Aula: Algoritmo HyperLogLog — Estimativa de Cardinalidade em Grandes Conjuntos

1. Introdução e História

O problema de estimar o número de elementos distintos (cardinalidade) em um conjunto é fundamental em ciência da computação, especialmente em áreas como bancos de dados, mineração de dados, análise de redes e sistemas distribuídos.

Antes do surgimento de técnicas probabilísticas, para saber o número exato de elementos distintos em um conjunto, a solução tradicional envolvia armazenar todos os elementos ou seus hashes em uma estrutura (como uma tabela hash) para contagem exata. Isso, porém, exige muita memória, especialmente em aplicações que lidam com grandes volumes de dados (big data).

Na década de 1980, começaram a surgir algoritmos probabilísticos para estimar cardinalidades com uso de memória reduzido, como o **LogLog**, desenvolvido por Philippe Flajolet e seus colegas. Em 2007, Flajolet et al. publicaram o artigo seminal onde introduziram o **HyperLogLog** — uma melhoria significativa no algoritmo LogLog original — que permitia estimar cardinalidades muito grandes usando pouquíssima memória, com alta precisão.

Referência chave:
Flajolet, P., Fusy, É., Gandouet, O., & Meunier, F. (2007). *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*. In *Proceedings of the 2007 International Conference on Analysis of Algorithms*.

---

2. Por que o HyperLogLog surgiu? Motivação

Com o crescimento exponencial dos dados, as soluções que armazenavam todos os elementos para contar distintamente tornaram-se inviáveis. Por exemplo, redes sociais, sistemas de recomendação e análise de logs geram milhões a bilhões de eventos únicos.

O desafio era: **Como estimar o número de elementos distintos em um fluxo de dados, com baixo uso de memória e alta precisão?**

Antes do HyperLogLog, o algoritmo **LogLog** e o **Probabilistic Counting with Stochastic Averaging** (PCSA) de Flajolet & Martin (1985) já existiam, mas tinham limitações quanto a precisão e eficiência de memória. O HyperLogLog melhorou drasticamente a precisão para uma dada quantidade de memória.

---

3. O que era feito antes?

- **Contagem exata:** armazenar todos os elementos em um hash set. Exato, porém linear em memória. Impraticável para grandes volumes.
- **Algoritmos probabilísticos anteriores:**

- **PCSA (Probabilistic Counting with Stochastic Averaging)** (Flajolet & Martin, 1985): usa posições de bits no hash para estimar cardinalidade, mas com precisão limitada.
- **LogLog (Flajolet et al., 2003):** melhora a PCSA ao usar técnicas de agrupamento e médias harmônicas, porém ainda com alta variância.
- **Linear Counting:** outra técnica probabilística eficaz para cardinalidades pequenas a médias, mas perde precisão em cardinalidades muito altas.

---

4. Como o HyperLogLog funciona?

Conceitos básicos

O HyperLogLog usa funções de hash e propriedades estatísticas do posicionamento do primeiro bit “1” em uma sequência binária para estimar cardinalidade.

Passos simplificados:

1. **Hashing:** Cada elemento é processado por uma função hash uniforme e de saída longa (ex: 64 bits).
2. **Divisão em registros:** A saída do hash é dividida em duas partes:
    - Os primeiros *p* bits identificam um registro (ou “bucket”) entre 2^p registros.
    - Os bits restantes são usados para calcular o “rank” — a posição do primeiro bit ‘1’ — que indica a raridade do hash.
3. **Registro máximo:** Para cada bucket, o algoritmo mantém o maior rank já visto.
4. **Estimativa:** A cardinalidade é estimada combinando os valores dos registros via média harmônica ajustada por um fator de correção.

Por que isso funciona?

A posição do primeiro bit ‘1’ em uma sequência hash uniforme é geometricamente distribuída. Quanto mais elementos distintos, maior a chance de encontrar hashes com posições altas de bits ‘1’, o que indica maior cardinalidade.

Fórmula geral

Seja $m = 2^p$ o número de registros, $M[i]$ o maior rank no registro i, a estimativa da cardinalidade E é dada por:

$$
E = \alpha_m \cdot m^2 \cdot \left(\sum_{i=1}^m 2^{-M[i]} \right)^{-1}
$$

onde α_m é uma constante de correção específica para m.

---

5. Vantagens do HyperLogLog

- **Baixa memória:** utiliza algumas kilobytes para estimar cardinalidades de bilhões de elementos.
- **Alta precisão:** erro padrão de aproximadamente $1.04/\sqrt{m}$, que pode ser ajustado aumentando m.
- **Velocidade:** operações de inserção são rápidas, baseadas em hash e atualização simples.
- **Simplicidade:** estrutura compacta, fácil de serializar e usar em sistemas distribuídos.
- **Escalabilidade:** combina facilmente estimativas de diferentes HyperLogLogs para somar cardinalidades.

