Estruturas de dados probabilísticas (Sketch Data Structures) e Go

(cc) Gleicon - gleicon@gmail.com

Introdução

Este ebook é um trabalho em progresso. Eu vou adicionando informações e modificando de acordo com meu aprendizado e também com novidades que encontro, que tenho costume de anotar. A minha intenção é ter um guia rápido em português que possa ajudar a resolver alguns problemas de engenharia de dados com Go. Os exemplos de código estão no Github junto ao fonte do livro em https://github.com/gleicon/ebook-go-sketch.

Minhas anotações começaram como meu diário de uso de Go para serviços de uso bem especifico: contar, acumular e registrar grandes volumes de dados em alta velocidade. Estes problemas são descritos há muito tempo e aparecem de várias formas: brokers de mensagens, servidores de coleta de dados, pixel de tracking para trafego web, coleta de metricas em dispositivos móveis, monitoração de serviços, entre outros.

Meu primeiro contato com Go foi em um curso em 2011 na OSCON, uma conferencia da O'Reilly. Eu havia lido sobre a linguagem e fiquei interessado na simplicidade. Nesta época, eu programava em Python usando o framework Twisted para fazer meus servidores. "Programava" é um eufemismo para o pouco que conseguia fazer entre outras atribuições. Após este curso eu tentei portar alguns projetos para Go sem muito compromisso.

Meu jeito de aprender uma linguagem ou framework é fazer uma versão simples do projeto Memcached e do meu projeto RESTMQ, para entender como fazer um servidor, como atender requisições, como guardar dados e serialização. Eles foram bem mas continuei sem compromisso até que o meu time resolveu um problema de roteamento de email em larga escala e volume em Go. Este projeto me chamou a atenção pela simplicidade e escalabilidade. Na época usando maquinas fisicas a redução de uso de recursos foi impressionante.

Comecei a investir tempo ajudando em alguns projetos opensource para pegar o jeito da linguagem, depois portei e criei pequenos projetos e tenho usado Go desde então.

O mesmo aconteceu com estruturas de dados probabilísticas. Eu vou contar como tive contato com elas, minha motivação e explica-las de uma forma simples com referências para quem se interessar em profundidade, para manter este livro breve. Em cada uma vou colocar referências para o artigo ou texto que utilizei para entender como funcionam. Para manter este ebook em um tamanho razoável vou usar um projeto meu como exemplo prático de aplicação.

Nesta jornada eu tive vários momentos que só posso descrever como "*solução a procura de um problema*": as vezes lia um artigo ou via um exemplo de código e revisava meus projetos e o trabalho a procura de uma oportunidade de usar a idéia.

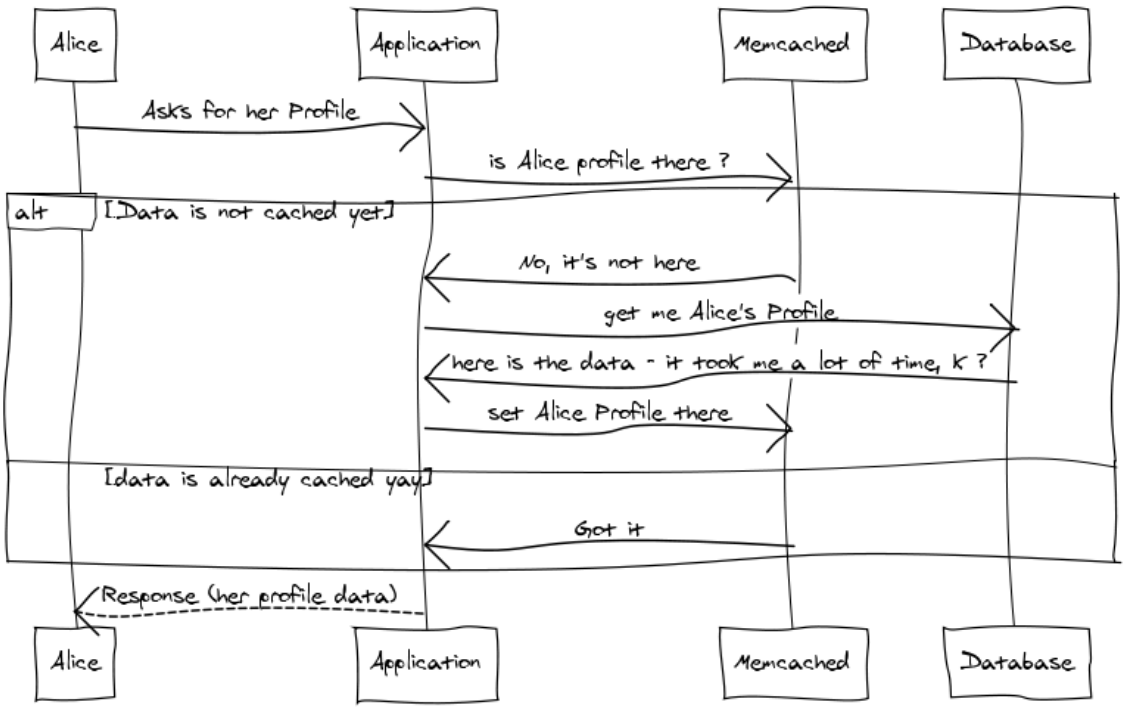
Cache

Meu primeiro contato com estrutura de dados probabilisticas foi provavelmente como o da maioria das pessoas em 2011, um post descrevendo como o Google Chrome usava Bloom Filters para seu recurso de "Safe Browsing". Hoje em dia o Chrome utiliza outra técnica que envolve uma estrutura chamada Prefix Set e você pode ler a transição aqui.

A idéia de ter um "cache" (estrutura que armazena valores pré calculados em memória ou disco local) não é nova, frameworks e aplicações fazem isso há algum tempo. O projeto SQLLite é usado pelo Chrome desta maneira e para a maioria das aplicações móveis.

Existem servidores como Memcached e Redis que provêm o serviço de guardar esta estrutura de forma acessivel para mais de uma instância da mesma aplicação, que é essencialmente um servidor com um "Map" (dicionario) que disponibiliza suas funções de Get e Set pela rede usando um protocolo de comunicação definido.

Para ilustrar este fluxo vou usar a figura abaixo:



Este fluxo é simples: Se imaginarmos um sistema em que uma usuária precisa acessar seu "Profile" guardado em um banco de dados para mostrar seus dados você pode acessar diretamente o banco em todas as requisições gerando I/O e uso de CPU concorrente com outras requisições a este banco.

Se analisarmos o padrão das solicitações a este banco de dados e como suas informações são atualizadas, veremos que nem todas as tabelas mudam em conjunto. Se modificarmos o fluxo de acesso podemos guardar a resposta a uma requisição em um sistema de cache e da proxima vez servi-lo da memória. Isso exige que os dados "expirem" ou se invalidem automaticamente para garantir que modificações sejam vistas (troca de sobrenome ou endereço por exemplo).

