**Part 1**

**Using Postgres as the source:**

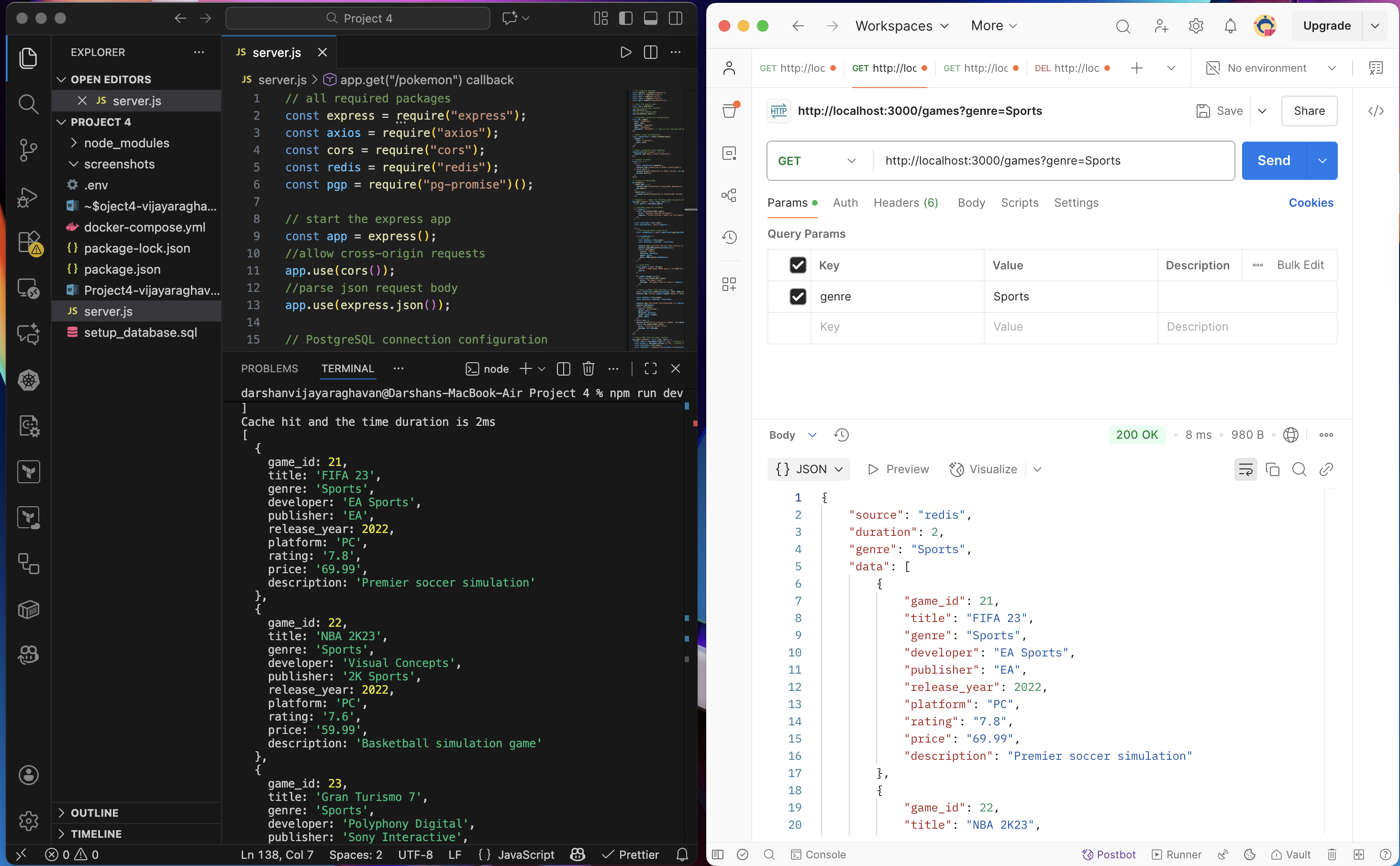
I used a games database and the /games endpoint with a genre query parameter to conduct this experiment. The genre can take values of strings, such as Action, Adventure, RPG, Strategy, and Sports. It first attempts to retrieve the values from Redis. If the query results are not stored in Redis, it then makes a call to the PostgreSQL database to obtain the values. Both Redis and Postgres are run as Docker containers. The Postgres container on startup runs the SQL file to create the games database and inserts records for the genres above. The server runs on port 3000, so the API calls need to be made here.

