# interview-task

> First write a token generator that creates a file with 10 million random tokens, one per line, each consisting of seven lowercase letters a-z. Then write a token reader that reads the file and stores the tokens in your DB. Naturally some tokens will occur more than once, so take care that these aren't duplicated in the DB, but do produce a list of all non-unique tokens and their frequencies. Find a clever way to do it efficiently in terms of network I/O, memory, and time and include documentation inline with your code or as txt file, describing your design decisions.

## Back of the Envelope Calculation

- $10 \cdot 10^6 \cdot 7 = 7 \cdot 10^7$ total bytes, so roughly `66` MB. Should fit into main memory
- $|{a,\cdots,z}| = 26 \implies 26^7$ combinations. Thinking of the Birthday problem, there is a good chance that we will have a collision of two random strings.

## Implementation thoughts

### Go

1. For a simple implementation, a line by line read in should do the job.
    - Further io/performance improvement with `bufio`, but needs testing
2. Checking for duplicates

    > $n = 10^7$

    - A `Map[string]int` can do the job with low programming complexity, because access is O(1) and filling it is O(n)
        - We can even hint `make` with $n$ to reduce the number of map resizes
    - Radix sort with 7 iterations will be O(n * 7)
        - But should be very memory intensive
    - Counting sort will be O(n + 26^7) which does not look promising because of the possible huge input space

    $\implies$ https://www.youtube.com/watch?v=kVgy1GSDHG8

My goal is to do most of the work in the main memory after loading in the tokens from a file.

### Database

> I have never used Postgres, so I will give it a try. The lib (PGX)[https://github.com/jackc/pgx] seems to be up to date & maintained, so I'll use it.

**Schema**

We just have to save unique token. Documentation and Stackoverflow suggest using `character varying` to save the token. We do not need to mark the token with `PRIMARY KEY` because it implies `UNIQUE` which

we already check beforehand. In addition, filtering duplicates reduces the calls to insert rows into the database.

```
CREATE TABLE tokens(
  token VARCHAR(7) NOT NULL
)
```

## Changelog

### Issue

> We issue one `Insert` when a new token pops up. Therefore, we have for $n$ unique tokens $n$ calls to `conn.Exec`. Also, it seems that the call is blocking such that it should be put into a `go-routine`.

This is visible, if we compare the time spent with and without sql insertions:

```
2021/11/26 19:44:54 Start scanning file
2021/11/26 20:25:35 Finish scanning file
```

$>$ 30 minutes with SQL

```
2021/11/26 22:46:47 Start scanning file
2021/11/26 22:46:49 Finish scanning file
```

2 seconds without SQL

### Ideas

1. Issues the `Insert` in batches to reduce IO and increase speed
2. We can spin up multiple connections to the database

### Results

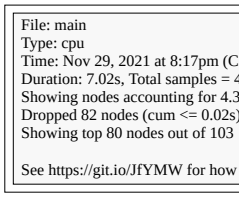
Implementing the first idea leads to a total runtime of roughly 4 seconds.

```
2021/11/27 00:39:43 Start scanning file
2021/11/27 00:39:43 Start writing to DB
2021/11/27 00:39:46 Finish scanning file
2021/11/27 00:39:46 Writing duplicates to file
2021/11/27 00:39:46 Finish writing to DB
```

However, `pooling` i.e. multiple connections did not increase performance. Therefore, batching solves our problem of slow db writes by reducing IO.

### Profiling

> Just wanted to test it out. Most of the time is spent in the hashmap.




## Setup

1. Be sure to have docker installed
2. Navigate to the main dir `cd interview-task`
3. Execute `setup.sh` which will pull a `postgres` image, adds a user and creates a table
4. Run `go run main.go`