Esta modificação adiciona um componente novo ao sistema e intercepta as requisições para verificar se a resposta a uma requisição já existe no cache. Isso é feito usando o modelo "K/V -

Key Value" (Chave/Valor). Esta é a assinatura de uma estrutura de dados do tipo Hash ou Dicionário. Você pode usar a query como Chave, e receber seu resultado como Valor.

Esta explicação é importante para voltarmos ao Bloom Filter e ao exemplo do SafeBrowsing. O Chrome tem vários caches que crescem em "velocidades" distintas. Safe Browsing é uma lista que o Google mantém baseado em duas pesquisas e reclamações recebidas sobre sites inseguros. Para que o browser não tivesse que consultar um serviço web a todo momento, com implicações de performance e segurança, foi decidido que esta lista poderia ser atualizada por download de tempos em tempos.

A lista poderia ficar - e ficou - maior que o que a memória de um processo poderia guardar, então guarda-la em um arquivo para consulta rápida foi considerado. Mas guardar as URLs inteiras seria um problema pelo tamanho que elas podem ter. Foi decidido guardar um Hash de cada URL, ou seja aplicada uma função em cada URL que mapearia um texto a um valor de tamanho fixo. Se usarmos os dados do HTTP Archive e colocassemos 25% das URLs existentes nesta lista, ficaria complexo conferir a cada request se uma URL esta na lista de URLs do safe browsing.

Á epoca foi decidido colocar esta lista em uma estrutura chamada Bloom Filter que depois mudou para outra estrutura por conta do tamanho em disco e outras limitações descritas no link acima. Mas vamos focar no Bloom Filter.

Bloom Filter

Bloom filter é uma estrutura de dado probabilística de baixo consumo de espaço e alta velocidade, que deixa testar a probabilidade de um membro pertencer a um conjunto. Para entender o bloom filter precisamos entender seus componentes:

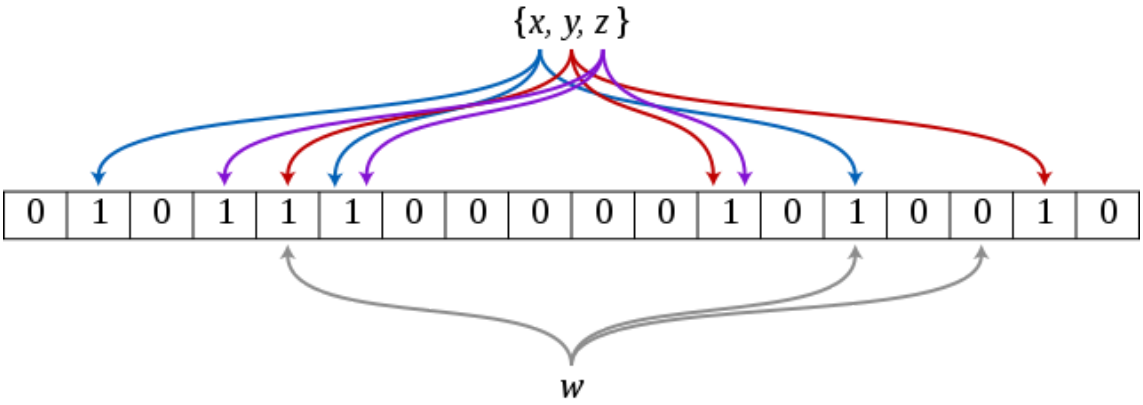
O QUE É UM CONJUNTO (SET)?

Um conjunto é uma estrutura de dados que de forma simplificada guarda um item único por posição. Conjuntos podem ser combinados, comparados e subtraidos (Union, Compare, Difference). No Bloom filter a estrutura usada é um BitSet, Conjunto de bits. Esta estrutura também é chamada de Bit Array e é uma lista grande em que cada posição representa um bit.

FUNÇÕES DE HASH

Dado um item a ser inserido, deve ser calculado seu Hash usando funções que vão mapear o item a varias posições do BitSet mencionado anteriormente.

Fica mais facil visualizar com um diagrama (fonte: wikipedia)



Os elementos **{x,y,z}** foram adicionados no BitSet. As setas coloridas sao as funções de hash utilizadas para modificar os bits no BitSet (a lista de 0 e 1). O elemento **{w}** não está no BitSet.

O BitSet resultante das adições cria uma situação interessante. O Bloom Filter tem *quase certeza* de que os elementos **{x,y,z}** estão lá. Mas tem certeza absoluta de que **{w}** não está representado. Porque quase certeza ? Existe a chance de que se procurarmos um elemento não existente no BitSet, uma de suas funções Hash indique um bit que esta em `1`.

Esta é uma característica do Bloom Filter. Ele pode dar falsos positivos (afirmar que um item existe mas ele não ter sido inserido) mas nunca dá falsos negativos. A chance de falso positivos aumenta conforme aumenta o número de elementos adicionados ao Bloom Filter.

Esta caracteristica se junta à velocidade e economia de espaço como pontos de escolha do algoritmo. O Bloom Filter não permite que um elemento seja "Deletado".

A tabela abaixo compara Bloom Filter com um Hash/Dictionary/Map nestas caracteristicas:

Tipo	Velocidade	Guarda todos os valores (uso de espaço)	Falso Positivos	Falso Negativo
Hash	Pior caso tem que percorrer todas as chaves	sim	não	não
Bloom Filter	Pior caso é relacionado ao tamanho do BitSet	Usa a representação menor dos valores com hash functions	sim	não

Vamos usar uma biblioteca para testar o Bloom Filter. Escolhi a biblioteca do site YourBasic pois é simples de usar e visualizar a implementação. O código fonte da biblioteca está em https://github.com/yourbasic/bloom. O playground para este codigo fica em https://play.golang.org/p/tDnQrVV3xBS.

```
package main

import (
   "fmt"

   "github.com/yourbasic/bloom"
)

var safeBrowsingList *bloom.Filter

func testAndReport(url string) {
   if safeBrowsingList.Test(url) {
       fmt.Println(url, "is not safe")
   } else {
```

```
        fmt.Println(url, "seems safe")
    }
}

func main() {
    // 1000 elementos, erro de 1/20 (0.5%)
    safeBrowsingList = bloom.New(1000, 20)

    safeBrowsingList.Add("https://badsite.com")
    safeBrowsingList.Add("https://anotherbadsite.com")

    fmt.Printf("Sites: %d\n", safeBrowsingList.Count())

    testAndReport("https://lerolero.com")
    testAndReport("https://badsite.com")
}
```

O código acima cria um Bloom Filter com 1000 posições e uma taxa de falsos positivos estimada de 1 em 20. Este número é utilizado para calcular quantos *lookups*(passadas ou buscas) serão feitas no bitset ao adicionar ou testar um item. É importante pois junto com as funções de Hash e o tamanho do BitSet ajuda a controlar a taxa de falso positivos.

