Open in app

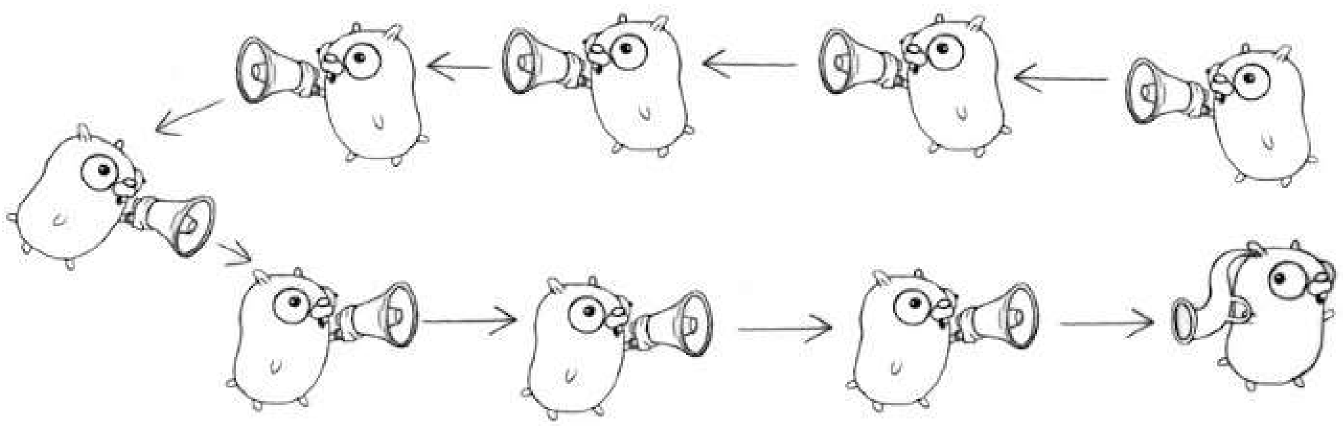
Lucas Godoy

Follow 11 Followers About

Go concurrency applied to data pipelines

A different approach to batch processing, and how to potentiate the power of data pipelines throughout the use of the Go concurrency model.

Lucas Godoy Mar 27 · 6 min read



Introduction to pipelines

The term **pipeline** applied to the computer science field, is nothing more than a series of stages that take data in, perform some operation on that data, and pass the processed data back out as a result.