First call to the games endpoint with genre Sports gives the results back from the PostgreSQL with a 10ms delay. (In VS Code)

A screenshot of a computer screen

AI-generated content may be incorrect.

Second call to the game’s endpoint with genre Sports gives the results back from the RedisCache with a 2ms delay, which is already 5 times faster than postgres.



**Source as a PUBLIC API:**

I am using the Pokémon public API for this purpose, since it does not require any API key. We can request Pokémon with this API, and it returns a list. We can select the number of Pokémon we need with the limit query param and an offset query param to get a new set of Pokémons with an offset. So, for the Redis cache key, we will have the limit and the offset values to store the results.

First call to the api, will make a call to the public API to fetch the results and store them in Redis with a key pokemon:limit${limit}:offset${offset}. The time taken is given below in VSCode in the terminal section. We get a response back after 481ms.

A screenshot of a computer screen

AI-generated content may be incorrect.

Second call to the api, will make a call to Redis to get the already stored values. The time taken is given below in VSCode in the terminal section. We get a response back after 2ms. This is 240 times faster than the public API call.

A screenshot of a computer screen

AI-generated content may be incorrect.

**Results:**

|  |  |  |
| --- | --- | --- |
| **Experiment** | **First Call(From Source)** | **Second Call( from Redis)** |
| Postgres DB call | 10ms | 2ms |
| Public Pokémon API | 481ms | 2ms |

**In case of PostgreSQL:**

As we can see, the Postgres DB call took 10ms to fetch the results, while fetching from Redis using a key took a very short amount of time, which is 2ms.

This is because the process of calling the postgresDB involves several delays getting added to the time. In the Cache miss scenario, User makes a call for an object, we check the Redis cache first to check for the object, and we don’t find it, we open a TCP connection with the Postgres DB and then Postgres DB parses the query, selects the best plan etc., and have to move the data from the disk to the buffer(Disk I/O time) which all takes a considerable amount of time. And then the result must be serialized and deserialized for it to be sent to the sockets, which adds some more delay. Thus, all these take 5 times more time than just checking Redis and returning the object values for a key.

In case of a Cache HIT from Redis, the User makes a request, goes to the Node server, and Redis fetches the data from in-memory for a key and returns the cached data using a simple O(1) lookup operation, which is much faster than the former. There is no Disk I/O, no table scan, lookup, sorting, or filtering, since data is already in the form of JSON, no need for further processing. Cutting all these unwanted delays results in faster processing.

In our case, the DB and the Redis are in the local network, so we don’t have network latency. Here, the Disk I/O, query processing, and connection overhead are the main bottlenecks.

**In case of a public API:**

In a Cache MISS, we first hit our Node.js server, we check the cache, and we do not have the result stored. So, we now must make a call to the public REST API. We make a DNS lookup for the public API, and we make a TCP/TLS connection with the PokeAPI server over the internet. The Poke API makes a call to internally process the query. It makes a call to its database and then returns a response over the internet, which adds much more network latency based on where the application is hosted. We might make multiple hops over the network to deliver the response. All these add latency and thus the very large value.

In the case of a Cache HIT, it is a similar process to what we described with the cache in the previous experiment. We still check for the Redis cache on localhost, and Redis does the O(1) hash lookup and returns the values, which are all stored in memory. Thus, it is blazing fast, and since the process is the same, it takes the same 2ms time delay for the entire process.