Esta biblioteca não implementa uma forma fácil de serialização de dados. Serializar dados é um modo de transformar uma estrutura de dados em um formato que pode ser guardado em um arquivo ou memória e recuperado posteriormente. Isso nos ajudaria a criar um Bloom Filter em um lugar e replica-lo para outro, como o Chrome fazia.

Uma aplicação interessante de Bloom Filters é em banco de dados. O Cassandra utiliza Bloom Filters no caminho de leitura, pois consolida dados em disco com memória. Para evitar acesso ao disco e um full scan (procurar um dado em todas as tabelas em disco) foi implementado um Bloom Filter para criar uma barreira de acesso https://cassandra.apache.org/doc/latest/operating/bloom_filters.html.

O LevelDB do Google, um banco de dados local chave/valor também utiliza Bloom Filters para mapear os blocos em disco em uma estrutura que chega a reduzir em 100 vezes a necessidade de I/O https://github.com/google/leveldb/blob/master/doc/index.md.

Se você se interessar por mais detalhes sobre Bloom Filters, a página da Wikipédia https://en.wikipedia.org/wiki/Bloom_filter tem um conteúdo interessante, que explica o artigo original e detalha suas configurações.

Cuckoo Filter

O Cuckoo Filter vive na mesma categoria que o Bloom Filter, é uma implementação das mesmas idéias mas que permite a remoção de um elemento e implementa pequenas mudanças que ajudam a diminuir os falsos positivos. Existem implementações de Bloom Filter que permitem remover itens também com uma troca de eficiencia ou espaço ocupado.

O Bloom Filter é mais eficiente em espaço e busca para largos volumes de dados mas as aplicações são semelhantes, você poderia trocar um pelo outro. A vantagem é a remoção de elementos e a melhora dos falsos positivos, que aumentam no Bloom Filter conforme mais dados são armazenados.

Este artigo https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf explica como o Cuckoo Filter implementa suas funções de hash e correção de erros. Como no exemplo anterior vou usar

uma biblioteca https://github.com/seiflotfy/cuckoofilter. Você pode rodar o exemplo em https://play.golang.org/p/zVIbXIbgSMl.

```
package main

import (
    "fmt"

    cuckoo "github.com/seiflotfy/cuckoofilter"
)

var safeBrowsingList *cuckoo.Filter

func testAndReport(url string) {
    uu := []byte(url)
    if safeBrowsingList.Lookup(uu) {
        fmt.Println(url, "is not safe")
    } else {
        fmt.Println(url, "seems safe")
    }
}

func main() {
    safeBrowsingList = cuckoo.NewFilter(1000)
    safeBrowsingList.InsertUnique([]byte("https://badsite.com"))
    safeBrowsingList.InsertUnique([]byte("https://anotherbadsite.com"))

    testAndReport("https://badsite.com")
    testAndReport("https://anotherbadsite.com")
    testAndReport("https://lerolero.com")

    count := safeBrowsingList.Count()
    fmt.Printf("Items: %d\n", count)

    // Delete a string (and it a miss)
    safeBrowsingList.Delete([]byte("hello"))

    count = safeBrowsingList.Count()
    fmt.Printf("Items: %d\n", count)

    // Delete a string (a hit)
    safeBrowsingList.Delete([]byte("https://badsite.com"))

    count = safeBrowsingList.Count()
    fmt.Printf("Items: %d\n", count)

    safeBrowsingList.Reset() // reset

    count = safeBrowsingList.Count()
    fmt.Printf("Items: %d\n", count)
}
```

O código é parecido com o que usei para mostrar o Bloom Filter, com a contagem de elementos no Cuckoo Filter entre as operações e também com a operação de DELETE de um item. Esta biblioteca fornece funções de serialização/deserialização também. O código é bem interessante de ler.

Posso alterar meu código para guardar o filtro e carregar depois com as funções **Encode** e **Decode** https://play.golang.org/p/urTVjJX6xHP.

```
package main

import (
    "fmt"

    cuckoo "github.com/seiflotfy/cuckoofilter"
)

var safeBrowsingList *cuckoo.Filter

func testAndReport(filter *cuckoo.Filter, url string) {
    uu := []byte(url)
    if filter.Lookup(uu) {
        fmt.Println(url, "is not safe")
    } else {
        fmt.Println(url, "seems safe")
    }
}

func main() {
    safeBrowsingList = cuckoo.NewFilter(1000)
    safeBrowsingList.InsertUnique([]byte("https://badsite.com"))
    safeBrowsingList.InsertUnique([]byte("https://anotherbadsite.com"))

    testAndReport(safeBrowsingList, "https://badsite.com")
    testAndReport(safeBrowsingList, "https://anotherbadsite.com")
    testAndReport(safeBrowsingList, "https://lerolero.com")

    count := safeBrowsingList.Count()
    fmt.Printf("Items: %d\n", count)

    // Delete a string (and it a miss)
    safeBrowsingList.Delete([]byte("hello"))

    count = safeBrowsingList.Count()
    fmt.Printf("Items: %d\n", count)

    fmt.Println("Encoding")

    serFilter := safeBrowsingList.Encode()

    fmt.Printf("Serialized: % x\n", serFilter)

    BackupsafeBrowsingList, _ := cuckoo.Decode(serFilter)

    count = BackupsafeBrowsingList.Count()
```

```
    fmt.Printf("Items: %d\n", count)

    testAndReport(BackupsafeBrowsingList, "https://badsite.com")
    testAndReport(BackupsafeBrowsingList, "https://anotherbadsite.com")
    testAndReport(BackupsafeBrowsingList, "https://lerolero.com")
}
```

Eu não gravei o filter em um arquivo, mas poderia ter feito com poucas modificações. Na parte final do programa acima usei a função Decode() do tipo *cuckoo.Filter* para gerar um "dump" que pode ser reconstruido. Em Go poderiamos usar um protocolo de serialização como Gob ou binary diretamente também, implementando as funções no tipo.

Minha intenção foi comparar os usos e demonstrar como a remoção de um elemento deste filter funciona. Na saida do programa você pode ver o tamanho da estrutura de dados, sabendo que mesmo que adicionasse mais sites, este tamanho não mudaria.

HyperLogLog

Existe um problema em ciência da computação chamado *count-distinct*- ou estimativa de cardinalidade que procura soluções para encontrar o número de elementos distintos (únicos) em um stream (sequência) de dados que pode conter elementos repetidos. Um pouco a frente vou falar de um problema que tivemos para contar um grande volume de clicks, ou visitas de dispositivos, mantendo contadores totais e de visitas únicas entre mais de 40 trilhões de documentos.