---

6. Desvantagens e limitações

- **Estimativa, não exatidão:** o resultado é uma aproximação, com margem de erro estatística.
- **Falsos positivos em cardinalidades muito baixas:** para conjuntos pequenos, o algoritmo pode superestimar. Técnicas híbridas combinam Linear Counting para essa faixa.
- **Dependência da função hash:** qualidade do hash impacta diretamente a precisão.
- **Complexidade na implementação correta:** cuidado com ajuste de parâmetros, tratamento de viés e combinação de contadores.

---

7. Conclusão

O HyperLogLog representa um avanço notável na área de algoritmos probabilísticos para estimativa de cardinalidade, combinando eficiência, precisão e baixo custo computacional. É amplamente usado em sistemas modernos de análise de dados e bancos NoSQL (ex: Redis, Cassandra) para resolver problemas antes impraticáveis em escala.

Como resumem Flajolet et al. (2007):
"HyperLogLog achieves cardinality estimation with near-optimal accuracy using minimal memory, making it indispensable in big data analytics."

8. Detalhes Técnicos e Ajustes Práticos do HyperLogLog

8.1 O papel do parâmetro p

O parâmetro p define o número de registros $m = 2^p$ usados na estrutura. Quanto maior p, mais registros, o que significa:

- **Menor erro:** O erro padrão da estimativa é aproximadamente $1.04/\sqrt{m}$. Por exemplo, para $p=14$ (ou $m=16384$), o erro esperado é cerca de 0.81%.
- **Maior memória:** Cada registro armazena um valor de rank (normalmente em poucos bits), portanto a memória cresce linearmente com m.

Na prática, valores típicos de p variam entre 10 e 16, equilibrando precisão e uso de memória.

8.2 Correções para cardinalidades muito baixas ou muito altas

O algoritmo HyperLogLog clássico apresenta algumas imprecisões nas extremidades da faixa de cardinalidade:

- Para cardinalidades muito **baixas**, ele tende a superestimar. Para corrigir isso, a literatura sugere o uso do **Linear Counting**:
  Quando a estimativa E é menor que $2.5 \cdot m$, utiliza-se uma fórmula baseada no número de buckets vazios para ajustar a estimativa.

- Para cardinalidades muito **altas** (próximas ao limite do espaço de hash), existe uma correção para evitar saturação e viés.

Essas correções foram detalhadas por Flajolet et al. (2007) e são essenciais para garantir a robustez do algoritmo em aplicações reais.

8.3 Combinação de HyperLogLogs

Um recurso fundamental é que HyperLogLogs podem ser combinados via operação bit-a-bit entre seus registros, permitindo:

- Estimar a cardinalidade da união de conjuntos (ex: usuários únicos em múltiplas janelas temporais).
- Ser usados em sistemas distribuídos, onde cada nó mantém um HyperLogLog e o resultado final é obtido agregando-os.

Isso permite processamento em larga escala e sistemas altamente distribuídos como Hadoop, Spark, Redis, entre outros.

---

9. Variantes e Extensões do HyperLogLog

Além do HyperLogLog básico, várias extensões foram propostas:

- **HyperLogLog++** (Google, 2013): adiciona técnicas para reduzir ainda mais o viés e melhora o desempenho para conjuntos pequenos, combinando Linear Counting, correção de viés empírica e compactação eficiente.
  Referência: Heule, S., Nunkesser, M., & Hall, A. (2013). *HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm*. Proceedings of the 16th International Conference on Extending Database Technology (EDBT).

- **Sparse HyperLogLog:** para conjuntos pequenos, onde a estrutura é armazenada de forma esparsa, reduzindo memória e acelerando operações.

Essas variantes são usadas em ferramentas modernas de análise de dados, garantindo alta performance e precisão mesmo em cenários variados.

---

10. Aplicações Reais do HyperLogLog

O HyperLogLog é amplamente adotado no mercado e em sistemas acadêmicos. Algumas aplicações típicas:

- **Análise de logs e telemetria:** Contar usuários únicos, visitas a páginas, eventos distintos em sistemas web, monitoramento de redes.

- **Bancos de dados NoSQL:** Redis (desde a versão 2.8) tem suporte nativo a HyperLogLog via comando `PFCOUNT` e `PFADD`, oferecendo cardinalidade estimada com poucos kilobytes de memória.
- **Sistemas de recomendação:** para identificar a diversidade de itens, usuários únicos ou sessões distintas.
- **Redes sociais:** estimar seguidores únicos, alcance de publicações, interações distintas.
- **Big data e computação distribuída:** frameworks como Apache Spark, Flink e Hadoop implementam algoritmos baseados em HyperLogLog para análise rápida de grandes volumes de dados.

