Busca de alta performance com GraphQL e Elasticsearch

Guilherme Baptista

TDC Florianópolis - Trilha Go, Abril de 2019.

Agenda

- O que é performance?
- Qual problema queremos resolver?
- Como escolhemos Go?
- Como trabalhamos com Elasticsearch?
- Onde entra GraphQL na história?
- Resultados
- Reflexões

O que é performance?

O que é performance?

Estrangeirismo predileto na área de T.I.

Não satisfeitos, ainda usamos:

*"Tá **performando** legal!"*

*"Será que vai **performar** esse código?"*

*"Essa API nova deu uma **performada** boa..."*

Performance

em português:

Desempenho

Desempenho

associado com:

Velocidade


Velocidade?

Tempo de resposta:

1 milissegundo

- Servidores: 150 mil dólares por mês;
- Taxa de erros: 97% das requisições.

Serviço de alta performance:

- Alta velocidade;
- Baixa taxa de erros;
- Mínimo custo financeiro;
- Etc.

Simplificando,

vamos associar

performance

com:

Tempo de resposta

"Essa página demora quantos segundos para abrir?"

Throughput suportado (rpm)

"Quantos acessos por minuto esse site aguenta?

Tempo de resposta

Por que isso importa?

O óbvio:

Ninguém gosta de site lento.

Sites que demoram mais que

3 segundos

v

- 80% nunca voltam;
- 57% desistem do acesso;
- 50% falam mal para outras pessoas.

Para cada

100 milissegundos

a mais no tempo de resposta

a *Amazon* perde

-1% de suas vendas.

Para cada

1 segundo

a menos no tempo de resposta

o *Walmart* ganha

+2% de aumento em suas vendas.

Com uma melhoria de

40%

no tempo de carregamento de suas páginas

o *Pinterest* conseguiu um aumento de

+15% no cadastro de usuários.

Para cada

1 segundo

a mais no tempo de resposta

a *BBC* perde

-10% de seus usuários.

Throughput suportado

Por que isso importa?

Um estudo identificou que quando sites ficam

fora do ar

v

- Queda de 100% nos acessos;
- Queda de 100% nas vendas;
- Aumento de 300% nos batimentos cardíacos de quem cuida do site.

Por onde começar então?

1968

George A. Miller

Ciência cognitiva

George A. Miller

- Mnemônica;
- Lei de Miller: 7 coisas de cada vez;
- Palácio da memória (Sherlock Holmes);
- Programação neurolinguística (PNL);
- Psicologia e informação.

1968

A interação entre seres humanos e sistemas de informação.

Quais os limites para manter a atenção de seres humanos ao interagirem com sistemas de informações?


.EXTERNAL REFERENCES - Names and identification
of references to items defined
outside of or accessible outside
of this procedure (system
communication cells, calls to
subroutines, EXTERNAL Data areas,
etc.).
.EXITS, NORMAL - Statement of

Até 100 milissegundos

Ter a sensação de instantâneo.

Até 1 segundo

Ter a sensação de continuidade.

Entre 1 e 10 segundos

Manter a atenção enquanto aguarda.

Acima de 10 segundos

Atenção perdida.

1968

2019?

FPS (*quadros por segundo*)

Olho humano:

12 FPS

Tela de um celular moderno:

60 FPS

Olho humano (12 FPS):

83 milissegundos

Tela de um celular moderno (60 FPS):

16 milissegundos

Até 16 milissegundos

Não existe diferença real para renderização.

Mais que

100 milissegundos

Seres humanos perdem a percepção de instantâneo.

Renderização progressiva

https://developers.google.com

Diretrizes para tempo de resposta

API's:

Idealmente, mantenha tudo abaixo de

100 milissegundos

Nunca ultrapasse os

200 milissegundos

Interfaces:

Idealmente, renderize tudo em até

100 milissegundos

Utilize técnicas de renderização progressiva e nunca ultrapasse

1 segundo

Qual problema queremos resolver?

enjoei.com.br

Marketplace de moda com milhões de produtos que precisa:

- Fazer buscas em todos esses milhões de produtos;
- Aguentar picos de acessos por conta de programas na TV.

O cenário

- Altos custos com infraestrutura;
- Tempo de resposta acima dos 200 milissegundos;
- Grande dificuldade em aguentar picos de acessos instantâneos.

Características

- API's em Ruby on Rails;
- Muito código legado;
- Elasticsearch com problemas de performance.

Como escolhemos Go?

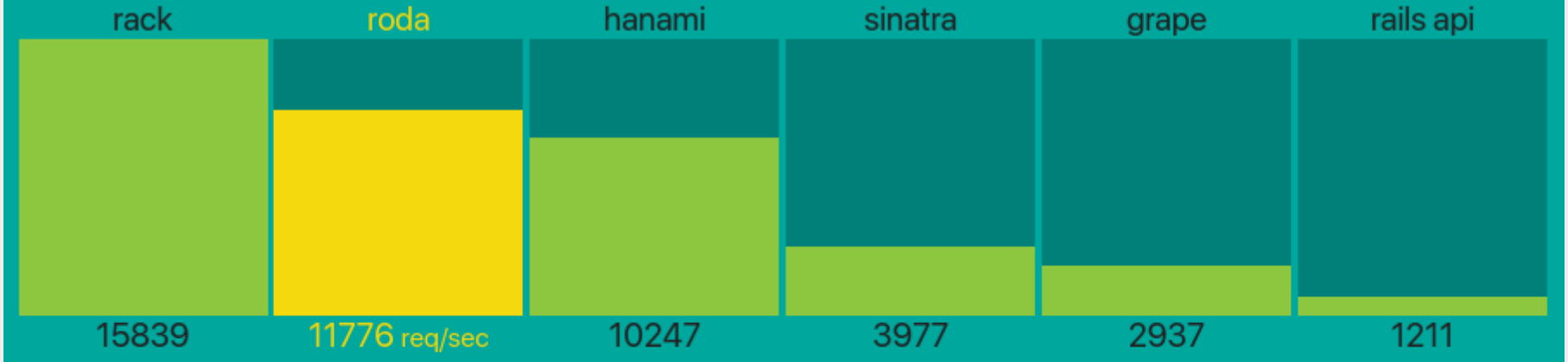
"Não dá mais."

O custo de reescrever tudo

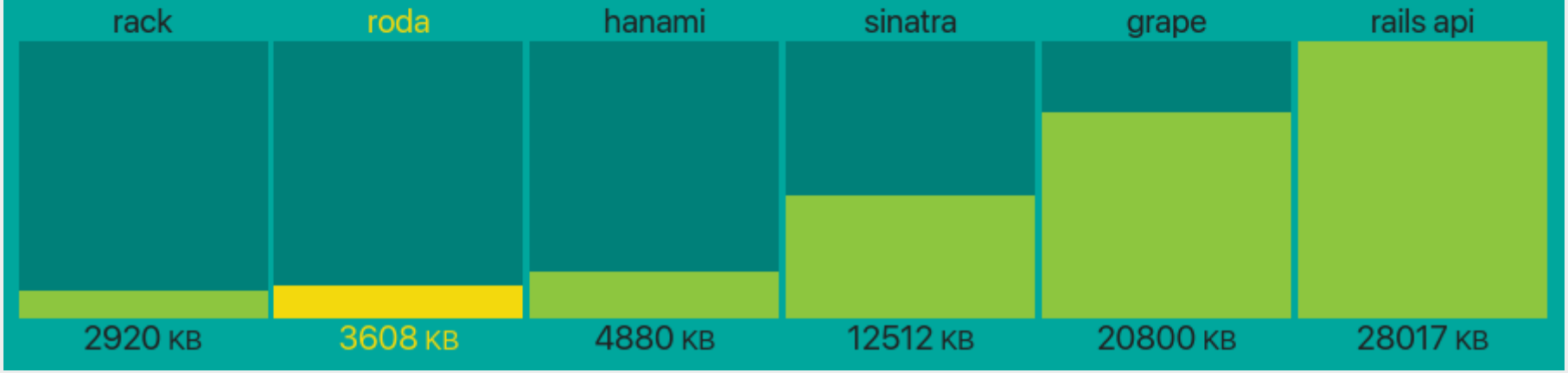
"Se for pra mudar, tem que valer a pena."

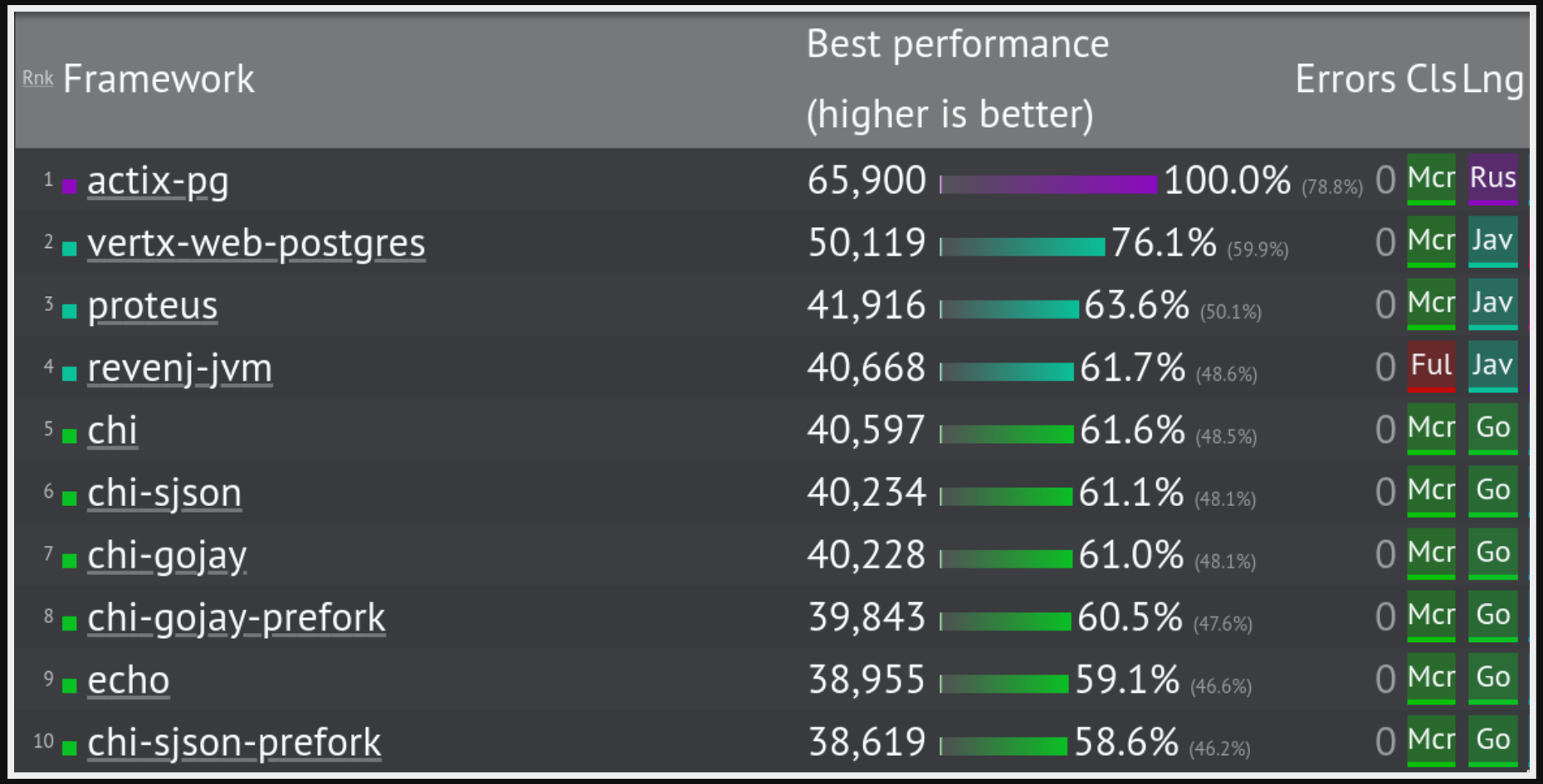
"Será que precisa mesmo?"
https://roda.jeremyevans.net/