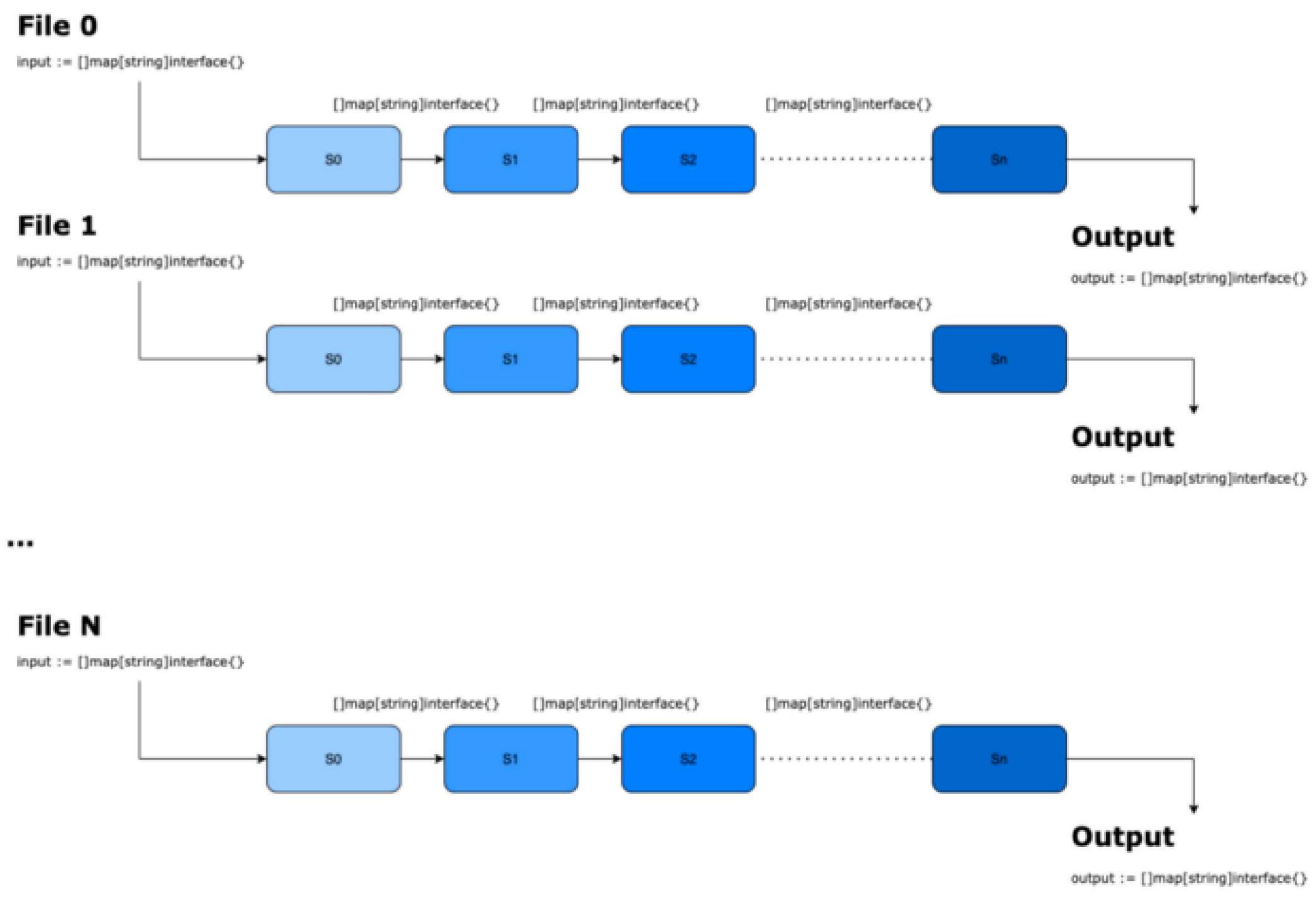
Thus, when using this pattern, you can encapsulate the logic of each stage and scale your features quickly by adding/removing/modifying stages, each stage becomes easy to test, nor mentioning the huge benefit of leveraging it by using concurrency, which was the motto for this article.

Previous problem and solution

to integrate retailer's product availability within the main's company app. After running that integration, the users were able to buy groceries with fewer stockouts risks.

To accomplish this feature, we built this "availability engine" in GoLang. So, what was it about? Long story short, it ingested several retailer's CSV files with product availability information executing a couple of steps to enrich and filter data out based on certain business logic. At the end of the process, new files were made, with all the resultant products to be integrated into the company's app for users to buy.

Overall, this data processing pipe looked something like this:


File 0
input := []map[string]interface{}
[]map[string]interface{}
S0
S1
S2
Sn
Output
output := []map[string]interface{}
File 1
File N
...

Example of architecture for handling in batches

As the diagram states, the first stage of the pipe takes in a set of CSV rows, processes them all, and puts the results on a new batch (new slice of maps). And the very same process repeats it as many times as the number of stages the pipe actually has. The particularity of this pattern was the fact that none of the next stages

To speed up and optimize the job we start using concurrency at the CSV file level, so we were able to process files concurrently. This approach fits us very well, but there is no silver bullet as we always say…

When working on this project, I gave in the concurrency to one of the more experienced engineers in the team, so I mostly took part in coding each of those business logic steps.

Motivated by the curiosity about different languages leads me to give it a try to the Go concurrency model and patterns to have more tools on my toolbelt. Hence I started reading some posts and books about concurrency in Go. While doing so, I stumbled upon a fantastic pattern, pipelines leveraged by the use of channels, which ***if I would have come across it by that time, I would rather suggest implementing it to solve this feature for sure!***

A better approach for data pipelines: streams of data

In my team's approach, we were using batch processing between stages, which was good enough for us, but there are certainly other options that would have suited better to make it more performant.

