Mejoras en el rendimiento del programa indexador

Daniel Esteban Marquez Upegui

Proyecto: https://github.com/DanilsGit/vue-go-zincsearch.git

Tecnologías usadas

Go - Zincsearch

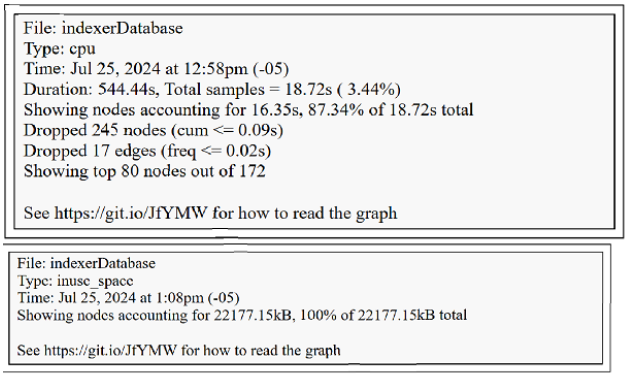
Colombia – Valle del Cauca

2024

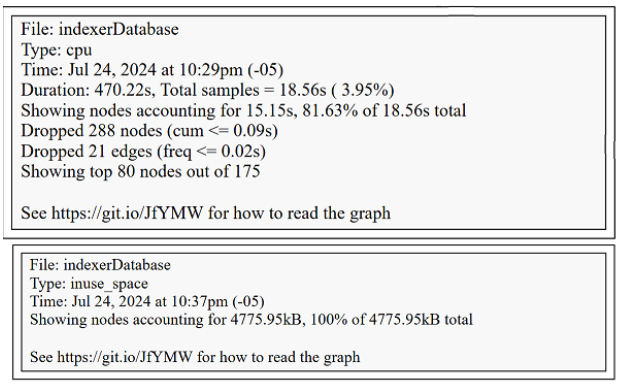
Inicialmente el programa recorría todos los archivos de la carpeta y cuando acumulaba un lote lo enviaba a zincsearch

```go
err := filepath.Walk(data, func(path string, info os.FileInfo, err error) error {
    // If there is an error, return it
    if err != nil {
        return err
    }

    // If the file is a directory, return nil
    if info.IsDir() {
        return nil
    }

    // Read the email file
    email, err := readEmailFile(path)
    if err != nil {
        fmt.Printf("Error reading email file %s: %v\n", path, err)
        return nil
    }

    emails = append(emails, email)
    if len(emails) >= batchSize {
        uploadToDatabase(emails)
        emails = nil
    }

    return nil
})
```