Serve More Requests



Use Less Memory

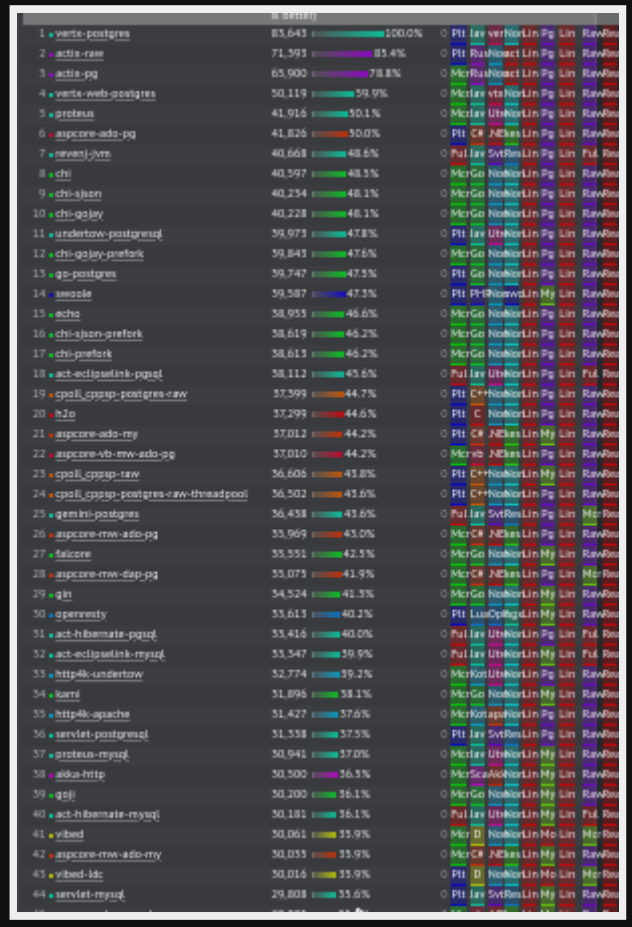




Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng
1	actix-pg	65,900	100.0% (78.8%)	0	Mcr	Rus
2	vertx-web-postgres	50,119	76.1% (59.9%)	0	Mcr	Jav
3	proteus	41,916	63.6% (50.1%)	0	Mcr	Jav
4	revenj-jvm	40,668	61.7% (48.6%)	0	Ful	Jav
5	chi	40,597	61.6% (48.5%)	0	Mcr	Go
6	chi-sjson	40,234	61.1% (48.1%)	0	Mcr	Go
7	chi-gojay	40,228	61.0% (48.1%)	0	Mcr	Go
8	chi-gojay-prefork	39,843	60.5% (47.6%)	0	Mcr	Go
9	echo	38,955	59.1% (46.6%)	0	Mcr	Go
10	chi-sjson-prefork	38,619	58.6% (46.2%)	0	Mcr	Go

https://www.techempower.com/benchmarks/

E o roda?


1 vertx-postgres 83,643 100.0%
2 actix-raw 71,393 85.4%
3 actix-pg 65,900 78.8%
4 vertx-web-postgres 50,119 59.9%
5 proteus 41,916 50.1%
6 aspcore-ado-pg 41,826 50.0%
7 nevenj-jvm 40,668 48.6%
8 chi 40,597 48.5%
9 chi-sjson 40,234 48.1%
10 chi-gojay 40,228 48.1%
11 undertow-postgresql 39,973 47.8%
12 chi-gojay-prefork 39,843 47.6%
13 go-postgres 39,747 47.5%
14 swoole 39,587 47.3%
15 echo 38,955 46.6%
16 chi-sjson-prefork 38,619 46.2%
17 chi-prefork 38,613 46.2%
18 act-eclipselink-pgsql 38,112 45.6%
19 cpoli_cppsp-postgres-raw 37,399 44.7%
20 h2o 37,299 44.6%
21 aspcore-ado-my 37,012 44.2%
22 aspcore-vb-mw-ado-pg 37,010 44.2%
23 cpoli_cppsp-raw 36,606 43.8%
24 cpoli_cppsp-postgres-raw-threadpool 36,502 43.6%
25 gemini-postgres 36,438 43.6%
26 aspcore-mw-ado-pg 35,969 43.0%
27 falcore 35,551 42.5%
28 aspcore-mw-dap-pg 35,075 41.9%
29 gin 34,524 41.3%
30 openresty 33,613 40.2%
31 act-hibernate-pgsql 33,416 40.0%
32 act-eclipselink-mysql 33,347 39.9%
33 http4k-undertow 32,774 39.2%
34 kami 31,896 38.1%
35 http4k-apache 31,427 37.6%
36 servlet-postgresql 31,338 37.5%
37 proteus-mysql 30,941 37.0%
38 akka-http 30,500 36.5%
39 goji 30,200 36.1%
40 act-hibernate-mysql 30,181 36.1%
41 vibed 30,061 35.9%
42 aspcore-mw-ado-my 30,035 35.9%
43 vibed-ldc 30,016 35.9%
44 servlet-mysql 29,808 35.6%

E o roda? ...


56 • revel-raw 26,143 31.5%
57 • actix-diesel 26,039 31.1%
58 • fintrospect 25,642 30.7%
59 • light-4j 24,357 29.1%
60 • ktor-jetty 24,216 29.0%
61 • crystal 24,113 28.8%
62 • http4k 23,378 27.9%
63 • revel-qbs 23,306 27.9%
64 • nodejs-mongodb-raw 23,247 27.8%
65 • nevernj 23,232 27.8%
66 • pedestal 23,210 27.7%
67 • vibed-dmd-pgsql 22,884 27.4%
68 • vibed-ldc-pgsql 22,433 26.8%
69 • revel-jet 22,358 26.7%
70 • jawn 22,204 26.5%
71 • aspcore-mw-ef-pg 22,154 26.5%
72 • hexagon-jetty-postgresql 22,065 26.4%
73 • lapis 21,796 26.1%
74 • aspcore-mvc-ado-pg 21,249 25.4%
75 • restexpress 21,210 25.4%
76 • fasthttp-postgresql 21,002 25.1%
77 • compojure-raw 20,787 24.9%
78 • ktor-cio 20,341 24.3%
79 • luminus 20,275 24.2%
80 • hexagon 20,214 24.2%
81 • play2-scala-anorm-netty 20,146 24.1%
82 • aspcore-mvc-dap-pg 19,891 23.8%
83 • play2-scala-anorm 19,025 22.7%
84 • wicket 18,945 22.6%
85 • aspcore-mvc-ado-my 18,750 22.4%
86 • servant 18,501 22.1%
87 • silicon 17,807 21.3%
88 • play2-java-jpa-hikaricp-netty 17,798 21.3%
89 • aspcore-mvc-dap-my 17,589 21.0%
90 • yesod-mongodb-raw 16,934 20.2%
91 • aspcore-mono-pg 15,982 19.1%
92 • aspcore-mvc-ef-pg 15,590 18.6%
93 • ratpack 15,543 18.6%
94 • fastify 15,516 18.6%
95 • ktor 14,551 17.2%
96 • play2-java-jpa-hikaricp 14,313 17.1%
97 • aspcore-mono-raw-pg 14,177 16.9%
98 • blade 13,974 16.7%
99 • ash-mysql 13,745 16.4%
100 • php-php5-raw 13,711 16.4%
101 • dropwizard-jdbi-postgres 13,275 15.9%
102 • php-raw7-tcp 13,109 15.7%
103 • compojure 12,996 15.5%
104 • nancy-netcore 12,796 15.3%
105 • php-raw7 12,762 15.3%
106 • aspcore-mono-my 12,613 15.1%

E o roda?

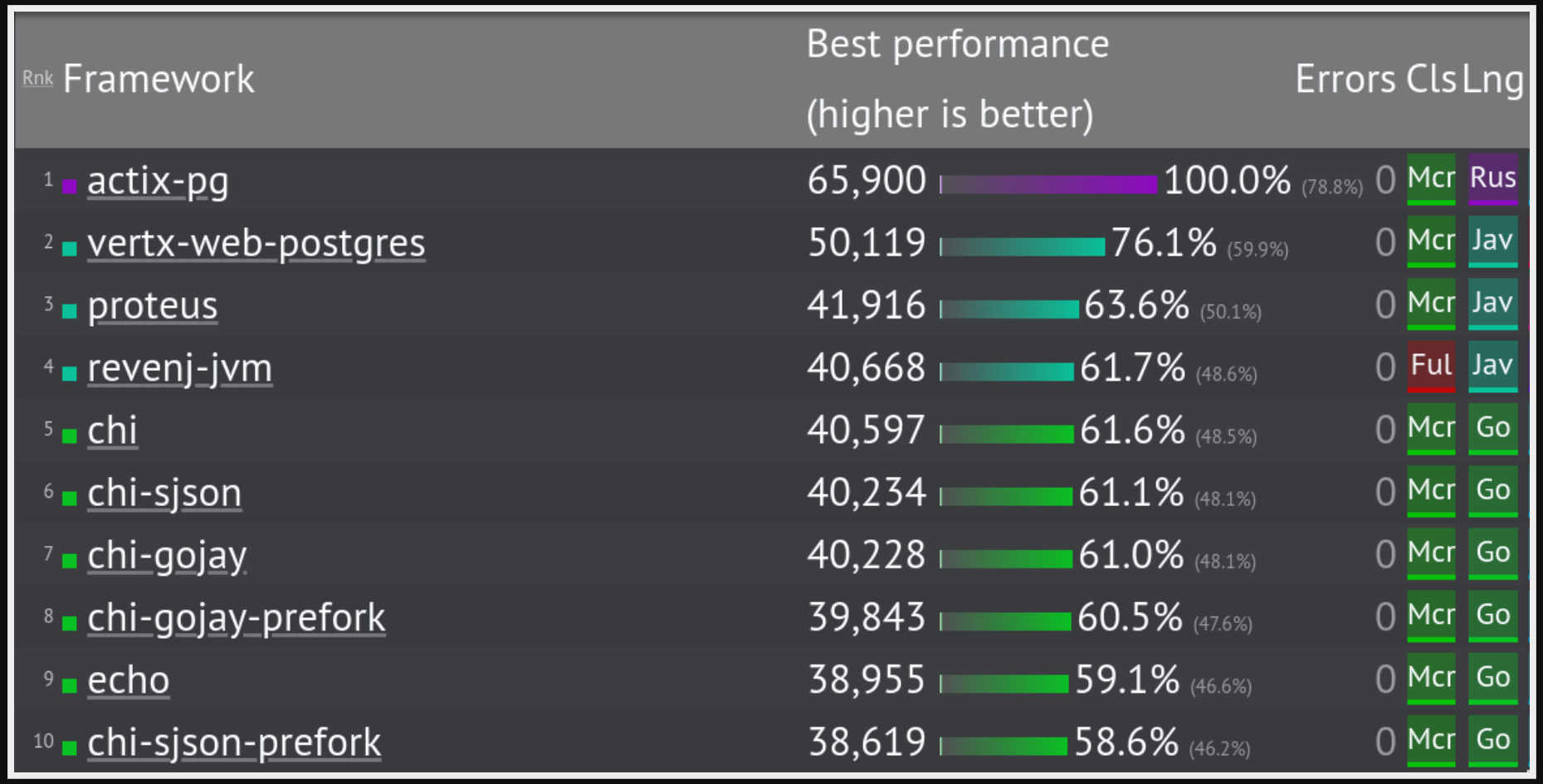
109	dropwizard	12,134	14.5%	0	Ful Jav
110	redstone-mongodb	12,062	14.4%	0	Mcr Dar
111	roda-sequel-postgres	11,924	14.3%	0	Mcr Rby
112	uvicorn	11,867	14.2%	0	Plt Py
113	wildfly-ee7	11,580	13.8%	0	Ful Jav
114	tokio-minihttp	11,530	13.8%	0	Mcr Rus
115	activeweb-jackson	11,498	13.7%	0	Ful Jav

"Se for pra mudar, tem que valer a pena."

Ruby,

até mais, e obrigado pelos peixes!

♥



Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng
1	actix-pg	65,900	100.0% (78.8%)	0	Mcr	Rus
2	vertx-web-postgres	50,119	76.1% (59.9%)	0	Mcr	Jav
3	proteus	41,916	63.6% (50.1%)	0	Mcr	Jav
4	revenj-jvm	40,668	61.7% (48.6%)	0	Ful	Jav
5	chi	40,597	61.6% (48.5%)	0	Mcr	Go
6	chi-sjson	40,234	61.1% (48.1%)	0	Mcr	Go
7	chi-gojay	40,228	61.0% (48.1%)	0	Mcr	Go
8	chi-gojay-prefork	39,843	60.5% (47.6%)	0	Mcr	Go
9	echo	38,955	59.1% (46.6%)	0	Mcr	Go
10	chi-sjson-prefork	38,619	58.6% (46.2%)	0	Mcr	Go

Should I Rust, or Should I Go? https://codeburst.io

Muitos testes, estudos e benchmarks depois:

Let's Go!