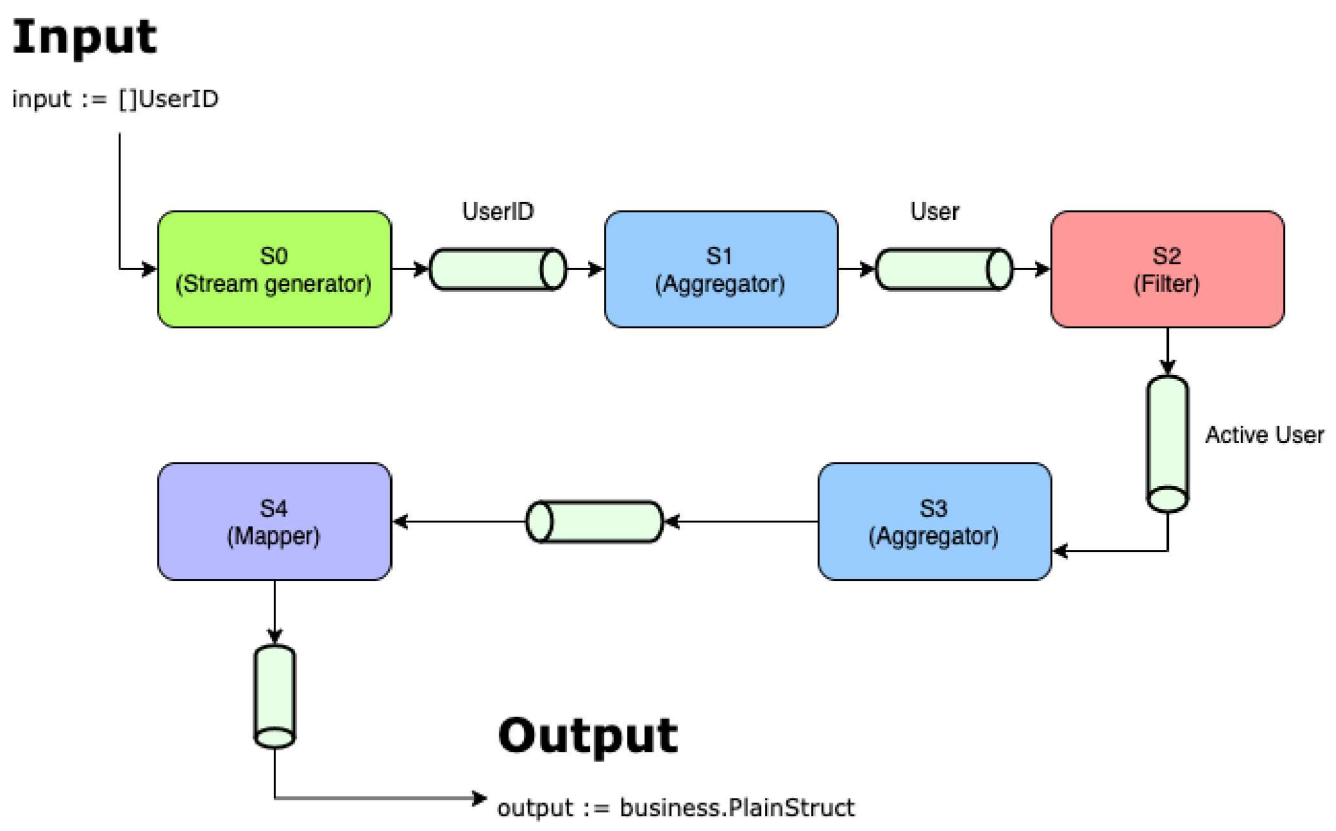
Particularly we are speaking of *streaming data* across the different pipeline's stages. This actually means that each stage receives and emits one element at a time instead of waiting for a full batch of results from the previous step to do its thing on them.

In contrast, if we have to compare the memory footprint between batching and streaming, the former is bigger, since each stage has to make a new slice of maps of equal length to store the results of its calculations. On the contrary, the streaming approach will end up receiving and emitting one element at a time so the memory footprint is back down to the size of the pipeline's input.