Esta opción es útil y se podía mejorar el tiempo cambiando el tamaño del lote

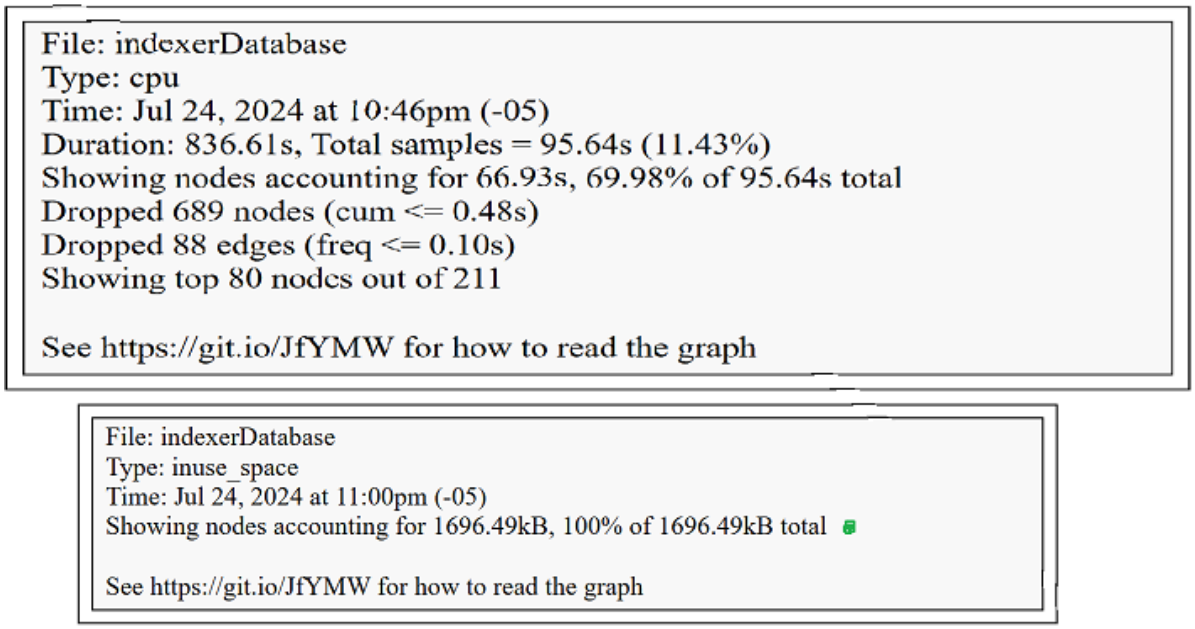


lote 5000

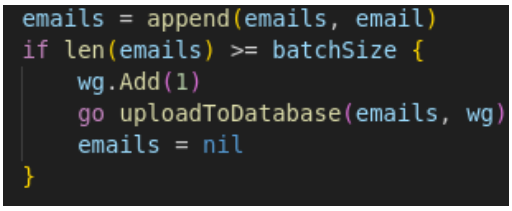


lote 2000

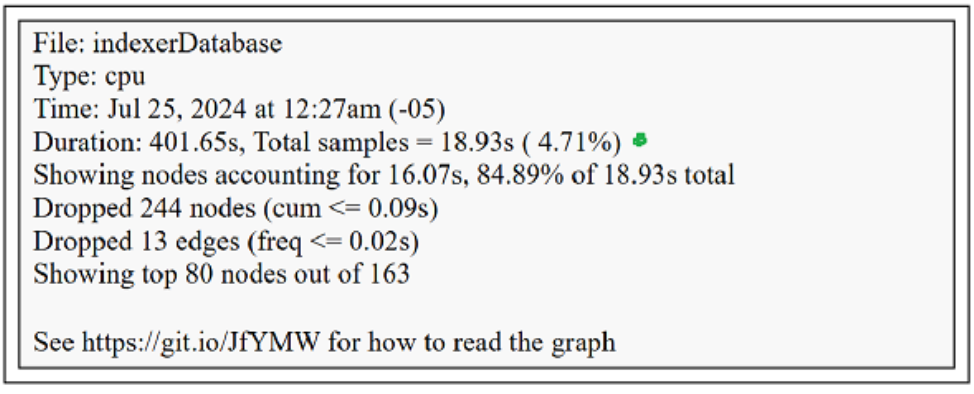
De forma experimental se decidió utilizar lotes de 1 unidad y sorprendentemente mostró el mejor uso de memoria en todas las pruebas de este documento. Sin embargo, en el tiempo de CPU fue el peor de todos.

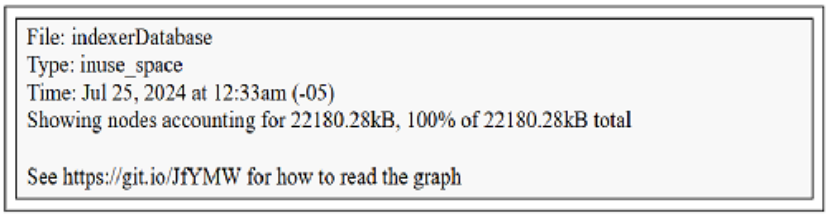


Una forma de mejorar el tiempo de CPU es seguir explorando mientras se envían los emails en segundo plano. Así que, con el mismo código y configurando goroutines y waitgroups se puede lograr una mejora del 26%.