---

11. Considerações para Implementação

Ao implementar ou utilizar o HyperLogLog, é importante considerar:

- **Qualidade da função hash:** Deve ser uniforme e ter saída suficientemente longa para evitar colisões que degradam a estimativa.
- **Serialização eficiente:** Para sistemas distribuídos, HyperLogLogs devem ser facilmente serializados, enviados pela rede e combinados sem perda.
- **Atualização incremental:** Inserir novos elementos é uma operação simples e rápida, basta atualizar o registro do bucket correspondente se o rank do novo hash for maior.
- **Parâmetros ajustados ao domínio:** Dependendo da faixa esperada de cardinalidade, escolha p apropriado para balancear erro e memória.

---

12. Comparação com Outras Técnicas Probabilísticas

Técnica	Uso de Memória	Precisão	Aplicabilidade	Notas
Contagem Exata	Alto (linear em n)	Exata	Pequenos conjuntos	Inviável para big data
PCSA (Flajolet-Martin)	Baixo	Baixa	Conjuntos grandes	Antecessor do HyperLogLog
LogLog	Baixo	Moderada	Grandes conjuntos	Substituído pelo HyperLogLog
Linear Counting	Baixo	Alta para pequenos conjuntos	Pequenos a médios conjuntos	Ineficaz para muito grandes conjuntos
HyperLogLog	Muito baixo	Alta	Grandes conjuntos	Estado da arte para estimativa

---

13. Considerações Finais

O algoritmo HyperLogLog é um exemplo clássico de como técnicas probabilísticas permitem resolver problemas impossíveis ou impraticáveis com abordagens determinísticas puras em cenários de big data. Sua simplicidade, baixo custo de memória e alta precisão o tornam indispensável na caixa de ferramentas do cientista da computação moderno.

Como ressaltam Flajolet e colaboradores, a criatividade na combinação de estatística, teoria dos algoritmos e engenharia é chave para enfrentar os desafios dos dados massivos que encontramos hoje.

---

14. Aplicações Práticas do HyperLogLog no Dia a Dia

O HyperLogLog é amplamente usado em diversos sistemas que precisam estimar grandes volumes de dados distintos de forma rápida e eficiente. A seguir, alguns exemplos reais e cotidianos de uso:

14.1 Análise de Visitantes Únicos em Sites e Apps

- **Problema:** Uma empresa que opera um site com milhões de visitas diárias quer saber quantos visitantes únicos acessam a plataforma, sem guardar o histórico de todos os IPs ou usuários (o que consumiria muita memória).
- **Como o HyperLogLog ajuda:**
  Em vez de armazenar todos os IPs, cada IP é processado por um hash e inserido no HyperLogLog. O algoritmo estima o número total de visitantes únicos com alta precisão usando poucos kilobytes de memória.
- **Exemplo:** Grandes portais de notícias, redes sociais e plataformas de streaming usam HyperLogLog para monitorar diariamente o alcance e a audiência.

14.2 Contagem de Usuários Ativos em Redes Sociais

- **Problema:** Uma rede social quer monitorar quantos usuários únicos interagiram com uma postagem, curtiram um comentário ou assistiram a um vídeo ao longo de uma campanha.
- **Como o HyperLogLog ajuda:**
  Em vez de armazenar listas gigantescas, o sistema adiciona o ID do usuário no HyperLogLog para estimar rapidamente quantos usuários distintos participaram da interação.
- **Exemplo:** Facebook, Twitter e Instagram podem usar técnicas semelhantes para medir alcance e engajamento sem impactar performance.

14.3 Sistemas de Recomendação e Publicidade Online

- **Problema:** Plataformas de recomendação querem saber quantos usuários distintos viram um anúncio ou uma recomendação, para avaliar a eficácia de campanhas.
- **Como o HyperLogLog ajuda:**
  A estimativa rápida de usuários únicos permite ajustar em tempo real a distribuição de anúncios e evitar a repetição excessiva.

- **Exemplo:** Google Ads e plataformas DSP (Demand Side Platform) usam essa técnica para otimizar a alocação de anúncios.

14.4 Infraestrutura de Banco de Dados e Cache Distribuído

- **Problema:** Serviços como Redis, Cassandra e outras bases NoSQL precisam monitorar cardinalidades de elementos em grandes coleções sem consumir muita memória.
- **Como o HyperLogLog ajuda:**
  Redis implementa comandos como `PFADD` e `PFCOUNT`, que usam HyperLogLog para estimar o número de elementos únicos em uma chave, útil para métricas internas, controle de uso, etc.
- **Exemplo:** Um sistema de caching pode usar isso para saber quantos usuários únicos acessaram um recurso armazenado em cache.

