To begin with, we need to be clear on our requirements. The system needs to be overbuilt for peak load with some level of margin. So, with an expected peak QPS of 800 I would expect my system to handle at least +25% or **1000 requests per second.**

Let's assume we start with a database server, with a Postgres instance. Postgres is free, it's open-source, extensible, and has best of breed geo-spatial features out of the box (cube/earthdistance and postgis). I choose it for those reasons, and for familiarity (which means productivity).

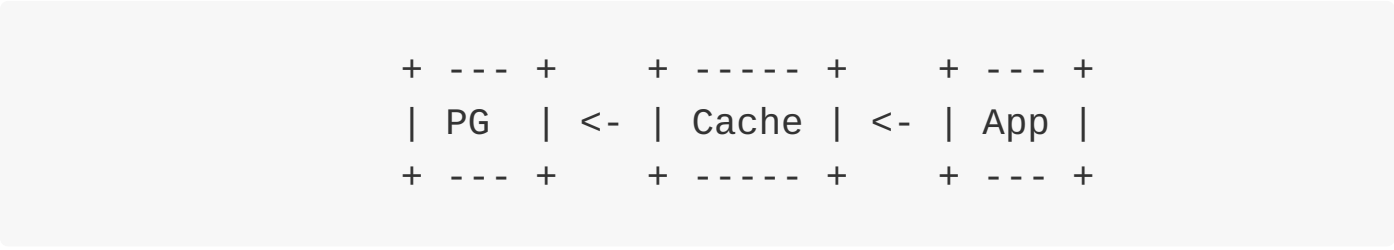
We also have a single application server running a single Flask instance (Flask is lightweight and performant, perfect for a simple API like we have here). The hosting service isn't too important, but we need to be able to fully manage our servers, so full access via AWS or Digital Ocean is preferred. Let's assume we use Digital Ocean for simplicity.

The general path is to scale your application out and your database up until you need a different strategy. The question here is, what is the bottleneck going to be? The application or the database?

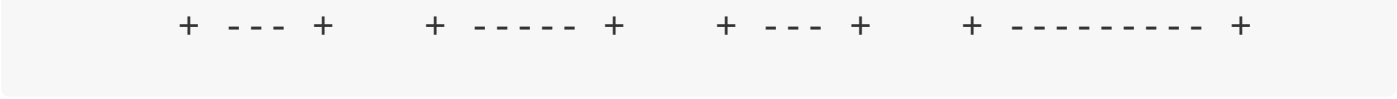
It is hard to say without benchmarks, and system monitoring but I will assume we won't need to do anything more than scale up the hardware on our database server. We will want to make sure to have enough memory so that we don't have to hit the disk on query and at a minimum keep all indexes in cache. 1000 QPS likely isn't an issue for PG. The situations spelled out in P2-P4 will be computing differences between lat/long and then doing lookups. I/O (time on the wire between the app server and the db) is almost certaintly going to be the bottleneck.

How can we decrease latency? The first low-hanging fruit is too make sure our app server is in the same datacenter as our DB server to reduce that round trip. We may want to do some simple indexing the on most frequently looked up columns (lat/long), this won't decrease that latency but we can optimize lookup. Beyond that, if we need to reduce latency further we can implement a caching scheme.

We could use a cache-aside or cache-through scheme because, as caching theory supposes, the most likely lookups are the lookups that have been already made. For popular metropolitan areas this is probably true, so there is likely a performance gain to be had with some kind of caching scheme for computed values. Regardless, we will still have to perform a significant amount of fresh computation. Let's assume 50% of the time we have a cache-miss and need to query the database.



Let's assume, that it is still too slow, and our application instance is waiting on I/O. What can we do? First, we optimize hardware by scaling up application instances, then we scale out to multiple servers. Once, we have multiple servers we could implement some kind of reverse proxy load balancing scheme. With Digital Ocean and AWS we can get this as a built in feature with some setup cost or we could implement it from 'scratch' with Nginx.



I believe this is all that is necessary at this QPS, but there also strategies to scale out the DB, if necessary, as the user base grows. I'll also mention, that if we are concerned about handling QPS we should throttle greedy/malicious IPs so if it does happen that we approach our capacity we won't degrade performance for all users due to a single greedy user.