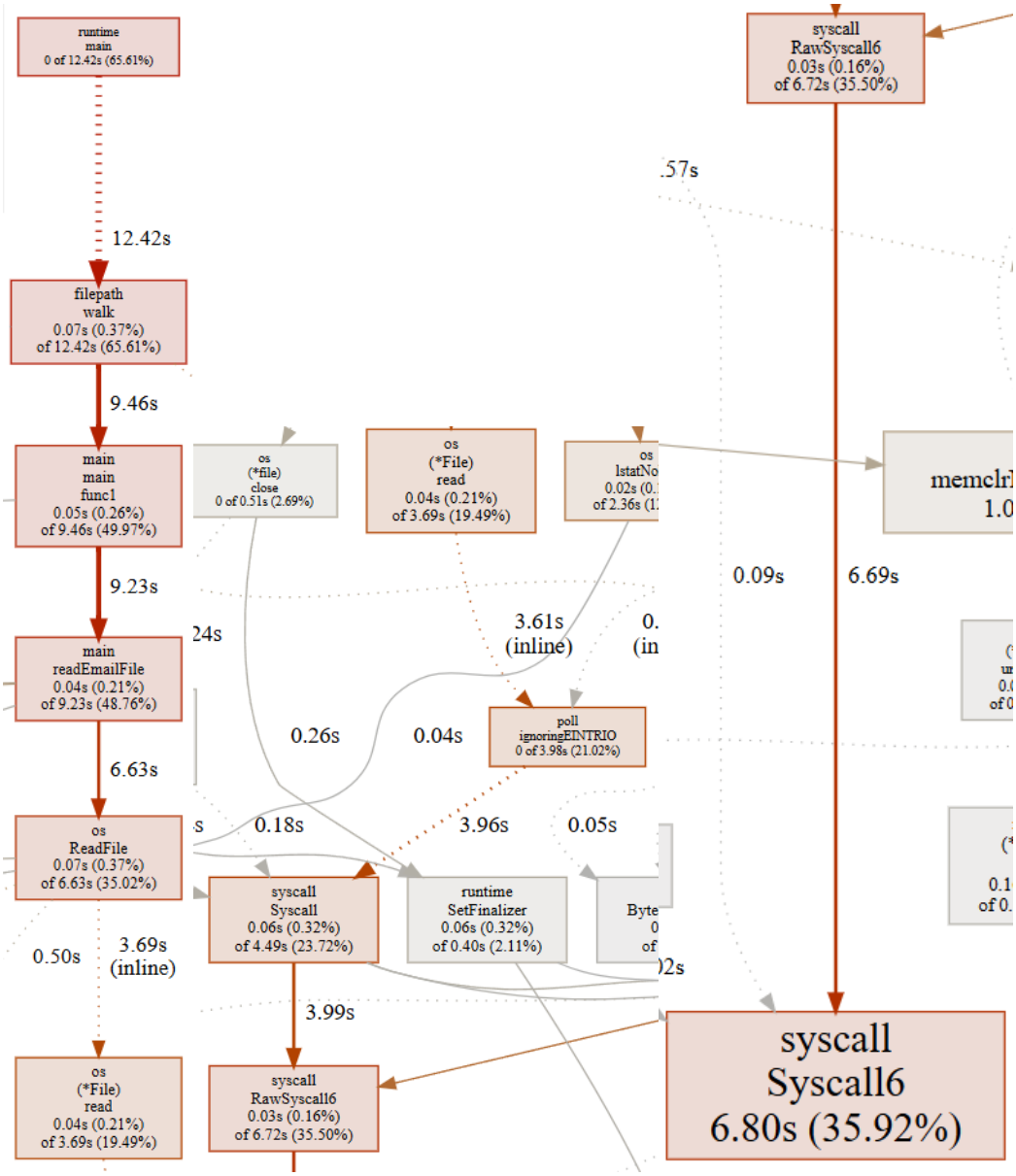


```
emails = append(emails, email)
if len(emails) >= batchSize {
    wg.Add(1)
    go uploadToDatabase(emails, wg)
    emails = nil
}
```





Para tener un mejor rendimiento es importante leer e interpretar los gráficos.



Parece ser que el recorrido y la lectura de archivos es lo que más está consumiendo tiempo de CPU. Una de las muchas formas de solucionarlo y la que se decidió implementar fue dividir en partes iguales el trabajo en cada goroutine.

```go
// Walk the directory and read the email paths
err := filepath.Walk(data, func(path string, info os.FileInfo, err error) error {
    // If there is an error, return it
    if err != nil {
        return err
    }

    // If the file is a directory, return nil
    if info.IsDir() {
        return nil
    }

    // Append the path to the pathEmails slice
    pathEmails = append(pathEmails, path)

    return nil
})
```