- Sempre no top 3 de qualquer benchmark;
- Mais opções de bibliotecas;
- Curva de aprendizado menor;
- Maior produtividade no dia a dia.

```go
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

E agora?

Como servir dados via HTTP?

Sem reinventar a roda ou fazer tudo na mão.

Equilíbrio entre alta performance e usabilidade.

Nossa escolha:

https://github.com/go-chi/chi

```go
router := chi.NewRouter()

router.Use(
  middleware.Logger,
  render.SetContentType(render.ContentTypeJSON),
)

router.Get("/status", statusHandler)

func statusHandler(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("ok"))
}

http.ListenAndServe(":8484", router)
```

Primeira surpresa

```
"GET http://localhost:8484/status HTTP/1.1" from [::1]:42932 - 200 2B in 12.283µs
"GET http://localhost:8484/status HTTP/1.1" from [::1]:42932 - 200 2B in 20.923µs
"GET http://localhost:8484/status HTTP/1.1" from [::1]:42932 - 200 2B in 9.991µs
"GET http://localhost:8484/status HTTP/1.1" from [::1]:42932 - 200 2B in 10.276µs
"GET http://localhost:8484/status HTTP/1.1" from [::1]:42932 - 200 2B in 13.137µs
"GET http://localhost:8484/status HTTP/1.1" from [::1]:42932 - 200 2B in 16.656µs
"GET http://localhost:8484/status HTTP/1.1" from [::1]:42932 - 200 2B in 19.608µs
```

9,991µs

µs

µs

significa

microssegundos

1 segundo

é igual a

1 mil milissegundos

que é igual a

1 milhão de microssegundos

9,991µs

é igual a

0,000009991 segundos

Como trabalhamos com Elasticsearch?

Do jeito errado.

O que aprendemos:

É a melhor opção para buscas complexas em um volume enorme de dados.

É tão simples e prático que mesmo fazendo tudo errado ele funciona.

A modelagem dos seus documentos é tudo.

O que aprendemos:

Não tentar usar o mesmo índice para responder perguntas diferentes.

"keyword" ao invés de "text" faz toda diferença.

Scripts são malignos.

Arredondar datas economiza muito dinheiro.

Como ler dados do Elasticsearch com Go?

Nossa escolha:

https://github.com/olivere/elastic

```
client, _ := elastic.NewClient(
    elastic.SetURL("http://127.0.0.1:9201"))

searchResult, _ := client.Search().
Index("products").
Query(elastic.NewTermQuery("title", "sandália preta")).
Do(context.Background())

var product Product

for _, p := range searchResult.Each(reflect.TypeOf(product)) {
    product := p.(Product)
    fmt.Printf("%s", product.Title)
}
```

O primeiro problema...

"Já ví esse filme antes..."

Elasticsearch Query DSL

Uma lição que aprendemos:

Toda biblioteca que tenta criar uma abstração para Elasticsearch resulta em código que ninguém entende.

Funciona bem para SQL

```ruby
User.where(name: 'David', occupation: 'Code Artist')
    .order(created_at: :desc)

User.where(email: 'david@email.com').first

User.where('update_at > ?', 1989).each do |user|
  puts user.name
end
```

```
user := &User{}

db.Model(user)
    .Where("name", "David").Where("occupation", "Code Artist")
    .Order("created_at DESC").Select()

db.Model(user).Where("email", "david@email.com").Select()

var users []User

db.Model(&users).Where('update_at > ?', 1989).Select()

for _, user := range users {
    fmt.Printf("%s", user.Name)
}
```

Não funciona bem para Elasticsearch

```
query do
  script_fields distance: {
    script: "doc['location'].distance(lat, lon)",
    params: {lat: latitude, lon:longitude}
  }
  fields ['_source']
  query do
    filtered do
      query do
        multi_match do
          query str
          fields %w[title name]
        end
```

```
query := elastic.NewMatchAllQuery()
aggs := elastic.NewDateHistogramAggregation().
    Field("time").
    Interval("hour").
    Format("yyyy-MM-dd HH")
aggs.SubAggregation(
    "count", elastic.NewSumAggregation().Field("no_of_hits"))
searchResult, _ := client.Search().
    Index("my_index").
    Type("_doc").
    Query(query).
    Aggregation("date", aggs).
    Do(ctx)
hour_agg, found := searchResult.Aggregations.Terms("date")
```

Qual a solução então?

Templates JSON

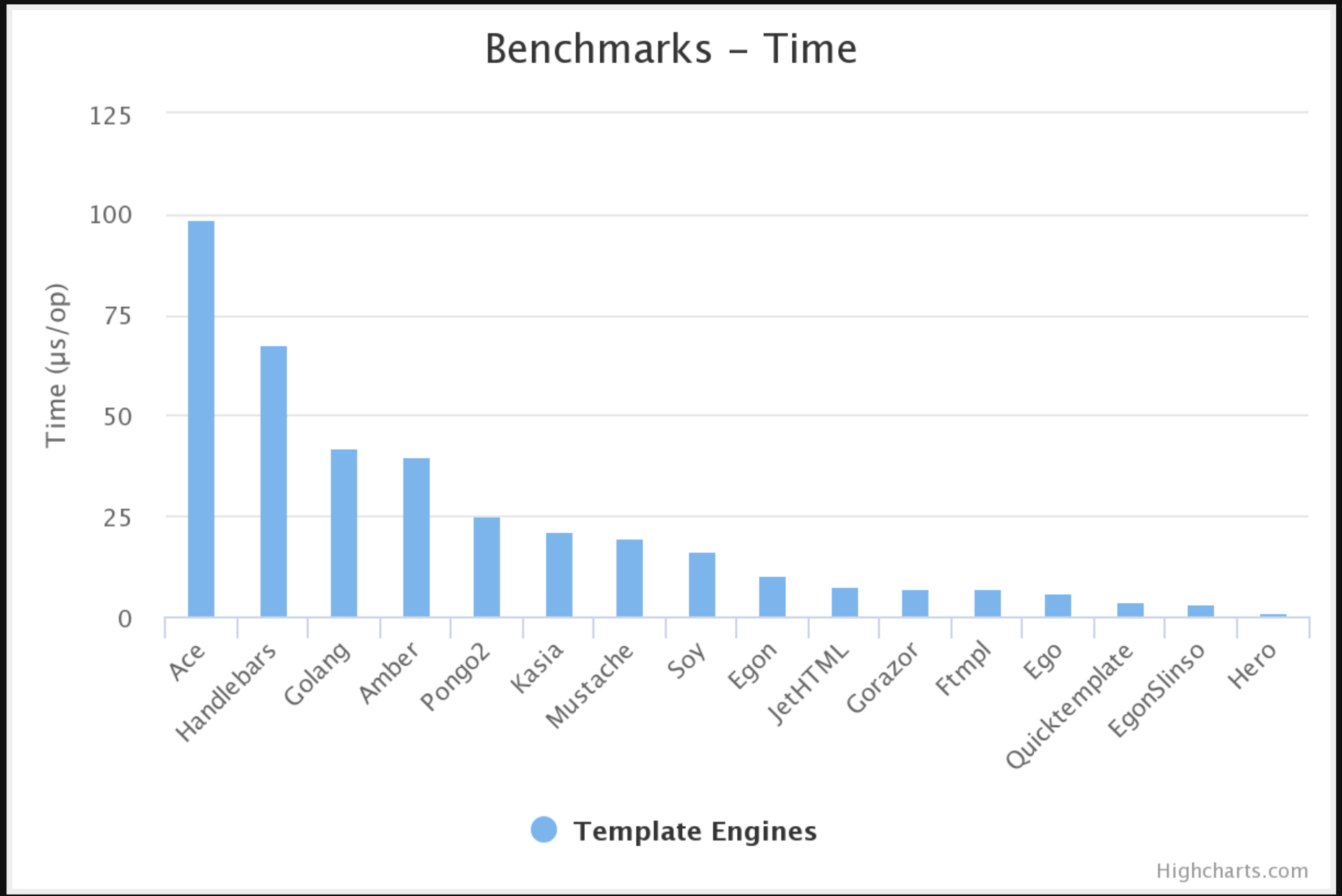
```
{
  "size": 10,
  "query": {
    "bool": {
      "must": {
        "match": { "title": "vestido" }
      },
      "filter": {
        { "term":  { "color": "vermelho" } }
      }
    }
  }
}
```

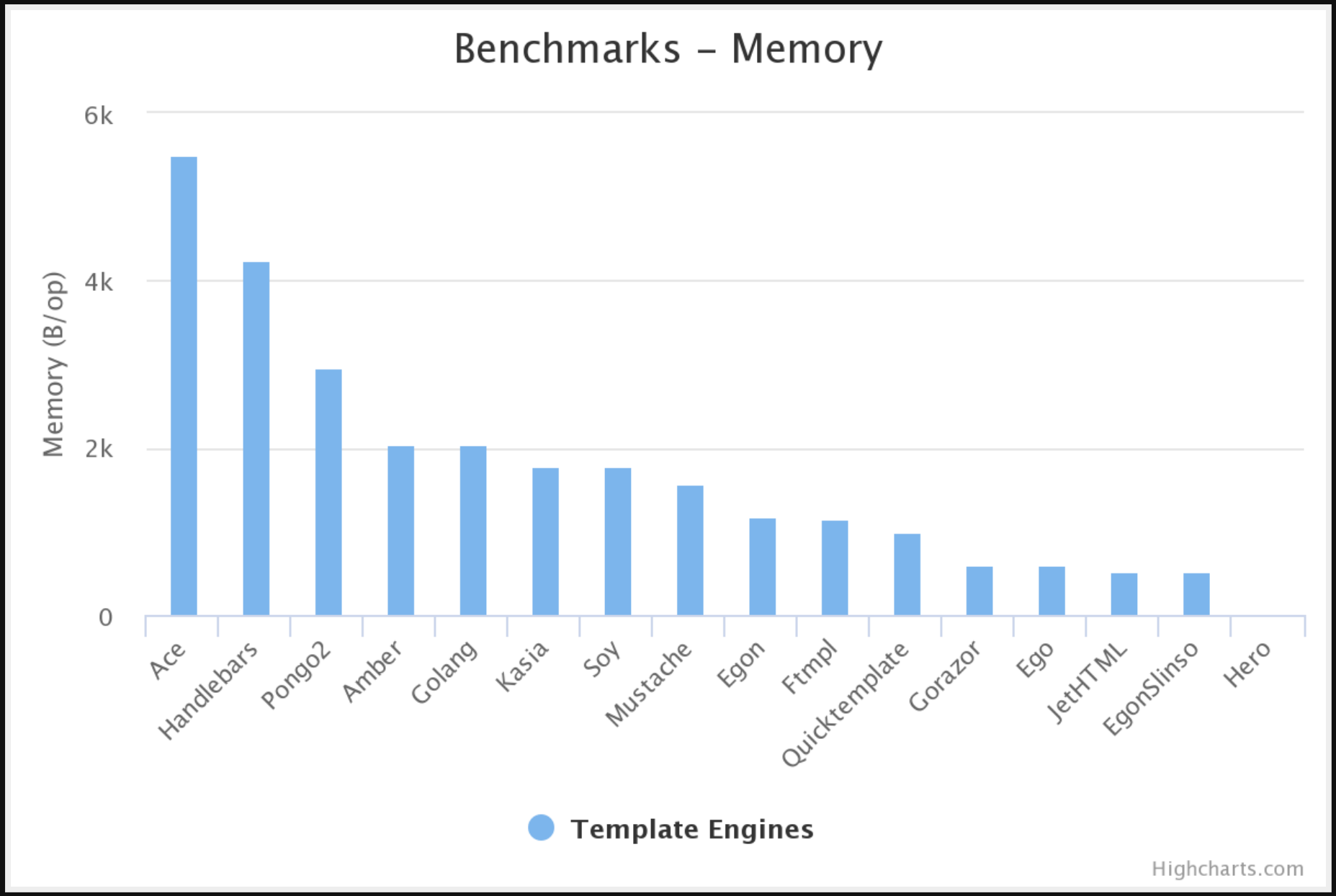
Qual engine de templates?

Equilíbrio entre alta performance e usabilidade.

Nossa escolha:

https://github.com/shiyanhui/hero


Benchmarks – Time
125
100
75
50
25
0
Time (µs/op)
Ace
Handlebars
Golang
Amber
Pongo2
Kasia
Mustache
Soy
Egon
JetHTML
Gorazor
Ftmpl
Ego
Quicktemplate
EgonSlinso
Hero
Template Engines
Highcharts.com


Benchmarks – Memory
Memory (B/op)
6k
4k
2k
0
Ace
Handlebars
Pongo2
Amber
Golang
Kasia
Soy
Mustache
Egon
Ftmpl
Quicktemplate
Gorazor
Ego
JetHTML
EgonSlinso
Hero
Template Engines
Highcharts.com