Implementation example

To demonstrate the pipeline implementation in Go, I had coded a very silly example of this pattern (if you want you can find the source code here). This pipeline is built on top of a chain of stages whose first input would be a series of *UserIDs* (currently just mocked by code). So then, the pipeline stages are the following:

- **S1**: Fetch the users
- **S2**: Filter the inactive users
- **S3**: Fetch the user's profile and aggregating it into the payload
- **S4**: Convert the overall aggregation as a plain object to be saved later on somewhere



Here below, at the *main function,* we can see how the pipeline invocation will look like at line 24.

```go
1   package main
2
3   import (
4           "fmt"
5
6           "github.com/godoylucase/go-data-pipelines-example/business"
7           "github.com/godoylucase/go-data-pipelines-example/pipeline/stages/aggregate"
8           "github.com/godoylucase/go-data-pipelines-example/pipeline/stages/filter"
9           "github.com/godoylucase/go-data-pipelines-example/pipeline/stages/map_from"
10          "github.com/godoylucase/go-data-pipelines-example/pipeline/stages/stream"
11  )
```

```go
func main() {
	done := make(chan interface{})
	defer close(done)

	userIDs := make([]business.UserID, maxUserID)
	for i := 1; i <= maxUserID; i++ {
		userIDs = append(userIDs, business.UserID(i))
	}

	plainStructs := map_from.UPAggToPlainStruct(done,
		aggregate.Profile(done,
			filter.InactiveUsers(done,
				aggregate.User(done,
					stream.UserIDs(done, userIDs...)))),
	)

	for ps := range plainStructs {
		fmt.Printf("[result] plain struct for UserID %v is: -> %v \n", ps.UserID, ps)
	}
}
```

Let's start from the innermost function call from our pipeline. The first stage *stream.UserIDs(done, userIDs...)* is the one that will feed the pipeline by streaming the *UserIDs* values. To accomplish this, I used a **generator** pattern, which receives a slice of *UserIDs* (input) and by ranging over it will start pushing each value into a *channel* (output). Therefore, the returned channel will be in turn the input for the next stage.

```go
package stream

import (
	"fmt"

	"github.com/godoylucase/go-data-pipelines-example/business"
)

func UserIDs(done <-chan interface{}, uids ...business.UserID) <-chan business.UserID {
	uidsStream := make(chan business.UserID)
	go func() {

			select {
			case <-done:
				return
			case uidsStream <- uid:
			}
		}
	}()
	return uidsStream
}
```

pipeline-generator.go hosted with ♡ by GitHub view raw

Generator pattern to stream values by using a channel for it

As I mentioned before, conceptually, every stage is fed by input and spits an output out for results. So then, the stages are going to *receive a read-only channel* as input to do the data processing and will *return a read-only channel* on which the resulting outputs will be posted on. Doing so, the next stage on the pipeline will take the previous stage's output as its input for executing its own logic, and so forth till the end of the pipe.

Because of that, the use of channels across the pipe will allow us to safely execute concurrently each pipeline stage because our inputs and outputs are safe in concurrent contexts.

Let's have a look at the following stages on the chain, where based on the streamed data from the first stage (generator) we fetch the actual user data, filter inactive users out, enrich them with its profile, and finally rip some data apart to make a plain object out of the whole aggregation/filtering process.

```go
package aggregate

import (
	"fmt"

	"github.com/godoylucase/go-data-pipelines-example/business"
)

type UserFetcher interface {
	Get(ID business.UserID) (business.User, error)
}
```

```go
	go func() {
		defer close(uStream)
		for id := range uids {
			fmt.Printf("[aggregate] User for UserID %v\n", id)

			user, err := getUser(id)
			if err != nil {
				// TODO address errors in a better way
				fmt.Println("some error ocurred")
			}

			select {
			case <-done:
				return
			case uStream <- user:
			}
		}
	}()

	return uStream
}

// getUser dummy function to simulate some fetching action on some user repository
func getUser(ID business.UserID) (business.User, error) {
	username := fmt.Sprintf("username_%v", ID)
	user := business.User{
		ID:       ID,
		Username: username,
		Email:    fmt.Sprintf("%v@pipeliner.com"),
		IsActive: true,
	}

	if ID%3 == 0 {
		user.IsActive = false
	}

	return user, nil
}
```