Este problema tem aplicações bem interessantes em IoT, monitoração, data science e analytics e com o crescente volume de dados é um grupo de algoritmos interessante de estudar para entender o impacto de performance e precisão das soluções existentes.

Produtos como Elasticsearch e InfluxDB utilizam o algoritmo HyperLogLog++ para estimar a cardinalidade de agregações e conjuntos de dados acima de um certo número.

Em https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics-cardinality-aggregation.html podemos ver que agregações são guardadas em uma estrutura chamada HyperLogLog++. A documentação fala sobre cardinalidade, o tamanho do conjunto de documentos e também sobre a precisão.

No InfluxDB as metricas internas também usam HyperLogLog para prover contadores de monitoração https://docs.influxdata.com/platform/monitoring/influxdata-platform/tools/measurements-internal/. O Redis provê um tipo baseado em HLL(abreviação para HyperLogLog) nos comandos iniciados com PF*.

O Google publicou um artigo sobre este algoritmo com melhorias para seus casos e comparação com o algoritmo original https://research.google/pubs/pub40671/. Pesquisadores continuam implementando mudanças para casos especificos como este conjunto de modificações descritas nestes slides https://csqjxiao.github.io/PersonalPage/csqjxiao_files/papers/INFOCOM17-slides.pdf.

A biblioteca que vou usar é baseada neste ultimo trabalho , https://github.com/axiomhq/hyperloglog e provê uma implementação interessante de HLL, com serialização e otimização da função de hash com redução do espaço utilizado.

Você pode rodar o exemplo abaixo no play: https://play.golang.org/p/S5cHfBGLpcF.

```
package main

import (
```

```
    "fmt"
    "strconv"

    "github.com/axiomhq/hyperloglog"
)

func estimateError(got, exp uint64) float64 {
    var delta uint64
    if got > exp {
        delta = got - exp
    } else {
        delta = exp - got
    }
    return float64(delta) / float64(exp)
}

func main() {
    axiom := hyperloglog.New16()

    step := 10
    unique := map[string]bool{}

    for i := 1; len(unique) < 10000000; i++ {
        str := "stream-" + strconv.Itoa(i)
        axiom.Insert([]byte(str))
        unique[str] = true

        if len(unique)%step == 0 || len(unique) == 10000000 {
            step *= 5
            exact := uint64(len(unique))
            res := axiom.Estimate()
            ratio := 100 * estimateError(res, exact)
            fmt.Printf("Exact count %d \nHLL count %d (%.4f%% off)\n\n", exact,
res, ratio)
        }
    }

    data2, err := axiom.MarshalBinary()
    if err != nil {
        panic(err)
    }
    fmt.Println("HLL total size:\t", len(data2))
    fmt.Println("Map total size:\t", len(unique))

}
```

Este exemplo foi adaptado da documentação do código, e compara duas estruturas: um slice de 10MM de itens com um bool para cada item, e um HLL. Se executarmos o programa:

```
$ go run hll.go
Exact count 10
HLL count 10 (0.0000% off)
```

```
Exact count 50
HLL count 50 (0.0000% off)

Exact count 250
HLL count 250 (0.0000% off)

Exact count 1250
HLL count 1250 (0.0000% off)

Exact count 6250
HLL count 6250 (0.0000% off)

Exact count 31250
HLL count 31253 (0.0096% off)

Exact count 156250
HLL count 155914 (0.2150% off)

Exact count 781250
HLL count 778300 (0.3776% off)

Exact count 3906250
HLL count 3874441 (0.8143% off)

Exact count 10000000
HLL count 9969753 (0.3025% off)

HLL total size:     32776
Map total size:     10000000
```

Quanto mais itens são adicionados no HLL, aumenta o erro ao estimar o tamanho. Com 10MM de itens a diferença é de 0.3025% em relação ao slice com todos os itens. Se compararmos o tamanho em bytes, o HLL tem 32776 bytes depois de serializado. O slice de booleans tem 10MM itens * 1byte, quase 10MB.

Para estimar *Visitantes Unicos* baseados em endereço IP ou outra forma de fingerprint em um sistema como de Analytics este é um trade-off interessante. Como mencionei acima, o Elasticsearch utiliza um HLL configuravel para guardar agregações salvando espaço de disco, memoria e banda de rede em troca de um tradeoff configuravel de precisão.

A função de *Merge* de dois ou mais HLL é interessante. Você pode coletar dados de stream distintos, sem usar um lock ou banco centralizado e periodicamente juntar (merge) os HLL para ter um resultando que contem a estimativa de todos os itens vistos em instancias distintas.

O que procurar em uma biblioteca?

Neste ponto quero explicar o que procuro em uma biblioteca que oferece este tipo de estrutura de dados.

Quando aprendo uma nova estrutura de dados eu procuro saber dos *trade-offs*, o que ela oferece de vantagem e o que preciso entender que não terei em relação a outras estruturas que conheço melhor. Procuro entender se as implementações vão permitir que consiga serializar e deserializar.

Também procuro o termo "Mergeable" ou "Merge" que significa que posso juntar mais de uma instância deste tipo. Saber se os elementos podem ser deletados de alguma maneira vai me dizer um pouco disso também.

Olhando o código da biblioteca e conhecendo um pouco como Go funciona, eu tento ver como a implementação infere tipos - se isso acontece - e como usa reflection, uma técnica de inspeção que pode afetar a performance.

Depois disso tento fazer um exemplo relacionando com uma estrutura conhecida, como comparei Bloom Filter com Map.

Com isso vou melhorando meu entendimento e consigo interpretar melhor o artigo ou origem da estrutura. Eu mantenho alguns projetos que facilitam este entendimento e vou usar um deles para contextualizar as estruturas que vimos até agora e como uso outra estrutura interessante, o HyperLogLog.