```
<%: func SearchProducts(term string, color string) %>

{
  "size": 10,
  "query": {
    "bool": {
      "must": {
        "match": { "title": "<%==s term %>" }
      },
      "filter": {
        { "term":  { "color": "<%==s color %>" } }
      }
    }
  }
}
```

```go
func Search() {
    jsonBody := new(bytes.Buffer)

    templates.SearchProducts("vestido", "vermelho", jsonBody)

    products := elastic.RequestFromJson(
        "GET", "/products/_search", jsonBody)
}
```

Precisa da biblioteca para Elasticsearch então?

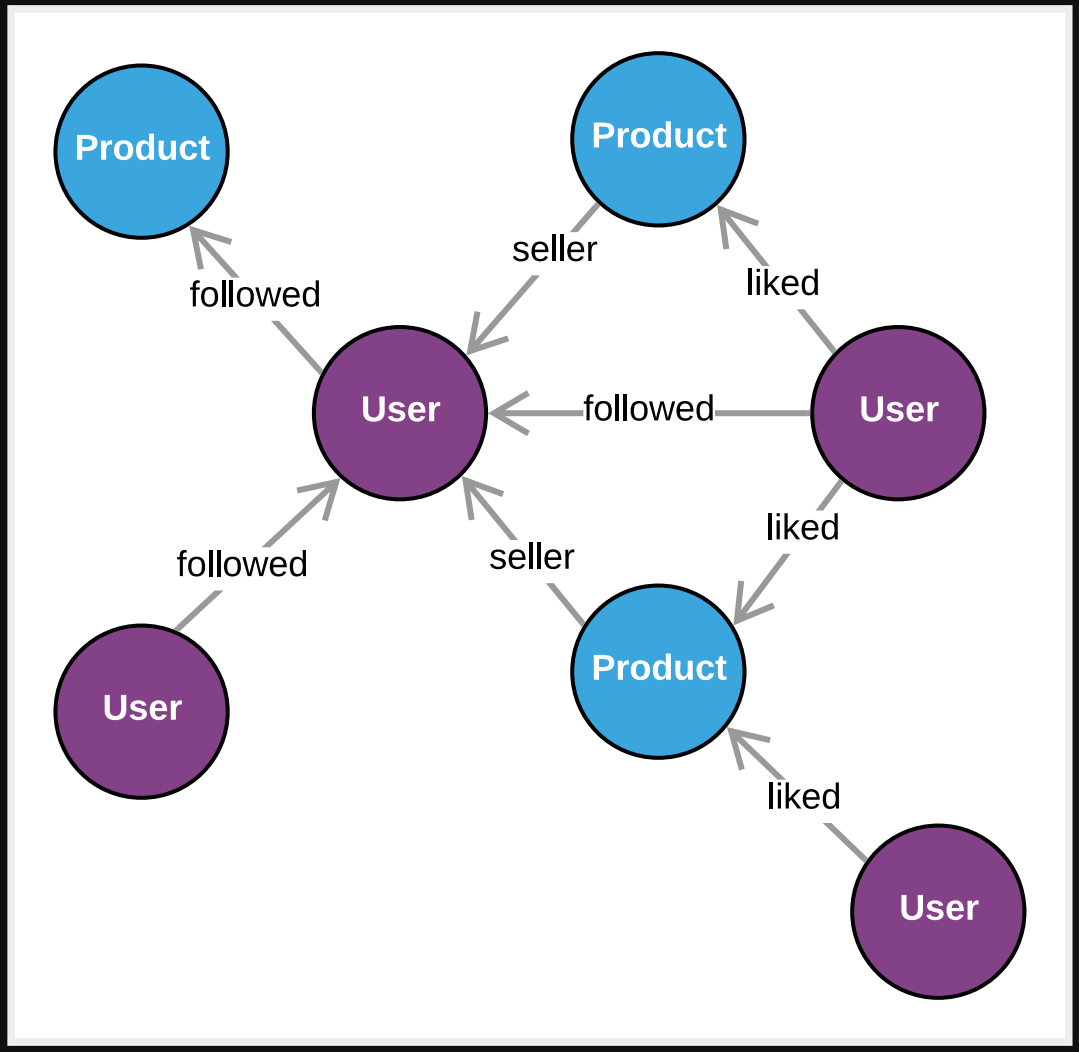
https://github.com/olivere/elastic

Sim

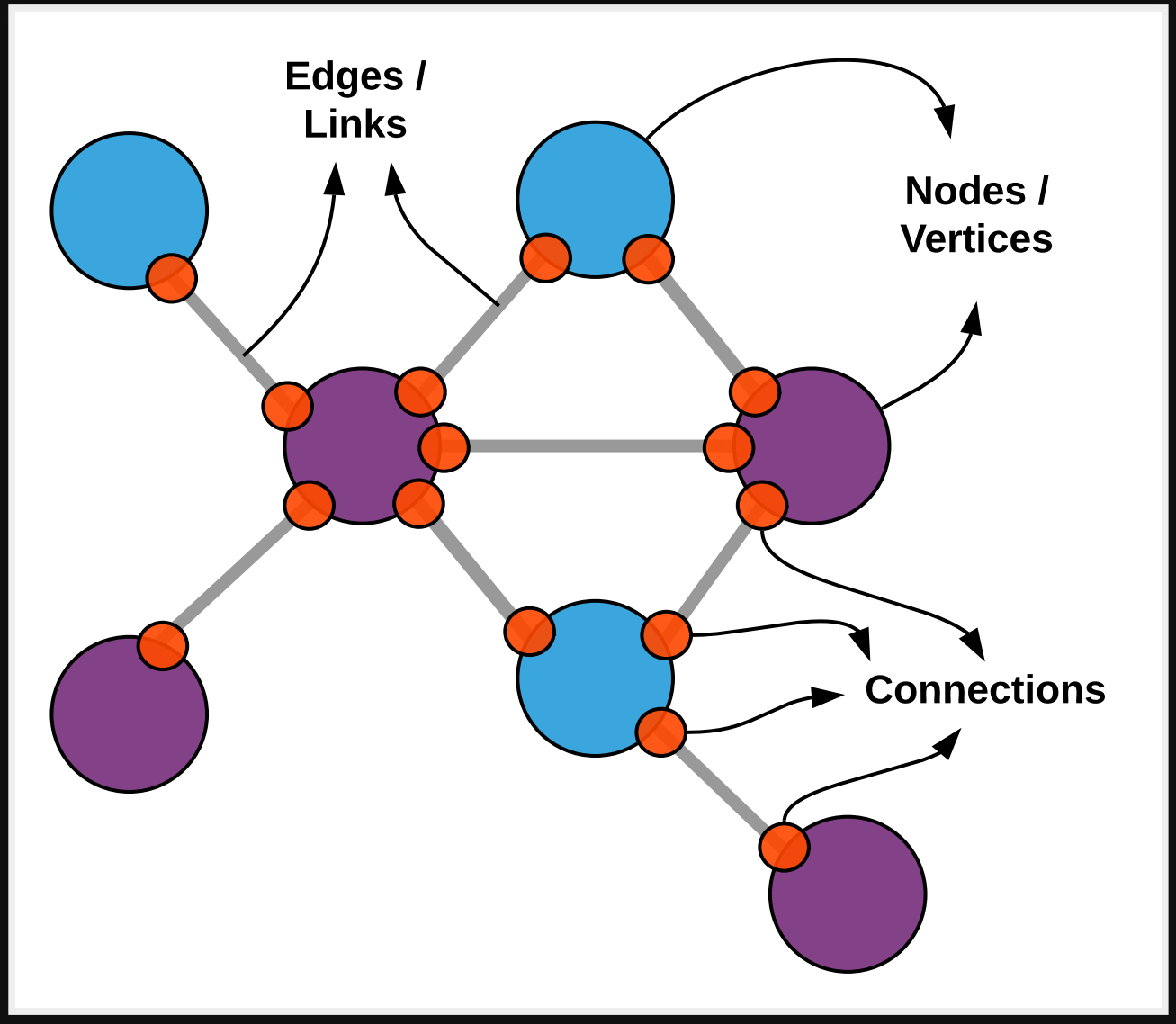
- Controle de conexões com o Elasticsearch;
- Autenticação do cliente;
- Parser de resultados;
- Bulk API para inserção de dados.

Onde entra GraphQL na história?

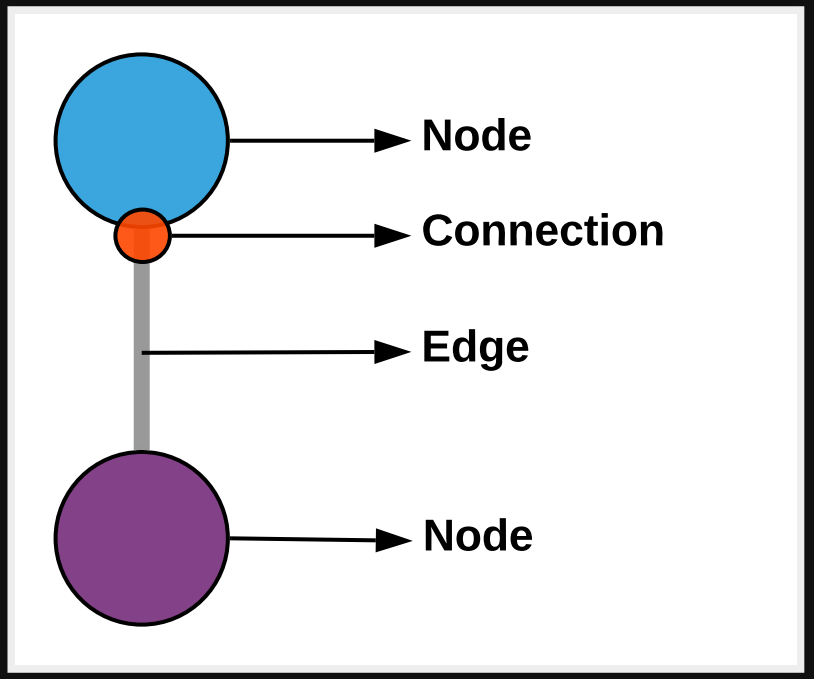
Como funciona conceitualmente



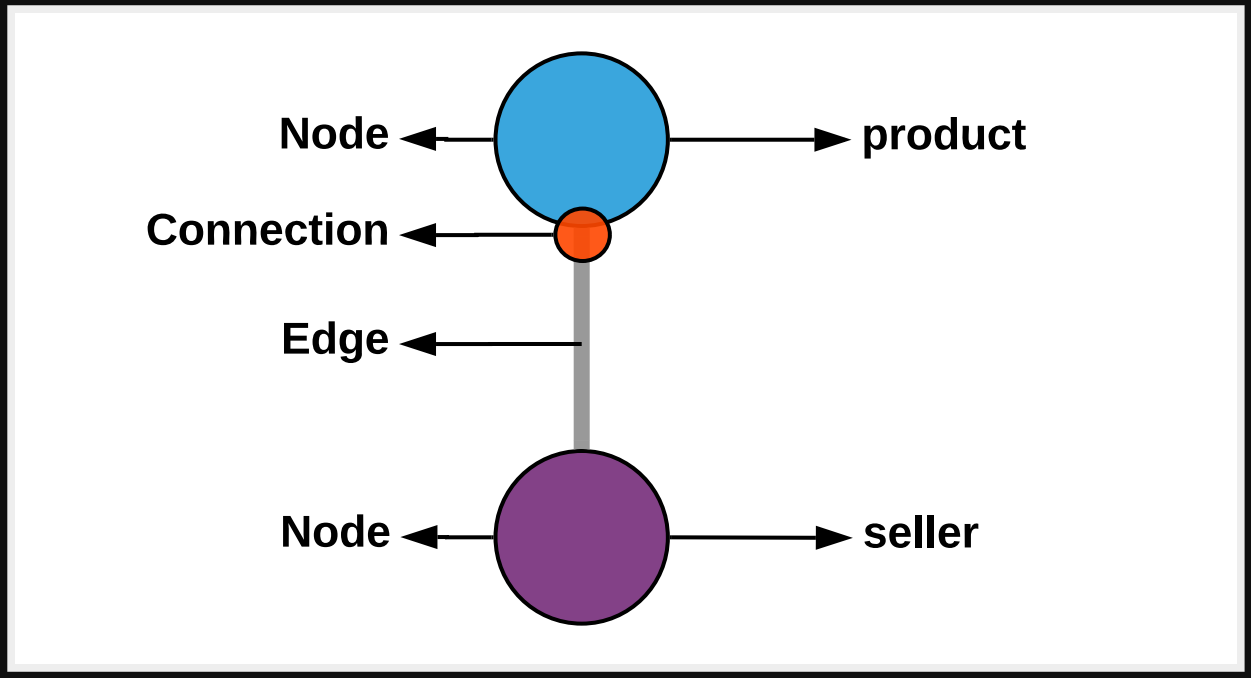
Teoria dos grafos



Representação de recursos



Um produto e o seu vendedor



