, there are two ways / paradigms asynchronism can be done.   
 **Async #1**Let’s stay in the former bakery picture. The first way of async processing is the “bake the breads at night and sell them in the morning” way. No waiting time at the cash register and a happy customer.  Referring to a web app this means doing the time-consuming work in advance and serving the finished work with a low request time.  
  
Very often this paradigm is used to turn dynamic content into static content.  Pages of a website, maybe built with a massive framework or CMS, are pre-rendered and locally stored as static HTML files on every change. Often these computing tasks are done on a regular basis, maybe by a script which is called every hour by a cronjob. This pre-computing of overall general data can extremely improve websites and web apps and makes them very scalable and performant. Just imagine the scalability of your website if the script would upload these pre-rendered HTML pages to AWS S3 or Cloudfront or another Content Delivery Network! Your website would be super responsive and could handle millions of visitors per hour!  
 **Async #2**Back to the bakery. Unfortunately, sometimes customers has special requests like a birthday cake with “Happy Birthday, Steve!” on it. The bakery can not foresee these kind of customer wishes, so it must start the task when the customer is in the bakery and tell him to come back at the next day. Refering to a web service that means to handle tasks asynchronously.  
  
Here is a typical workflow:  
  
A user comes to your website and starts a very computing intensive task which would take several minutes to finish. So the frontend of your website sends a job onto a job queue and immediately signals back to the user: your job is in work, please continue to the browse the page. The job queue is constantly checked by a bunch of workers for new jobs. If there is a new job then the worker does the job and after some minutes sends a signal that the job was done. The frontend, which constantly checks for new “job is done” - signals, sees that the job was done and informs the user about it. I know, that was a very simplified example.   
  
If you now want to dive more into the details and actual technical design, I recommend you take a look at the first 3 tutorials on the [RabbitMQ website](https://href.li/?http://www.rabbitmq.com/). RabbitMQ is one of many systems which help to implement async processing. You could also use [ActiveMQ](https://href.li/?http://activemq.apache.org/) or a simple [Redis list](https://href.li/?http://redis.io/topics/data-types). The basic idea is to have a queue of tasks or jobs that a worker can process. Asynchronism seems complicated, but it is definitely worth your time to learn about it and implement it yourself. Backends become nearly infinitely scalable and frontends become snappy which is good for the overall user experience.   
  
If you do something time-consuming, try to do it always asynchronously.