14.5 Monitoramento de Redes e Segurança

- **Problema:** Analistas de segurança querem saber quantos endereços IP únicos tentaram acessar uma rede ou serviço para identificar padrões de ataques.
- **Como o HyperLogLog ajuda:**
  Em tempo real, os sistemas inserem os IPs em um HyperLogLog para manter estatísticas sobre a diversidade de acessos, ajudando a detectar anomalias sem armazenar logs detalhados.
- **Exemplo:** Firewalls e sistemas de detecção de intrusão (IDS) usam essa técnica para análise rápida e escalável.

14.6 Análise de Dados em Serviços de Streaming

- **Problema:** Serviços de streaming querem saber quantos usuários únicos assistiram a um conteúdo específico ou quantos títulos distintos foram assistidos em um período.
- **Como o HyperLogLog ajuda:**
  Com o HyperLogLog, é possível calcular essas métricas em tempo real, mesmo com milhões de acessos simultâneos.
- **Exemplo:** Netflix e Spotify podem usar essa estimativa para relatórios de audiência e personalização.

14.7 Sistemas de Log e Telemetria em Nuvem

- **Problema:** Plataformas de nuvem (AWS, Azure, GCP) precisam coletar e processar logs de centenas de milhares de máquinas e serviços, estimando cardinalidades de eventos únicos para monitoramento e billing.
- **Como o HyperLogLog ajuda:**
  Reduzindo drasticamente a memória necessária para essas métricas, possibilita o monitoramento

eficiente de grande escala.

Resumo das Aplicações Cotidianas

Setor/Área	Aplicação Exemplo	Benefício do HyperLogLog
Web e Mobile	Visitantes únicos em sites e apps	Memória baixa, alta escalabilidade
Redes Sociais	Usuários que interagem com posts	Rápida estimativa de alcance
Publicidade Digital	Contagem de usuários que viram anúncios	Otimização e ajuste em tempo real
Banco de Dados	Cardinalidade em coleções grandes	Eficiência na memória e velocidade
Segurança	Contagem de IPs únicos em ataques	Detecção rápida e escalável
Streaming	Usuários únicos por conteúdo	Métricas de audiência em tempo real
Nuvem e Telemetria	Eventos únicos em logs distribuídos	Escalabilidade e economia de recursos

Projeto FastAPI + Redis HyperLogLog + Docker

1. Estrutura básica do projeto

```
hyperloglog-project/
│
├── app/
│   ├── main.py
│   └── requirements.txt
│
├── Dockerfile
├── docker-compose.yml
└── README.md
```

---

2. Código FastAPI (`app/main.py`)

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import redis.asyncio as redis

app = FastAPI()
r = redis.Redis(host='redis', port=6379, db=0)
```

```
HLL_KEY = "hyperloglog:hll"

class Item(BaseModel):
    element: str

@app.post("/add/")
async def add_element(item: Item):
    added = await r.pfadd(HLL_KEY, item.element)
    return {"added": bool(added)}

@app.get("/count/")
async def get_count():
    count = await r.pfcount(HLL_KEY)
    return {"estimated_unique_count": count}

@app.post("/reset/")
async def reset():
    await r.delete(HLL_KEY)
    return {"status": "reset done"}
```

---

3. Dépendências (`app/requirements.txt`)

```
fastapi
uvicorn[standard]
redis[async]
```

---

4. Dockerfile para FastAPI

```
FROM python:3.11-slim

WORKDIR /app

COPY app/requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY app/ .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

---

5. docker-compose.yml (Redis + FastAPI)

```
version: '3.8'

services:
  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  api:
    build: .
    depends_on:
      - redis
    ports:
      - "8000:8000"
```

---

6. Como usar

- Para rodar tudo junto (no mesmo diretório do docker-compose.yml):

```
docker-compose up --build
```

- O FastAPI estará disponível em:
  `http://localhost:8000`

---

7. Testando a API

- Adicionar elemento:

```
curl -X POST "http://localhost:8000/add/" -H "Content-Type:
application/json" -d '{"element":"usuario123"}'
```

- Obter contagem estimada:

```
curl "http://localhost:8000/count/"
```

- Resetar HyperLogLog:

```
curl -X POST "http://localhost:8000/reset/"
```

---

8. Explicações

- O Redis gerencia internamente o HyperLogLog, você não precisa se preocupar com detalhes de implementação.
- A API é simples, você pode ampliar para suportar conjuntos diferentes, métricas mais complexas, autenticação, etc.
- O uso do `redis.asyncio` é para aproveitar a natureza assíncrona do FastAPI e melhorar desempenho.

Referências

- Flajolet, P., Fusy, É., Gandouet, O., & Meunier, F. (2007). *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*. Proceedings of the 2007 International Conference on Analysis of Algorithms.
- Flajolet, P., & Martin, G. N. (1985). *Probabilistic Counting Algorithms for Data Base Applications*. Journal of Computer and System Sciences.
- Flajolet, P., et al. (2003). *LogLog counting of large cardinalities*. Proceedings of the 2003 European Symposium on Algorithms.