Estudo de caso

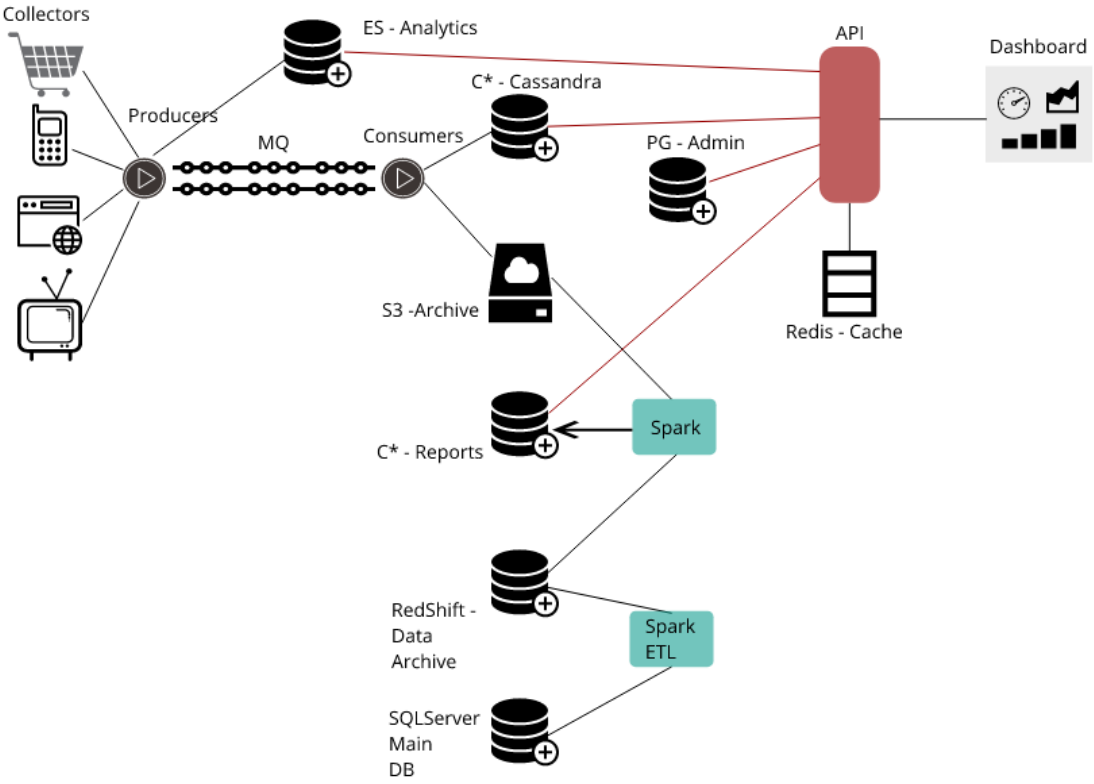
A minha motivação para usar HyperLogLog foi um estudo para armazenar dados de clickstream em um projeto que utilizava o Elasticsearch. O volume de dados armazenados era grande (mais de 40 trilhões de documentos) e a maioria das pesquisas eram contadores e sumarizações.

Com o crescimento do produto a solução de guardar documentos no Elasticsearch e solicitar agregações ficou insustentável. O cluster estava grande, caro e os problemas aconteciam todo dia. Pensamos em pré-calcular algumas agregações, usar contadores e procurar uma alternativa com os mesmos principios para não causar um grande impacto na arquitetura existente.

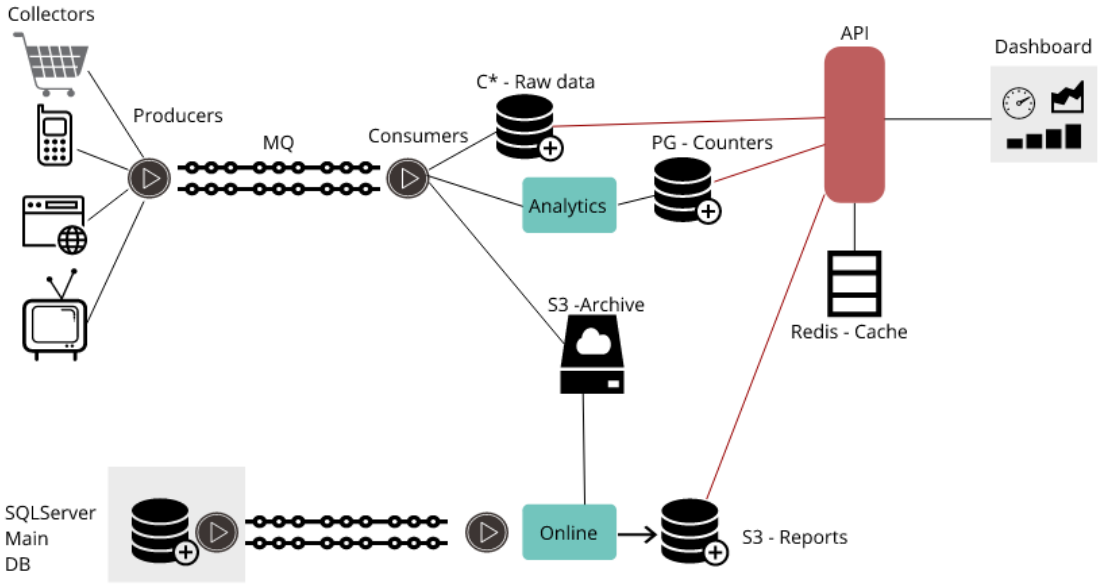
A documentação do Elasticsearch indicava que acima de um certo numero de documentos, as agregações eram gaurdadas em um HLL. Como o Redis oferece um tipo baseado em HyperLogLog https://redis.io/commands/pfcount pensamos em modificar nosso código para testar se teriamos uma arquitetura mais resiliente.

Uma das preocupações com o volume de dados que operavamos é o backfill em caso de problemas, e haviamos passado um problema semelhante em que perdemos um cluster grande de ScyllaDB, um banco de dados distribuidos. Encontramos todo tipo de gargalo com maquinas virtuais para reabastecer o cluster com os dados que precisavamos. Um rebalanceamento normal do cluster durou 18h e nem foi uma copia completa de dados.

A arquitetura do sistema era em streaming e a idéia é que contadores simples (um numero incremental) não ajudaria em consultas especificas de agregações em atributos como endereço IP de um click. Para ilustrar coloquei um diagrama abaixo:



Esta era a arquitetura antiga, incluindo ETLs e bancos de dados diversos. É um caso de feature creep interessante pois além de código bancos de dados foram acompanhando o crescimento do produto. Entre os repositórios de dados existiam Cassandra, S3, Elasticsearch e PGSQL. Os coletores de dados produziam mais de 15 mil documentos por segundo, e a retenção dos documentos variava de 15 a 90 dias. Para um volume pequeno de documentos (até um ou 2 bilhões) um cluster grande ainda era eficiente mas com mais de 40 trilhões e um volume alto de tráfego de rede todos os elementos desta arquitetura eram afetados.



Além da refatoração para remover alguns bancos de dados e o uso de eventos entre os produtos, a função do Elasticsearch foi inicialmente movida para um PGSQL como contadores. Essa arquitetura tem muitos elementos da Arquitetura Lambda.

Inicialmente tentamos guardar dados de um determinado periodo em um Redis e nas primeiras modelagens vimos que o consumo de memória era grande, e o tempo para fazer *backfill* (restaurar ou preencher uma nova instancia) era de dias. Em paralelo fiz um teste de usar uma implementação em Go do HyperLogLog para testar se conseguiria serializar os contadores e ter uma abordagem diferente do Redis, que quando persiste os dados em disco usa apenas um arquivo com extensão .rdb.