El código anterior es similar a los ya vistos. Pero, ahora se utiliza Walk sólo para adquirir la ruta donde están los emails. Posteriormente, se dividen los correos en partes casi iguales (dependiendo si la división posee residuo). Finalmente, se crean los rangos que se utilizarán para dividir el trabajo entre las goroutines.

```go
// Number of subdivisions
const Subdivisions int = 100
```

```go
rangeOfParts := utils.RangeOfParts(len(pathEmails), constants.Subdivisions)
```

```go
func RangeOfParts(number, initParts int) []int {

    // First | divide the number by the initial number of parts
    equalDivision := make([]int, initParts)
    // Calculate the division and the remainder
    part := number / initParts
    rem := number % initParts

    // For each part, set the value
    for i := 0; i < initParts; i++ {
        equalDivision[i] = part
    }

    // Add the remainder to the first parts
    for i := 0; i < rem; i++ {
        equalDivision[i]++
    }

    //  Second | create a response slice with the range of parts
    response := make([]int, len(equalDivision)+1)
    response[0] = 0
    for i := 1; i < len(equalDivision)+1; i++ {
        response[i] = response[i-1] + equalDivision[i-1]
    }
    return response
}
```

La primera parte del código crea un array descomponiendo el número en partes casi iguales. Por ejemplo, 10 en 3 partes = [4,3,3] ó 10 en 4 partes = [3,3,2,2]

La segunda parte del código crea un nuevo array con len+1 espacios donde están los rangos iniciando con 0. Por ejemplo, [4,4,2] = [0,4,8,10]

Continuando el ejemplo, las goroutines leerán de forma paralela el array de correos.
0 al 3, 4 al 7, 8 al 9.

```go
// For each part, read the emails
for i := 0; i < len(rangeOfParts)-1; i++ {
    wg.Add(1)
    go utils.ReadPathEmails(pathEmails, rangeOfParts[i], rangeOfParts[i+1]-1, wg)
}
```

Cada goroutine se encargará de leer su parte y subirla a la base de datos.

```go
// readPathEmails reads the emails from the paths
func ReadPathEmails(pathEmails []string, start int, end int, wg *sync.WaitGroup) {

    defer wg.Done()

    var emails []models.Email

    for i := start; i <= end; i++ {
        email, err := readEmailFile(pathEmails[i])
        if err != nil {
            fmt.Printf("Error reading email file %s: %v\n", pathEmails[i], err)
            return
        }
        emails = append(emails, email)
    }

    wg.Add(1)
    go func() {
        defer func() {
            wg.Done()
        }()
        uploadToDatabase(emails)
    }()
}
```

Con este cambio se ha logrado mejorar en un 59% el tiempo de CPU respecto al último mejor resultado (401s a 161s). Considerando que no se espera la finalización de las peticiones enviadas a la base de datos. Se decidió utilizar este enfoque porque el servidor podría estar alojado en un ambiente con mejores recursos, logrando atender mejor las peticiones.

```
File: indexerDatabase
Type: cpu
Time: Jul 30, 2024 at 3:58pm (-05)
Duration: 161.93s, Total samples = 22.01s (13.59%)
Showing nodes accounting for 18.08s, 82.14% of 22.01s total
Dropped 284 nodes (cum <= 0.11s)
Dropped 19 edges (freq <= 0.02s)
Showing top 80 nodes out of 180

See https://git.io/JfYMW for how to read the graph
```

```
File: indexerDatabase
Type: inuse_space
Time: Jul 30, 2024 at 4:01pm (-05)
Showing nodes accounting for 137.32MB, 100% of 137.32MB total

See https://git.io/JfYMW for how to read the graph
```

No obstante, este resultado depende de los recursos de la máquina donde se ejecute el script. Por consiguiente, en ambientes con pocos recursos como la máquina t2.micro de aws el programa no puede ser ejecutado (Status: killed).

La solución está en crear un “semáforo” con subprocesos limitados

```go
// Number of workers to read the emails
const Workers int = 10
```

Y liberar el trabajador cuando se complete la subida a la base de datos

```go
sem <- struct{}{}
wg.Add(1)
go func() {
    defer func() {
        <-sem // Release
        wg.Done()
    }()
    uploadToDatabase(emails)
}()
```

Realizando el último cambio no se obtuvo una mejora de rendimiento. Sin embargo, fue posible controlar el manejo de recursos y ejecutar el programa en diferentes entornos, lo que hace variar el consumo de memoria y el tiempo de uso en CPU.