```
product(id: 8945) {
  title
  seller {
    name
  }
}
```

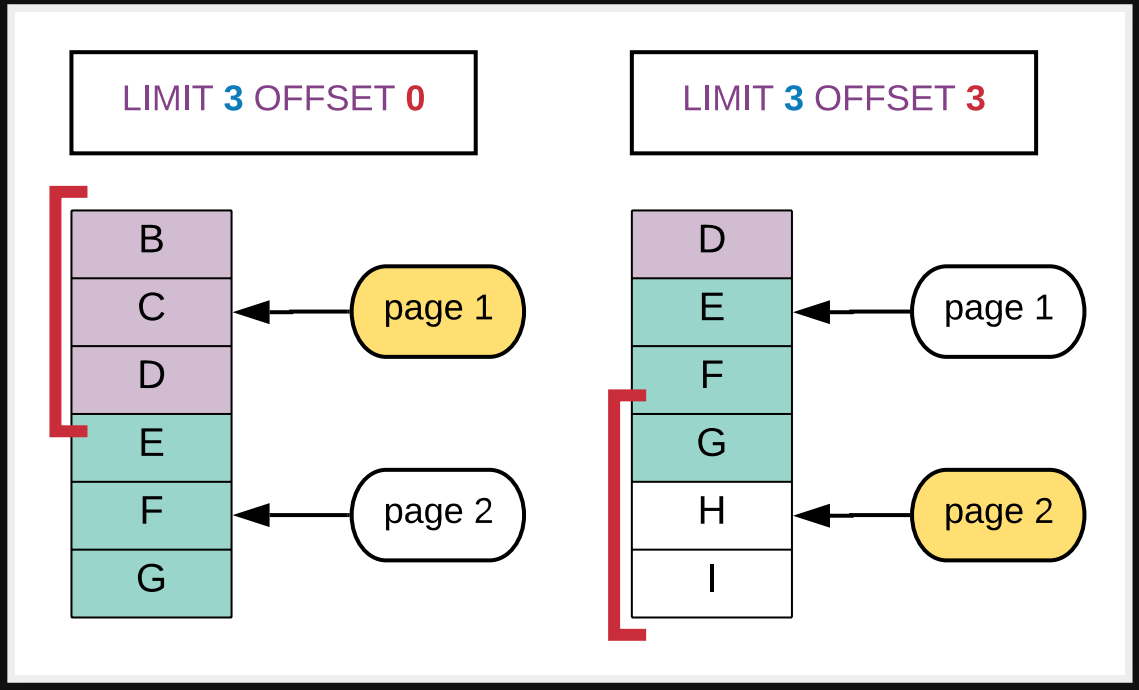
```
query productQuery($id: ID!) {
  product(id: $id) {
    title
      seller {
        name
      }
    }
  }
}
```

```
{
  "id": 8945
}
```

```
{
  "data": {
    "product": {
      "title": "vestido vermelho",
      "seller": {
        "name": "beatriz"
      }
    }
  }
}
```

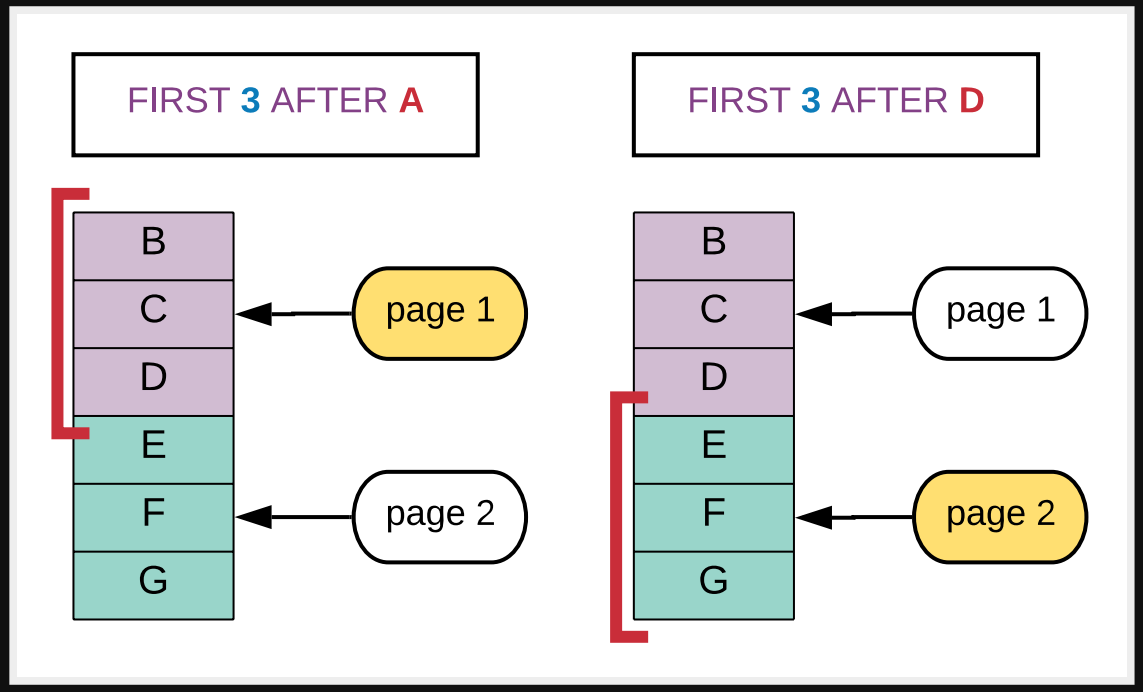
Paginação

Paginação baseada em Offset

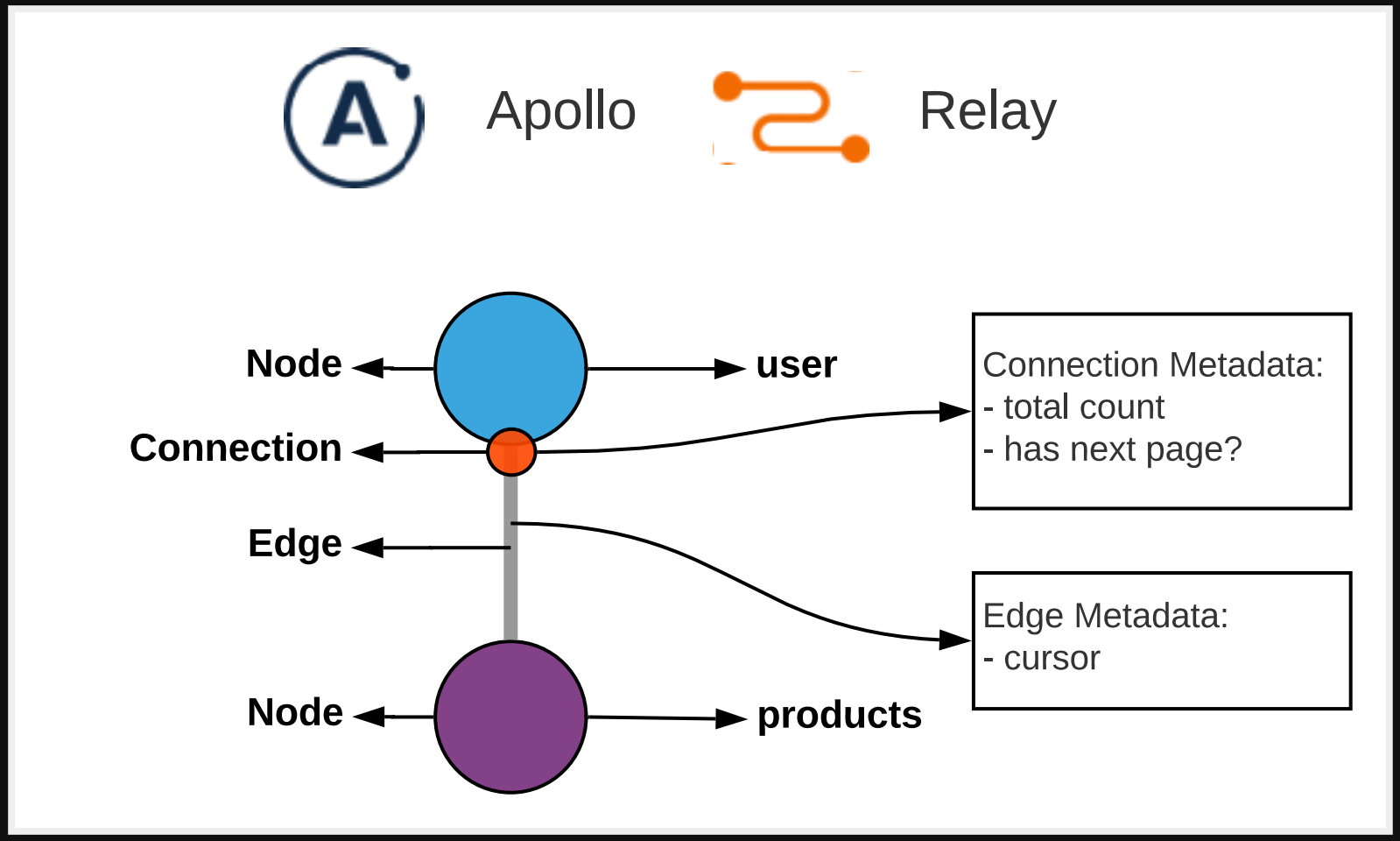


Esqueça paginação baseada em Offset

Paginação baseada em Cursores:



Metadados nas conexões e arestas



Como esses dados são representados?

Representação simplifcada:

```
{
  "product": {
    "title": "vestido vermelho",
    "seller": {
      "name": "lojinha da bia"
    }
  }
}
```

Representação com conexões e arestas:

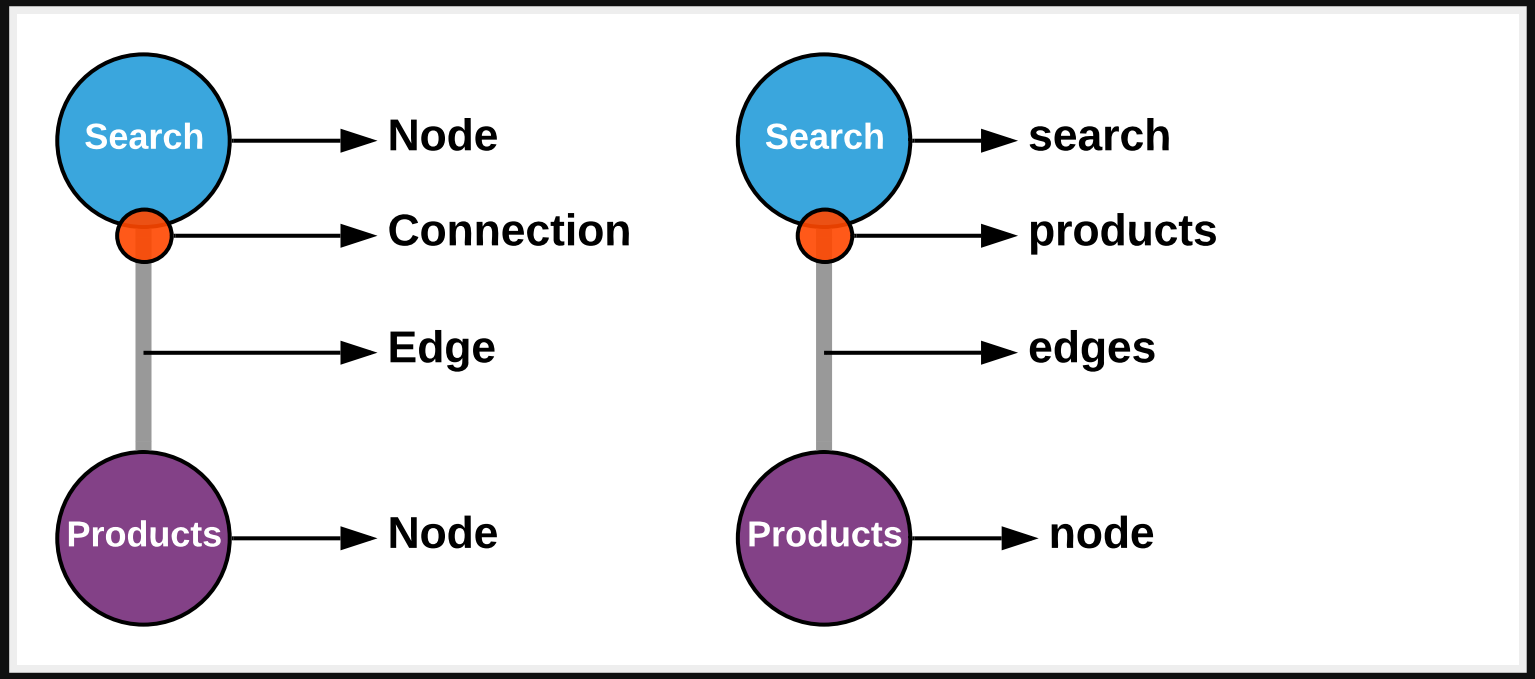
```
{
  "user": {
    "productConnection": {
      "totalCount": 37,
      "productEdges": [
        {
          "node": { "title": "vestido vermelho" },
          "cursor": "726"
        }
      ]
    }
  }
}
```

Como paginar os dados:

```
query userProducts($user_id: ID!, $first: Int, $after: ID) {
  user(id: $user_id) {
    products(first: $first, after: $after) {
      totalCount
      edges {
        node { title }
        cursor
      }
    }
  }
}
```

```
{ "user_id": 1873, "first": 10, "after": "726" }
```

A estrutura de uma busca



Como uma busca é feita:

```
query Search($search: Search, $first: Int, $after: ID) {
  search(search: $search) {
    term
    products(first: $first, after: $after) {
      totalCount
      edges {
        node {
          title
          seller { name }
        }
        cursor
      }
    }
  }
}
```

```
{
  "search": { "term": "vestido vermelho" },
  "first": 10, "after": "236"
}
```

O resultado:

```
"search": {
  "term": "vestido vermelho",
  "products": {
    "totalCount": 152836,
    "edges": [
      {
        "node": {
          "title": "vestido  vermelho",
          "seller": { "name": "beatriz" }
        },
        "cursor": "245"
      }
    ]
  }
}
```

Fragmentos

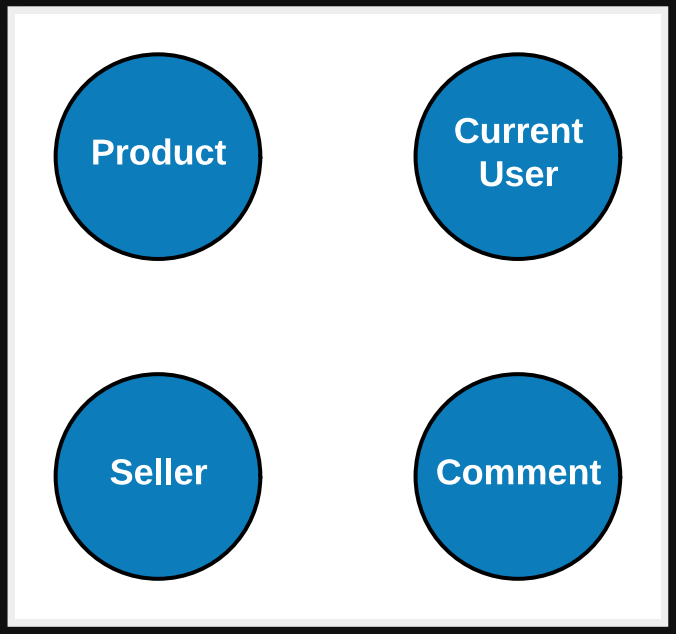
```
fragment productWithSeller on Product {
  title
  seller {
    name
  }
}
```

```
search(search: $search) {
  products(first: $first, after: $after) {
    edges {
      node {
        ...productWithSeller
      }
    }
  }
}
```

```
product(id: $id) {
  ...productWithSeller
}
```

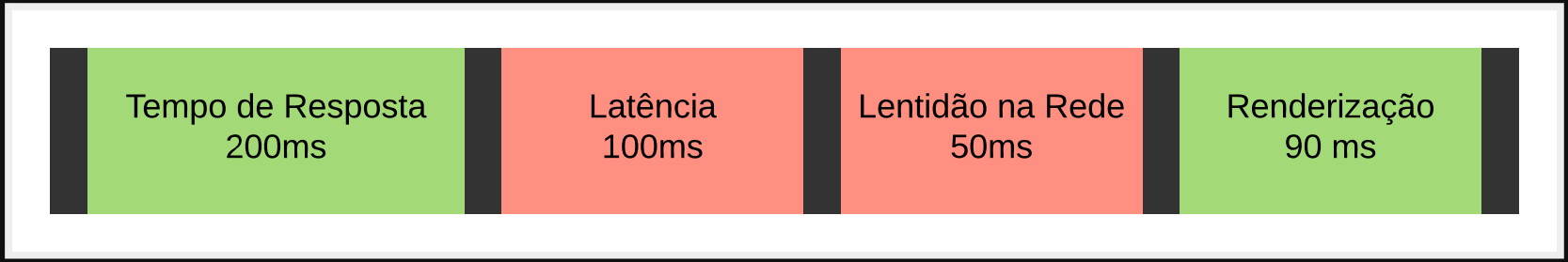
Mas por que tudo isso importa?

Em uma interface



Diversas requisições diferentes

- GET /current_user
- GET /product/7
- GET /product/7/seller
- GET /product/7/comment/4



440 milissegundos

em 1 requisição

Agora imagine fazer isso...

4 vezes;

Servidor nos EUA;

Celular com 3G.

100 milissegundos

Sensação de instantâneo.

1 requisição para resolver tudo:

```
currentUser { name gender }

product {
  title
  price
  seller { name }
  comments {
    author { name }
    message
    replies {
      author { name }
      message
    }
  }
}
```

GraphQL + Go

Goroutines

```
var waitGroup sync.WaitGroup

user := &User{}
product := &Product{}

waitGroup.Add(1)
go loadUser(&waitGroup, user)

waitGroup.Add(1)
go loadProduct(&waitGroup, product)

waitGroup.Wait()
```

Como servir dados via GraphQL?

Equilíbrio entre alta performance e usabilidade.

Nossa primeira tentativa:

https://github.com/graphql-go/graphql

```
graphql.NewObject(
    graphql.ObjectConfig{ Name: "GraphQL", Fields: fields})

graphql.ObjectConfig{
    Name: "Query",
    Fields: graphql.Fields{
        "source": &graphql.Field{Type: graphql.String}}})

graphql.NewSchema(graphql.SchemaConfig{Query: queryType})

var query_args = graphql.FieldConfigArgument{
    "term": &graphql.ArgumentConfig{Type: graphql.String},

var Query = &graphql.Field{
    Type: gq.Query, Args: query_args, Resolve: resolvers.Query,}
```

Verboso;

Complexo;

Confuso;

Nada produtivo.

Schema based

```
type Query {
  user(id: ID!): User
  product(id: ID!): Product
}

type User {
  id: ID
  name: String
}

type Product {
  id: ID
  title: String
  seller: User
}
```

Nossa segunda tentativa:

https://github.com/99designs/gqlgen

Geração de código;
Funcionamento obscuro;
Muita mágica.

Nossa escolha final:

https://github.com/graph-gophers/graphql-go

```
type Query {
  user(id: ID!): User
  product(id: ID!): Product
}

type User {
  id: ID
  name: String
}

type Product {
  id: ID
  title: String
  seller: User
}
```

```
type UserResolver struct {
  Field fields.Field `graphql:"user"`
  Model models.User
}

func (resolver *Resolver) User(id params.Id) *UserResolver {
  return &UserResolver{Model: userModel.Find(id)}
}

func (userResolver *UserResolver) Name() *string {
  return &userResolver.Model.Name
}
```

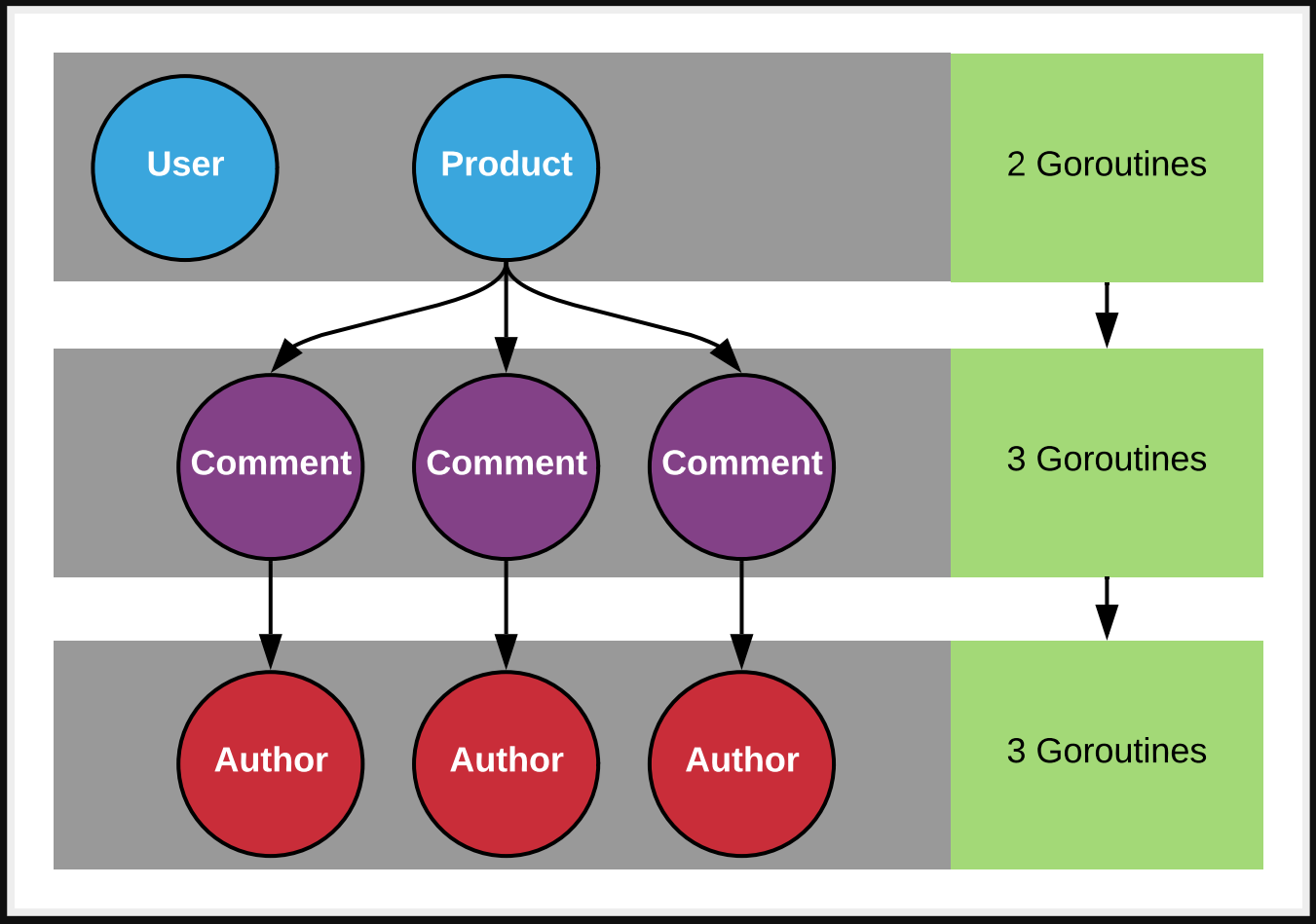
Simples;

Direto ao ponto;

Paralelismo por padrão.

Paralelismo por padrão:

```
user { name }
product {
  title
  comments {
    message
    author { name }
  }
}
```


User
Product
2 Goroutines
Comment
Comment
Comment
3 Goroutines
Author
Author
Author
3 Goroutines

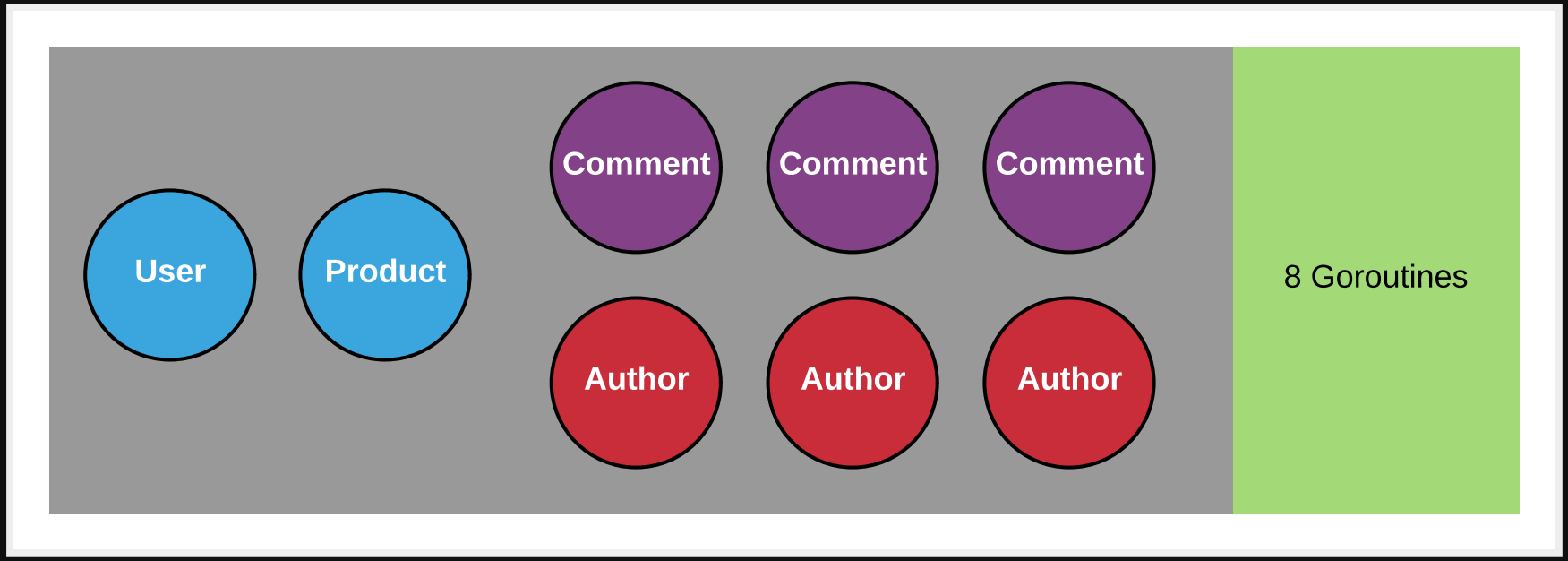
Tem como melhorar?

Sim

requested-fields

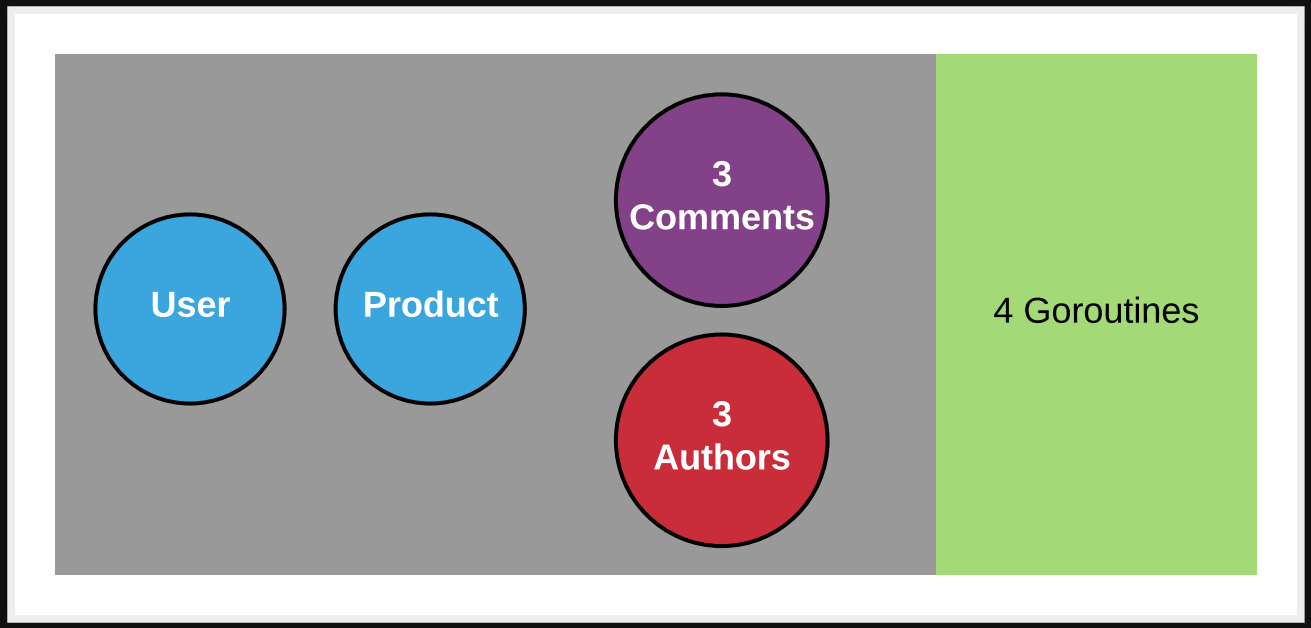
https://github.com/gbaptista/requested-fields

```
hasComments := fields.RequestedForAt(ctx,
    productResolver, "comments")

hasCommentAuthor := fields.RequestedForAt(ctx,
    productResolver, "comments.author")

if hasComments {
    waitGroup.Add(1)
    go loadCommentsFor(productResolver.Model)
}