Fetching users and return them on the channel

```go
package filter

import (
	"github.com/godoylucase/go-data-pipelines-example/business"
)

func InactiveUsers(done <-chan interface{}, users <-chan business.User) <-chan business.User {
	activeUsers := make(chan business.User)
	go func() {
		defer close(activeUsers)
		for user := range users {
			if !user.IsActive {
				fmt.Printf("[filter] Inactive user with UserID %v\n", user.ID)
				continue
			}

			select {
			case <-done:
				return
			case activeUsers <- user:
			}
		}
	}()

	return activeUsers
}
```

```go
package aggregate

import (
	"fmt"

	"github.com/godoylucase/go-data-pipelines-example/business"
)

type ProfileFetcher interface {
	GetByUserID(uid business.UserID) (business.Profile, error)
}

func Profile(done <-chan interface{}, users <-chan business.User) <-chan business.UserProfileAg
	upaStream := make(chan business.UserProfileAggregation)
	go func() {
		defer close(upaStream)
		for user := range users {
			fmt.Printf("[aggregate] Profile for UserID %v\n", user.ID)
```

```go
				// TODO address errors in a better way
				fmt.Println("some error ocurred")
			}

			select {
			case <-done:
				return
			case upaStream <- business.UserProfileAggregation{
				User:    user,
				Profile: profile,
			}:
			}
		}
	}()

	return upaStream
}

// getByUserID dummy function to simulate some fetching action on some profile repository
func getByUserID(uids business.UserID) (business.Profile, error) {
	p := business.Profile{
		ID:       business.ProfileID(uint(uids) + 100),
		PhotoURL: fmt.Sprintf("https://some-storage-url/%v-photo", uids),
	}

	return p, nil
}
```

pipeline_profile_agg.go hosted with ♡ by GitHub view raw

Aggregating user's profiles to the payload

```go
package map_from

import (
	"fmt"

	"github.com/godoylucase/go-data-pipelines-example/business"
)

func UPAggToPlainStruct(done <-chan interface{}, upAggregation <-chan business.UserProfileAggre
	psStream := make(chan business.PlainStruct)
	go func() {
		defer close(psStream)
```

```go
                        select {
                        case <-done:
                                return
                        case psStream <- upa.ToPlainStruct():
                        }
                }
        }()

        return psStream
}
```

pipeline-map-to-plain.go hosted with ♡ by GitHub view raw

Converting the payload to a reduced version of it

For sure you have noticed I passed a *done chan interface{}* around the stages. What is this for? It is worth mentioning that goroutines are not garbage collected in runtime, so we have to make sure as programmers to make them all preemptable. By doing that, consequently, we will not leak any goroutine (I will be writing more about this in another post later on) and freeing up memory. Thus, any invocation to the pipeline could be stopped by just closing the *done* channel. This action will lead to the termination of all the spawn children's goroutines and cleaning them up.

All in all, after the latest stage on the pipe, start pushing data out through its output channel another routine (in this case the main one) could start reading out of that channel by iterating over it as stated on main function line 31 and do something with it.

And that is pretty much it! We implemented our first data pipeline, elegant and efficient! (more than batching pipes).

Final thoughts

In brief, if I any time on I got the chance to work with a similar problem as I had before, I will go for sure for this pattern, which not only is way more performant in terms of memory footprint but also it is faster than doing a batching approach since we can get data being processed concurrently.

Also, there are many other things we can do with pipelines, like rate limiting and fanning it in/out. This topic will be covered later in upcoming posts, where the idea is

This is my first time publishing an article, so any critics are more than welcome, so I can keep learning from all of you guys.

Cheers!

PS: there are a couple of things to be improved in this example that will do make it happen in one upcoming post in which we can talk about error handling for these types of concurrency pattern implementations.

Go Concurrency Programming Data Pipeline