A solução que implementamos após estas pesquisas foi hibrida, utilizou um banco de dados relacionais para contadores e um ElasticSearch bem menor com expiração de documentos pelo *curator*, modificamos algumas características do produto para refletir a margem de erro que existia.

Eu continuei trabalhando naquele código que simulava o Redis e expandindo os comandos. Era uma plataforma boa para conhecer melhor estas estruturas de dados probabilísticas e inventar um Redis que não tivesse muita certeza das coisas, uma alusão aos trade-offs destas estruturas em favor de espaço e velocidade.

Nazaré

Meu objetivo ao usar o Cuckoo Filter foi recriar este servidor de cache probabilistico, usando um protocolo conhecido e que me permitisse "trocar" o cache com uma operação apenas. Parece complicado mas a idéia é simples: Ao serializar um Cuckoo Filter com os dados que preciso consultar e gravar em disco, posso copiar com ferramentas simples entre containers. O tamanho do arquivo será pequeno, a eficiencia é alta e não preciso implementar nada mais complexo que "treinar" o filtro e distribui-lo.

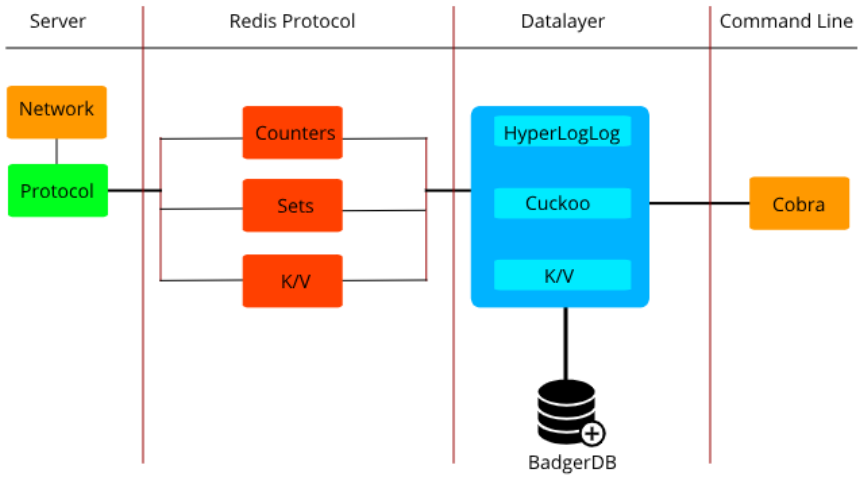
Além disso usar um protocolo conhecido facilita a fazer um "drop in replacement" de serviços como Memcached e Redis sem ter que inventar uma semantica nova, só alterando o comportamento interno do servidor. É a minha maneira de relacionar algo novo com o comportamento de um sistema que já conheço. Este artificio já me ajudou ao trabalhar com sistemas legados em que eu não tinha outra saida a não ser clients que falavam o protocolo Memcached, por exemplo.

Este experimento virou um projeto chamado Nazaré, que está em meu github https://github.com/gleicon/nazare e utilizo mostrar estes algoritmos na pratica.

A estrutura deste serviço é simples. É um servidor que entende o protocolo do Redis e implementa poucos dos seus comandos. Para cada grupo de comando escolhi um algoritmo probabilistico e como persistencia de dados utilizei o BadgerDB - https://github.com/dgraph-io/badger.

Depois de algum tempo mantendo meu código de rede e o parsing dos comandos do Redis decidi usar uma biblioteca que implementa seu protocolo de maneira simples https://github.com/tidwall/redcon.

Para facilitar os testes eu separei o projeto em modulos que usei para criar um command line, *nazare-cli*, que tem os mesmos tipos apresentados no servidor. Para isso usei a biblioteca Cobra https://github.com/spf13/cobra.



COMANDOS REDIS IMPLEMENTADOS E BIBLIOTECAS UTILIZADAS

Comando	Algoritmo	Biblioteca
GET	K/V Storage	Armazenamento no BadgerDB
SET	K/V Storage	Armazenamento no BadgerDB

DEL	K/V Storage	Armazenamento no BadgerDB
PFADD	HyperLogLog	github.com/axiomhq/hyperloglog
PFCOUNT	HyperLogLog	github.com/axiomhq/hyperloglog
SADD	Cuckoo Filter	github.com/seiflotfy/cuckoofilter
SREM	Cuckoo Filter	github.com/seiflotfy/cuckoofilter
SCARD	Cuckoo Filter	github.com/seiflotfy/cuckoofilter
SISMEMBER	Cuckoo Filter	github.com/seiflotfy/cuckoofilter

ORGANIZAÇÃO DO CÓDIGO

O código do Nazaré é organizado por modulos: server, datalayer, counters, sets e db. Cada modulo entende uma parte do modelo de dados. Eu usei as funções de serialização de cada biblioteca para gerar um byte slice e guardar no BadgerDB como valor de uma chave: https://github.com/gleicon/nazare/blob/master/counters/hllcounters.go#L117-L153

```
/*
IncrementCounter increments <<name>> counter by adding <<item&gt>> to it.
The naive implementation locks(), get, increment and set
The counter and its lock are automatically created if it is empty.
*/
func (hc *HLLCounters) IncrementCounter(key []byte, item []byte) error {
    if hc.hllrwlocks[string(key)] == nil {
        hc.hllrwlocks[string(key)] = new(sync.RWMutex)
    }

    localMutex := hc.hllrwlocks[string(key)]
    localMutex.Lock()
    defer localMutex.Unlock()

    cc, _ := hc.datastorage.Get([]byte(key))

    hll := hyperloglog.New16()
    if cc != nil {
        if err := hll.UnmarshalBinary(cc); err != nil {
            return err
        }
    } else {
        hc.stats.ActiveCounters++
    }

    hll.Insert(item)
    var bd []byte
    var err error

    if bd, err = hll.MarshalBinary(); err != nil {
```

```
        return err
    }

    if err = hc.datastorage.Add([]byte(key), bd); err != nil {
        return err
    }
    return nil
}
```

O mesmo padrão é utilizado para Sets com Cuckoo Filter
https://github.com/gleicon/nazare/blob/master/sets/cuckoo.go#L48-L82

```
/*
SAdd a member to a set
*/
func (ckset *CkSet) SAdd(key, member []byte) error {
    var sts *cuckoo.Filter

    localMutex := ckset.lockKey(key)
    localMutex.Lock()
    defer localMutex.Unlock()

    value, err := ckset.datastorage.Get(key)
    if err != nil {
        return errors.New("Error fetching set: " + string(key))
    }
    if value == nil {
        // tunable cuckoo size
        sts = cuckoo.NewFilter(1024 * 1024)
    } else {
        sts, err = cuckoo.Decode(value)
        if err != nil {
            return errors.New("Error decoding filter set: " + string(key))
        }
    }
    ok := sts.InsertUnique(member)
    if !ok {
        return errors.New("Error inserting at cuckoo set, key: " + string(key))
    }

    err = ckset.datastorage.Add(key, sts.Encode())
    if err != nil {
        return errors.New("Error inserting at datastore, key: " + string(key))
    }

    return nil
}
```