The main performance bottleneck here is the network latency, since the request has to go through the internet, it is adds to the delay, and this is the main difference between the public API request time and the Postgres DB request time.

**PART 2**

Using Redis-benchmark for benchmarking the Redis running in my Docker container. The server is using Redis version 7.4.7, in standalone mode for the arm64 process, which is in append-only file mode.

**Test cases:**

1. **General comparison of the different functions in REDIS:**

redis-benchmark -q -n 100000 : this tests for all functions within Redis for 100000 requests, and the -q outputs how many requests it can handle per second. In the below screenshot, we can see the main functions we use, like as Set, Get can handle

SET: 341296.91 requests per second

GET: 299401.22 requests per second

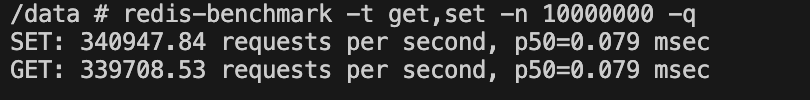
which is very fast, we can use this as a baseline.

A screenshot of a computer program

AI-generated content may be incorrect.

1. **Impact of the number of requests on the performance:**

redis-benchmark -t get,set -n 10000000 -q: this runs 10 million requests for the GET and SET functions, which are the most frequently used. Although the number of requests increased by 100 times, it is still at the same level of performance.



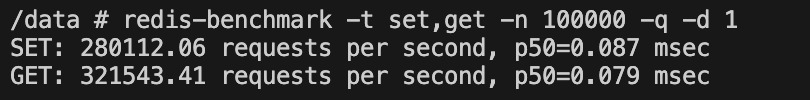
1. **How increase in Data Size decrease performance:**

Redis-benchmark -t set,get -n 100000 -q -d 10000. As we see below, the -d flag denotes the data size, which we increased to 10kb from the default 3 bytes, and we compare it with the data size of 1byte. An increase in the data size has clearly resulted in a decrease in performance, with the number of set operations per second in 10kb data size decreasing to more than half of that of the 1 byte data size, while the Get operation sees a 30% decrease in performance.

**10kb transfer:**



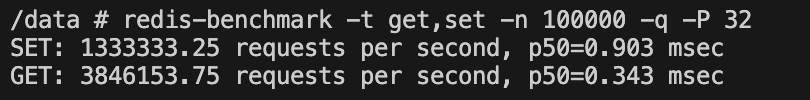
**1byte transfer:**



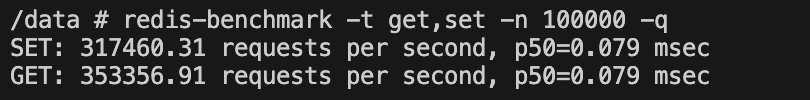
1. **How pipelining improves performance:**

We use the command redis-benchmark -t get,set -n 100000 -q -P 32 and compare it with no pipelining.

**With pipelining of 32 batches:**



**Without pipelining:**



We can see that with pipelining, the throughput increases in SET by 4 times, while with get, the throughput increases by 11 times. Thus, batching/pipelining helps in increasing the performance by increasing the throughput by multiple folds.

1. **Concurrent connections test**:

We compare how the number of concurrent users impacts the performance of the system. We run our benchmark for 10000 parallel connections and compare it with 1000 parallel users. We see a significant reduction in the number of requests per second, thus showing us that the increase in the number of parallel connections actually decreases the performance of the Redis server.

A black background with white text

AI-generated content may be incorrect.

1. **Latency test in Redis:**

We now test the average latency and the intrinsic latency to check for the raw performance of Redis. The average latency is 0.34 ms, and the latency history also confirms that this latency is stable for the server. The intrinsic latency is Redis’s internal processing time only, and as we can see, it is 35.64 nanoseconds, and the rest of the time is from Docker and local network latency. So Redis is not the bottleneck of our system, because it is capable of processing 841 million operations in 30 seconds.

A computer screen shot of a black screen

AI-generated content may be incorrect.