if hasCommentAuthor {
    waitGroup.Add(1)
    go loadCommentAuthorsFor(productResolver.Model)
}
```


User
Product
Comment
Comment
Comment
Author
Author
Author
8 Goroutines

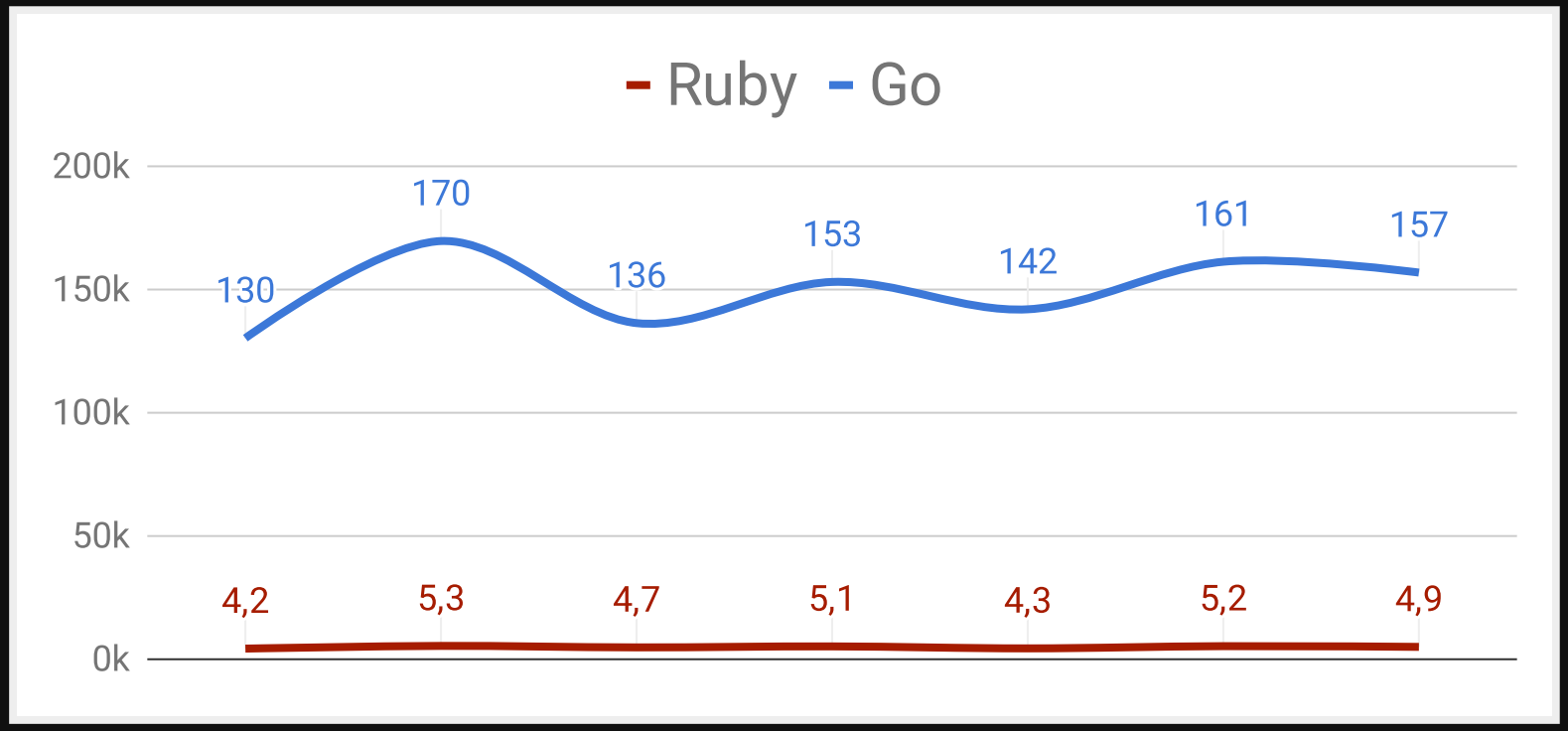
Tem como melhorar?

Sim

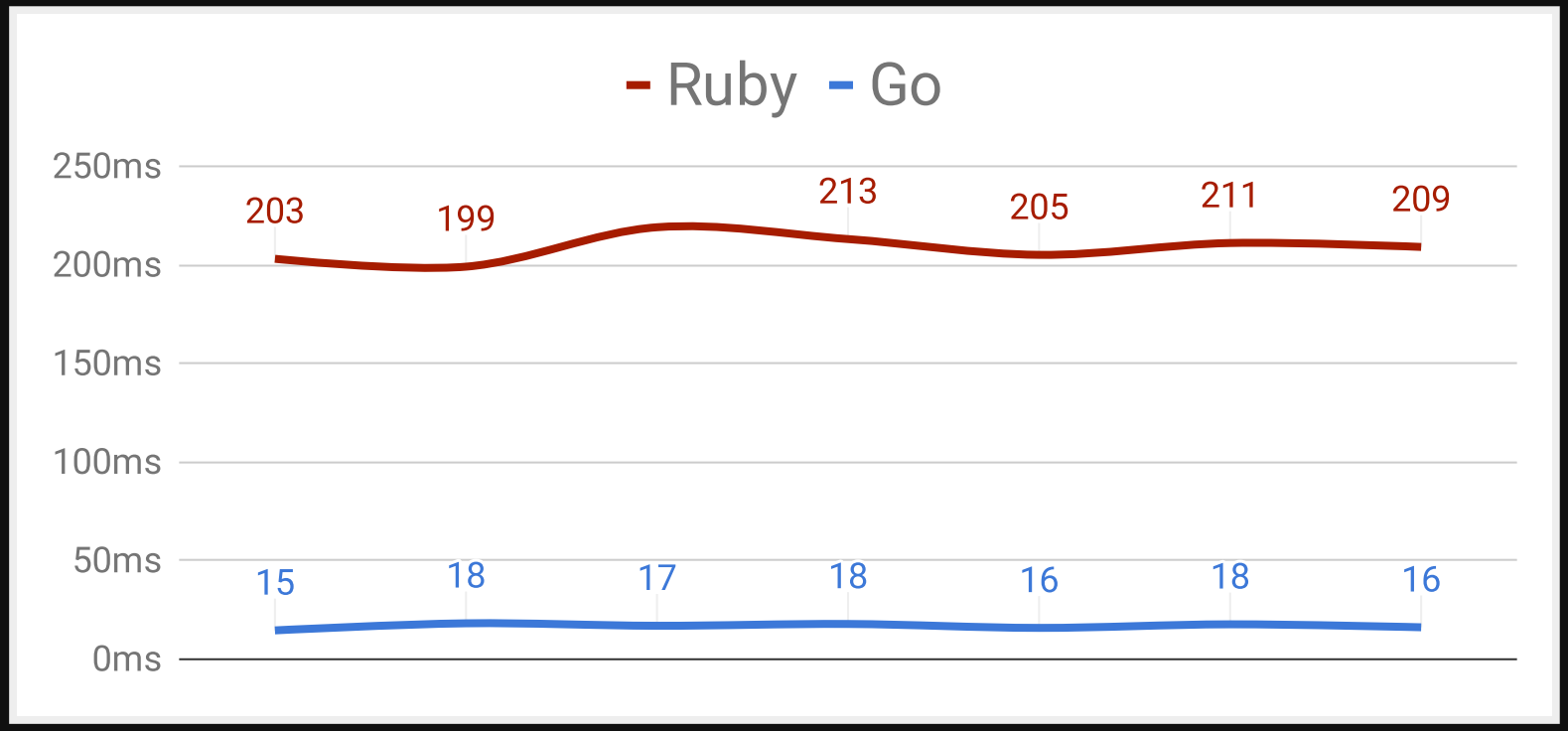

User
Product
3
Comments
3
Authors
4 Goroutines

Resultados

Throughput suportado (rpm)

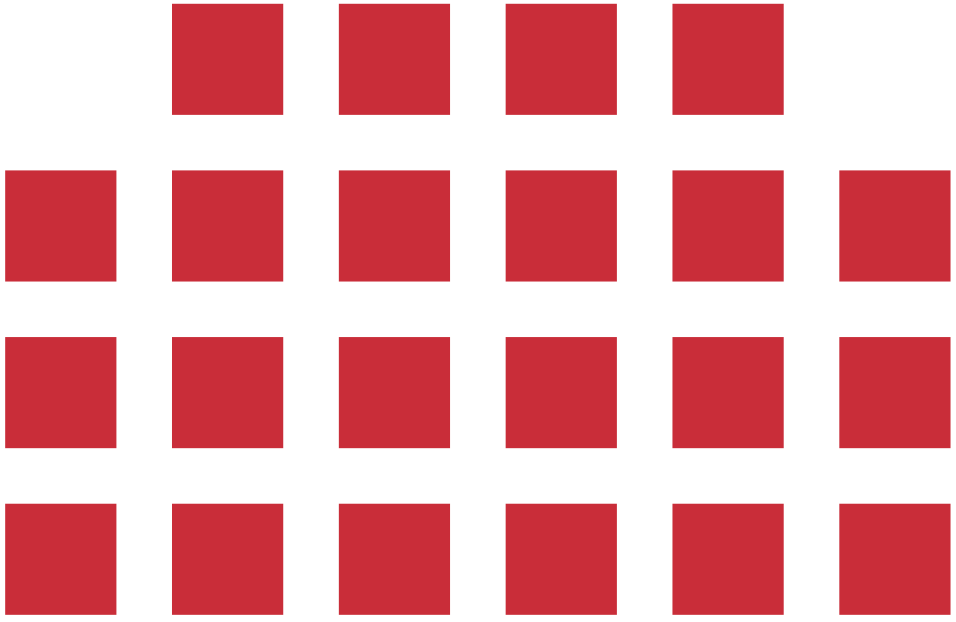


Tempo de resposta

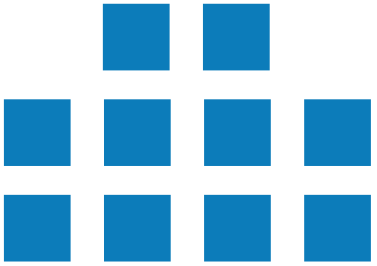


Servidores

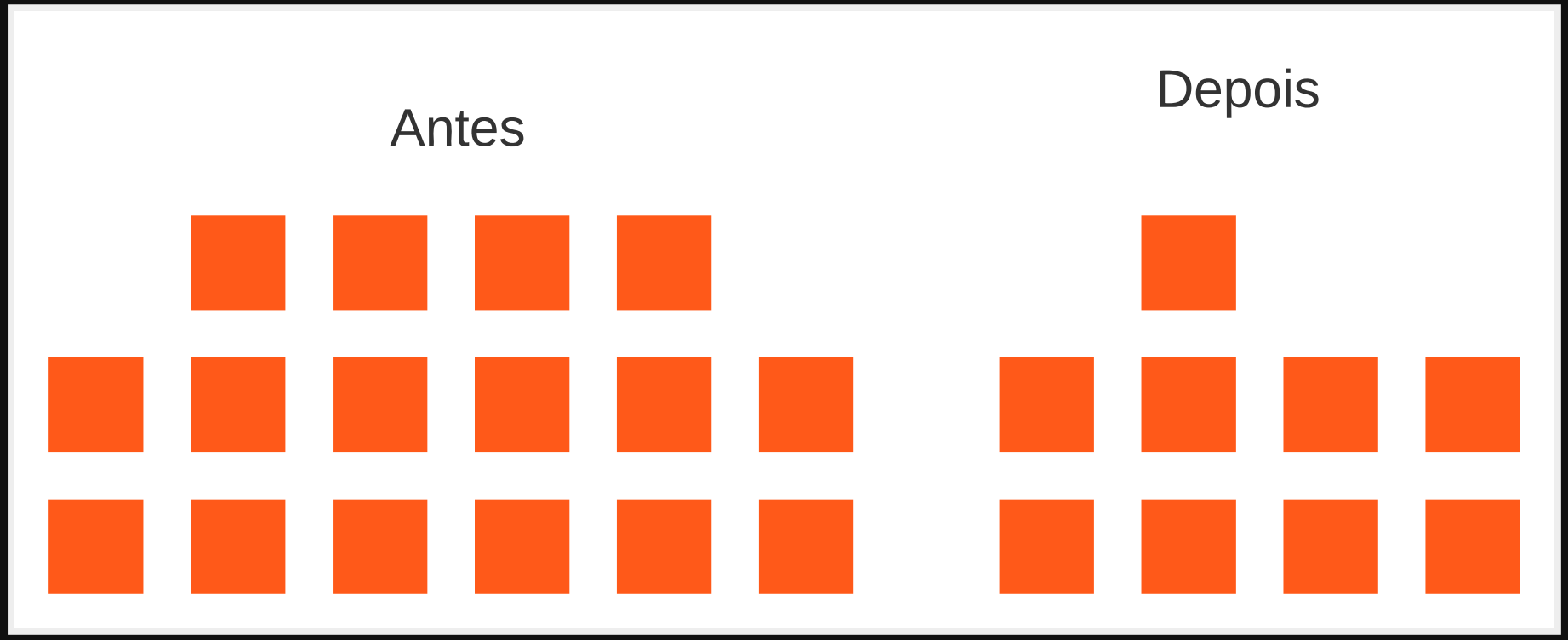
Ruby: 500 MB



Go: 10 MB



Elasticserach



Reflexões

Não se apegue a tecnologias.

Ruby é incrível, mas não precisa resolver tudo.

Go é só uma das opções.

Os problemas de amanhã serão maiores que os de hoje.

Rust está vindo com tudo.

Sempre tem como melhorar.

Ainda podemos:

Usar HTTP2 ao invés de HTTP1;

Usar gRPC como protocolo de comunicação;

Criar estratégias agressivas de cache;

Trabalhar com WebAssembly no navegador.

Não vai ficar perfeito.

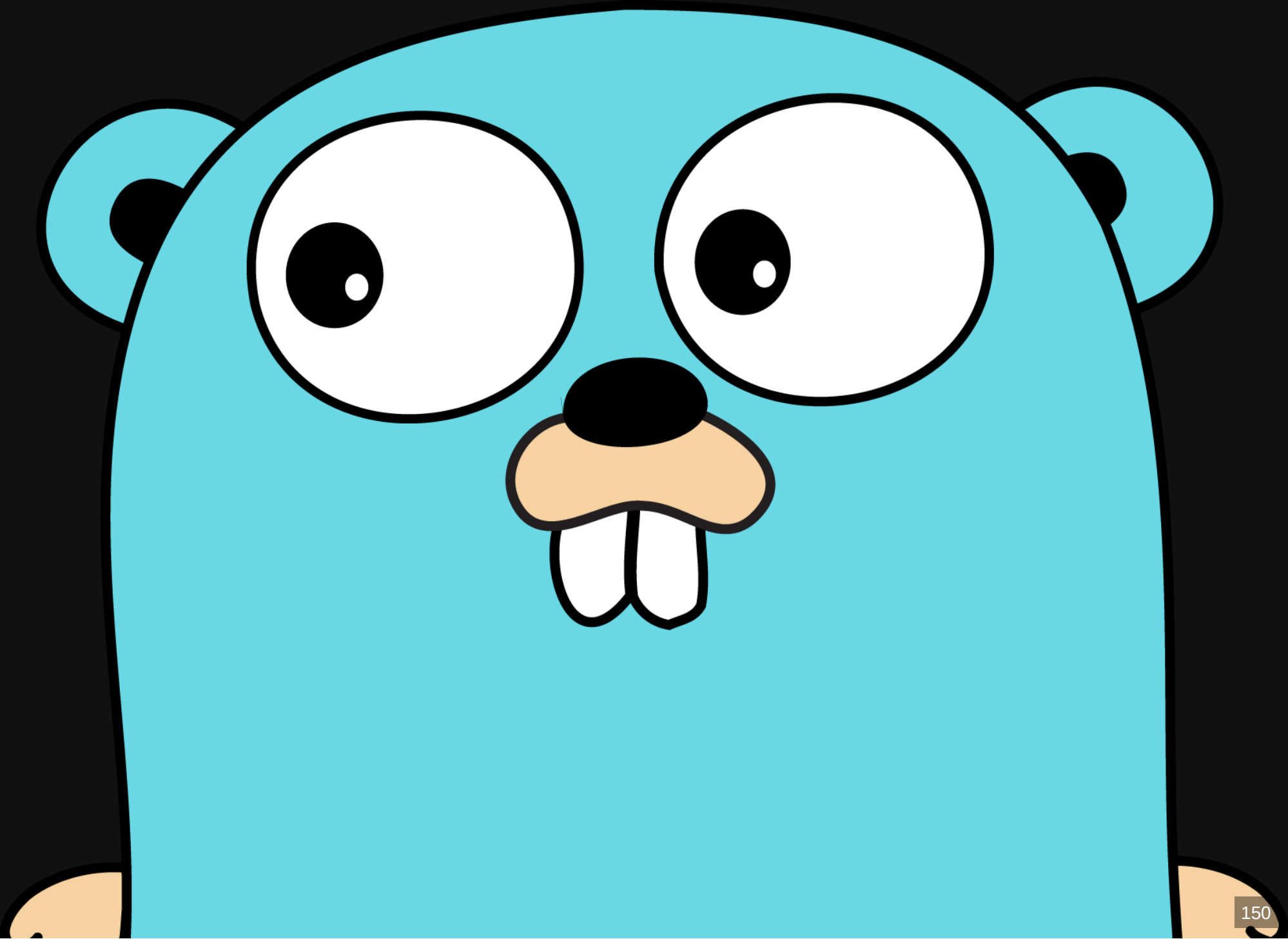
E tudo bem.

```
query(
  $products: ProductSearch, $first: Int, $after: ID
) {
  search(search: $products) {
    term
    products(first: $first, after: $after) {
      edges { node { title } cursor }
    }
  }
}
```

```
query(
  $products: ProductSearch, $users: UserSearch,
  $first: Int, $after: ID
) {
  search(products: $products, users: $users) {
    term
    products(first: $first, after: $after) {
      edges { node { title } cursor }
    }

    users(first: $first, after: $after) {
      edges { node { name } cursor }
    }
  }
}
```

Estudar novas linguagens abre a sua mente e acaba sendo divertido.

Não tenha medo.



Obrigado!

https://github.com/gbaptista

https://www.linkedin.com/in/guilhermebaptista