Nos dois casos a abstração do banco de dados (`datastorage`) conhece como armazenar o valor em uma chave. Se eu optasse por trocar o BadgerDB por outro banco, teria que implementar o método seguindo a interface em https://github.com/gleicon/nazare/blob/master/db/datastore.go

```
package db
/*
Datastorage describes the generic interface to store counters.
*/
type Datastorage interface {
    Add([]byte, []byte) error
    Get([]byte) ([]byte, error)
    Delete(\[\]byte) (bool, error)
    Close()
    Flush()
}
```

A implementação da interface de Add para o BadgerDB em https://github.com/gleicon/nazare/blob/master/db/badger.go#L61-L81

```
/*
Add data
*/
func (bds *BadgerDatastorage) Add(key, value []byte) error {
    var vals []byte
    if bds.compression {
        vals := snappy.Encode(nil, value)

        if vals == nil {
            return errors.New("Error compressing payload")
        }
    } else {
        vals = value
    }

    err := bds.db.Update(func(txn *badger.Txn) error {
        err := txn.Set([]byte(key), vals)
        if err != nil {
            return err
        }
        return nil
    })
    return err
}
```

Esta implementação de datalayer usando o BadgerDB permite compressão dos dados com Snappy. No mesmo package tem a implementação em memória do datalayer https://github.com/gleicon/nazare/blob/master/db/mapds.go

```
/*
Add a new key
*/
func (hds *HLLDatastorage) Add(key []byte, payload []byte) error {
    hds.bytemap[string(key)] = payload
    return nil
}
```

As implementações são independentes das escolhas de tipos de dados em camadas superiores. As mesmas implementações são usadas para a versão command line https://github.com/gleicon/nazare/blob/master/nazare-cli/cmd/root.go#L43-L52

```
func init() {

    cobra.OnInitialize(initConfig)

    rootCmd.PersistentFlags().StringVar(&cfgFile, "config", "", "config file
(default is $HOME/.nazare-cli.yaml)")

    dbPath := rootCmd.Flags().StringP("database", "b", "nazare.db", "full
database path and name. defaults to nazare.db at local dir")
    ldb = datalayer.NewLocalDB()
    ldb.Start(*dbPath)
}
```

E assim no comando de adicionar um elemento a um cuckoo filter https://github.com/gleicon/nazare/blob/master/nazare-cli/cmd/sets.go#L52-L59

```
if addFlag {
        if len(args) < 2 {
            return errors.New("Invalid parameters, Add requires <setname>
<item>")
        }
        if err = ldb.LocalSets.SAdd([]byte(args[0]), []byte(args[1])); err !=
nil {
            return errors.Unwrap(fmt.Errorf("Error adding to set: %w", err))
        }
    }
```

Adicionar uma nova estrutura no datalayer é facil e permite expo-la para o servidor ou command line sem grandes mudanças na camada de command parsing ou storage.

TESTES

Para testar fiz um programa São 8811008 (até minha ultima contagem) que vou inserir duas vezes no Redis e no Nazaré para comparar algumas características. Os IPs não tem importancia, quis usar uma massa de dados que fosse parecida com a realidade. Este programa pode ser alterado facilmente para outros propósitos como testar outros algoritmos de estimativa ou cache.

Se você quiser testar um grande volume de inserções no nazaré ou no redis, eu coloquei um programa que adiciona endereços IP pelo protocolo do Redis, usando as faixas de endereços da AWS disponíveis em https://ip-ranges.amazonaws.com/ip-ranges.json. no repositório junto com um arquivo de configuração do redis simplificado. É um teste que depende da máquina que que vai ser executado e das configurações de arquivos e sockets abertos.

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
```

```
    "log"
    "net"
    "net/http"
    "sync"
    "sync/atomic"
    "time"

    "github.com/mediocregopher/radix"
)

/*
{
      "ip_prefix": "54.239.1.96/28",
      "region": "eu-north-1",
      "service": "AMAZON",
      "network_border_group": "eu-north-1"
    },

*/

// IPRange register
type IPRange struct {
    IPPrefix           string `json:"ip_prefix"`
    Region             string `json:"region"`
    Service            string `json:"service"`
    NetworkBorderGroup string `json:"network_border_group"`
}

// IPRanges struct
type IPRanges struct {
    SyncToken  string    `json:"syncToken"`
    CreateDate string    `json:"createDate"`
    Prefixes   []IPRange `json:"prefixes"`
}

func fetchAWSRanges() (*IPRanges, error) {
    // aws ip ranges
    // https://ip-ranges.amazonaws.com/ip-ranges.json

    var body []byte
    var err error

    httpClient := http.Client{
        Timeout: time.Second * 2, // Timeout after 2 seconds
    }

    req, err := http.NewRequest(http.MethodGet, "https://ip-
ranges.amazonaws.com/ip-ranges.json", nil)
    if err != nil {
        log.Fatal(err)
    }
```

```
    res, err := httpClient.Do(req)
    if err != nil {
        log.Fatal(err)
    }

    if res.Body != nil {
        defer res.Body.Close()
    }

    if body, err = ioutil.ReadAll(res.Body); err != nil {
        return nil, err
    }

    IPR := IPRanges{}
    if err = json.Unmarshal(body, &IPR); err != nil {
        log.Fatalf("unable to parse value: %q, error: %s", string(body),
err.Error())
        return nil, err
    }

    return &IPR, nil
}

func incrementIP(ip net.IP) {
    for j := len(ip) - 1; j >= 0; j-- {
        ip[j]++
        if ip[j] > 0 {
            break
        }
    }
}

func main() {
    var wg sync.WaitGroup
    var ipCount uint64

    RedisConnPool, err := radix.NewPool("tcp", "127.0.0.1:6379", 1000)
    if err != nil {
        log.Fatal(err)
    }
    defer RedisConnPool.Close()

    ipr, err := fetchAWSRanges()
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Ranges: %d\n", len(ipr.Prefixes))

    ipCount = 0
    rangeCount := 0
    for _, ipp := range ipr.Prefixes {
        rangeCount++
```

```
        /* Uncomment to test a shorter run
        if rangeCount > 100 {
            break
        }
        */
        wg.Add(1)
        go func(iprange *IPRange) {
            defer wg.Done()
            ip, ipnet, err := net.ParseCIDR(iprange.IPPrefix)
            if err != nil {
                log.Println(err)
            } else {
                for ip := ip.Mask(ipnet.Mask); ipnet.Contains(ip);
incrementIP(ip) {
                    err := RedisConnPool.Do(radix.Cmd(nil, "PFADD", "IPADDRS",
ip.String()))
                    if err != nil {
                        log.Println(err)
                    }
                    atomic.AddUint64(&ipCount, 1)
                }
            }

        }(&ipp)
    }
    wg.Wait()
    fmt.Printf("Unique IP Addresses: %d\n", ipCount)
}
```

Um dos testes que rodei tinha a intenção de ver é como o banco de dados do Nazaré cresce em relaçao ao do Redis e também como a concorrencia de Go favorece a arquitetura que escolhi. O Redis é bem maduro e não era minha intenção competir com um software que já se provou ha anos em produção, portanto você pode encontrar banco de dados que crescem mais que o redis ou tempos diferentes de inserção.

Bonus: TopK

Eu coloquei um algoritmo como bonus pois ainda estou estudando e fazendo testes simples. Este algoritmo pertence a um grupo chamado de Top-K que são utilizados para encontrar os "Top" K elementos mais frequentes em um stream de dados. Sistemas de coleta e monitoração de métrica podem se beneficiar deste algoritmo para manter um rascunho em tempo quase real de dados em um stream. Meu interesse especifico é deduplicação de dados e criação de padrões para detectar mudança de comportamento.

A biblioteca que estou usando para explorar este algoritmo é https://github.com/axiomhq/topk e um dos artigos em que ela é baseada esta aqui: https://www.hlt.inesc-id.pt/~fmmb/wiki/uploads/Work/misnis.ref0a.pdf. Esta implementação pode ser util para aplicações como load balancers (monitrar em tempo real stream de dados de servidores para definir qual upstream pode receber um request com folga) e também preparação para ajudar em problemas de classificação e recomendação de produtos (produtos mais procurados, clickados ou vendidos).

Quando falamos de implementar estes algoritmos é interessante observar quais otimizações foram feitas. Este artigo https://www.cs.rice.edu/~as143/Papers/topkapi.pdf foi interessante para entender a comparação entre maneiras de encontrar os Top-K itens em sistemas distribuidos.

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net"
    "net/http"
    "strings"
    "time"

    "github.com/axiomhq/topk"
)

/*
{
      "ip_prefix": "54.239.1.96/28",
      "region": "eu-north-1",
      "service": "AMAZON",
      "network_border_group": "eu-north-1"
    },

*/

// IPRange register
type IPRange struct {
    IPPrefix           string `json:"ip_prefix"`
    Region             string `json:"region"`
    Service            string `json:"service"`
    NetworkBorderGroup string `json:"network_border_group"`
}

// IPRanges struct
type IPRanges struct {
    SyncToken  string    `json:"syncToken"`
    CreateDate string    `json:"createDate"`
    Prefixes   []IPRange `json:"prefixes"`
}

func fetchAWSRanges() (*IPRanges, error) {
    // aws ip ranges
    // https://ip-ranges.amazonaws.com/ip-ranges.json

    var body []byte
    var err error

    httpClient := http.Client{
        Timeout: time.Second * 2, // Timeout after 2 seconds
    }

    req, err := http.NewRequest(http.MethodGet, "https://ip-
```

```
ranges.amazonaws.com/ip-ranges.json", nil)
    if err != nil {
        log.Fatal(err)
    }

    res, err := httpClient.Do(req)
    if err != nil {
        log.Fatal(err)
    }

    if res.Body != nil {
        defer res.Body.Close()
    }

    if body, err = ioutil.ReadAll(res.Body); err != nil {
        return nil, err
    }

    IPR := IPRanges{}
    if err = json.Unmarshal(body, &IPR); err != nil {
        log.Fatalf("unable to parse value: %q, error: %s", string(body),
err.Error())
        return nil, err
    }

    return &IPR, nil
}

func incrementIP(ip net.IP) {
    for j := len(ip) - 1; j >= 0; j-- {
        ip[j]++
        if ip[j] > 0 {
            break
        }
    }
}

func main() {
    var ipCount uint64

    ipr, err := fetchAWSRanges()
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Ranges: %d\n", len(ipr.Prefixes))

    tk := topk.New(100)

    ipCount = 0
    rangeCount := 0
    exactCount := make(map[string]int)
```

```
    for _, ipp := range ipr.Prefixes {
        rangeCount++
        //Uncomment to test a shorter run
        //if rangeCount > 100 {
        //    break
        //}

        ip, ipnet, err := net.ParseCIDR(ipp.IPPrefix)
        if err != nil {
            log.Println(err)
        } else {
            for ip := ip.Mask(ipnet.Mask); ipnet.Contains(ip); incrementIP(ip)
{
                // calculate most common first 2 octets from ip address (xx.yy)
                octetPrefix := strings.Split(ip.String(), ".")
                op := fmt.Sprintf("%s.%s", octetPrefix[0], octetPrefix[1])
                exactCount[op]++

                e := tk.Insert(op, 1)
                if e.Count < exactCount[op] {
                    fmt.Printf("Error: estimate lower than exact: key=%v,
exact=%v, estimate=%v\n", e.Key, exactCount[op], e.Count)
                }
                if e.Count-e.Error > exactCount[op] {
                    fmt.Printf("Error: error bounds too large: key=%v,
count=%v, error=%v, exact=%v\n", e.Key, e.Count, e.Error, exactCount[op])
                }
            }
        }

    }
    fmt.Printf("Unique IP Addresses: %d\n", ipCount)
    el := tk.Estimate("52.94")
    fmt.Printf("Prefix 52.94 ranks at %d\n ", el.Count)

    kk := tk.Keys()
    fmt.Println("List top 10 matches\nFirst 2 IP octets - Count")
    for i := 0; i < 10; i++ {
        fmt.Printf("%s - %d\n", kk[i].Key, kk[i].Count)
    }

}
```

O programa acima usa a mesma fonte de endereços IPs da AWS, mas comparando apenas os dois primeiros octetos do IP para ver quais grupos aparecem mais entre os ranges e aonde um par de octetos especifico está no conjunto total de dados.

```
$ go run topk.go
Ranges: 2946
Unique IP Addresses: 0
Prefix 52.94 ranks at 3088
 List top 10 matches
```

```
First 2 IP octets - Count

13.236 - 65536
13.237 - 65536
13.238 - 65536
13.239 - 65536
15.185 - 65536
18.200 - 65536
18.232 - 65536
18.233 - 65536
18.234 - 65536
18.235 - 65536
```

Conclusão

Como escrevi no inicio, este livro é um trabalho em evolução. Até aqui eu usei um projeto meu e algoritmos que aprendi para problemas que eu vi. Eu quis colocar estas informações em ordem para que qualquer pessoa interessada pudesse testar rapidamente e tirar suas conclusões.

Se você ver um erro, algoritmo que pode ser interessante ou tiver alguma idéia, por favor compartilhe comigo.

Gleicon