UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ALEXANDRE TOLOMEOTTI ENOKIDA

ESTUDO EXPERIMENTAL DA ESTIMATIVA DE CARDINALIDADE PARA AUTOJUNÇÕES

CAMPO MOURÃO

2025

ALEXANDRE TOLOMEOTTI ENOKIDA

ESTUDO EXPERIMENTAL DA ESTIMATIVA DE CARDINALIDADE PARA AUTOJUNÇÕES

Cardinality Estimation for Self-Joins: An Experimental Study

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.
Orientador(a): Prof. Dr. Eduardo Henrique Monteiro Pena

CAMPO MOURÃO

2025

4.0 Internacional

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

ALEXANDRE TOLOMEOTTI ENOKIDA

ESTUDO EXPERIMENTAL DA ESTIMATIVA DE CARDINALIDADE PARA AUTOJUNÇÕES

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 04/dezembro/2025

Prof. Dr. Eduardo Henrique Monteiro Pena
Doutorado
Universidade Tecnológica Federal do Paraná

Prof. Dr. André Luis Schwerz
Doutorado
Universidade Tecnológica Federal do Paraná

Prof. Dr. Marco Aurélio Graciotto Silva
Doutorado
Universidade Tecnológica Federal do Paraná

CAMPO MOURÃO

2025

RESUMO

O otimizador de consultas é um componente crítico em Sistemas de Gerenciamento de Banco de Dados (SGBDs), pois busca identificar planos de execução eficientes. SGBDs comerciais empregam modelos de custo que dependem da estimativa de seletividade de predicados e da cardinalidade dos operadores, utilizando estatísticas do catálogo e estruturas de dados como sketches. Contudo, as suposições sobre essas estimativas — como o princípio da inclusão em junções ou a independência entre atributos em conjunções de predicados — frequentemente introduzem imprecisões. Essas imprecisões podem levar à escolha de planos subótimos, com impacto severo no desempenho, especialmente em consultas mais complexas. A literatura recente tem se concentrado predominantemente na estimativa para consultas do tipo SPJ (Select-Project-Join) e de agrupamento, enquanto a eficácia das estimativas para consultas de autojunção (*self-join*) permanece pouco explorada. Esse tipo de consulta é amplamente utilizado em tarefas de limpeza de dados, como a validação de restrições de negação, mas frequentemente apresenta desempenho imprevisível e alto consumo de memória — muitas vezes em decorrência da adoção de algoritmos de junção ineficientes. Neste contexto, a proposta deste trabalho é investigar experimentalmente as técnicas de estimativa de cardinalidade para consultas de *self-join*. Para tal, as principais contribuições que o trabalho busca são: (i) uma revisão das principais técnicas de estimativa de cardinalidade para *self-joins*; (ii) a implementação de um gerador de dados sintéticos, especificamente projetado para *benchmarking* de técnicas de estimativa aplicadas a *self-joins*; e (iii) a condução de um experimento comparativo abrangente, avaliando o desempenho de diferentes estratégias de estimativa de cardinalidade em consultas de *self-join*. Espera-se que esta pesquisa forneça resultados sobre os limites e a eficácia das abordagens atuais para a estimativa de cardinalidade para consultas de *self-join*.

Palavras-chave: processamento de consulta; otimização de consulta; estimativa de cardinalidade; sketch.

ABSTRACT

Query optimization is a critical task employed by Database Management Systems (DBMSs) to search for efficient execution plans. Commercial DBMSs employ cost models that rely on estimating the selectivity of predicates and the cardinality of operators, using catalog statistics and data structures such as sketches. However, the assumptions underlying these estimations — such as the inclusion principle in joins or the independence of attributes — often introduce inaccuracies. These inaccuracies leads to the selection of suboptimal plans, severely impacting performance, especially in more complex queries. Recent literature has predominantly focused on estimation for SPJ (Select-Project-Join) and grouping queries, while the effectiveness of estimations for self-join queries remains under-explored. This type of query is widely used in data cleaning tasks, such as denial constraints, but often exhibits unpredictable performance and high memory consumption — frequently due to the adoption of inefficient join algorithms. In this context, the proposal of this work is to experimentally investigate cardinality estimation techniques for self-join queries. To this end, the main contributions the work seeks are: (i) a review of the main cardinality estimation techniques for self-joins; (ii) the implementation of a synthetic data generator, designed for benchmarking estimation techniques applied to self-joins; and (iii) conducting a comprehensive comparative experiment, evaluating the performance of different cardinality estimation strategies for self-join queries. This research is expected to provide results regarding the limitations and effectiveness of current approaches for self-join query cardinality estimation.

Keywords: query execution; query optimization; cardinality estimation; sketch.

LISTA DE ALGORITMOS

Algoritmo 1 – Verificação de DC com um predicado da forma $t.A = t^{'}.A$ 14
Algoritmo 2 – Tug-of-War Sketch 31

LISTA DE FIGURAS

Figura 1 – Exemplo de *equi-width histogram* 18
Figura 2 – Exemplo de *equi-depth histogram* 18
Figura 3 – Exemplo de *end-biased histogram* 18
Figura 4 – Diagrama ilustrativo do funcionamento do HyperLogLog com M = 4 *buckets* 20
Figura 5 – Estrutura generalizada de um *Count-Min Sketch* 21
Figura 6 – Os diferentes impactos da cardinalidade das colunas no tempo de execução dos algoritmos hsm, bhsm e iejoin 27
Figura 7 – Esquema do conjunto de dados IMDb 29
Figura 8 – Estimativas dos algoritmos sobre as consultas $\varphi_1 - \varphi_{12}$ 38
Figura 9 – Tempo de construção das estruturas sobre as consultas $\varphi_1 - \varphi_{12}$ 39
Figura 10 – Tempo de estimativa dos algoritmos sobre as consultas $\varphi_1 - \varphi_{12}$ 40

LISTA DE TABELAS

Tabela 1 – Relação `funcionario`. 13
Tabela 2 – Configurações disponíveis no gerador de dados 30
Tabela 3 – Parâmetros da distribuição dos atributos `age` e `salary` 35
Tabela 4 – Predicados utilizados nos experimentos 36
Tabela 5 – Média do *q-error* sobre os tamanhos: 10K, 100K, 500K, 1M e 2M 42

LISTA DE QUADROS

Quadro 1 – Relação R_1 .... **22**

LISTAGEM DE CÓDIGOS FONTE

Listagem 1 – Consulta que retorna pares de funcionários que violam a regra φ_1 13

SUMÁRIO

1	**INTRODUÇÃO**	**11**
2	**FUNDAMENTAÇÃO TEÓRICA**	**16**
2.1	**Processamento e Otimização de Consultas**	**16**
2.1.1	Estatísticas	16
2.1.1.1	Histogramas	17
2.1.1.2	Amostragem	18
2.1.1.3	*Sketches*	19
2.1.2	Técnicas para Estimativa	22
2.1.2.1	Estimativa de Seleção (σ)	22
2.1.2.2	Estimativa do Número de Valores Distintos	24
2.1.2.3	Estimativa do Tamanho de *Join*	25
2.1.2.4	Estimativa do Tamanho de *self-join*	26
3	**DESENVOLVIMENTO**	**28**
3.1	**Materiais e métodos**	**28**
3.1.1	Conjuntos de Dados	28
3.1.1.1	*Benchmark*	28
3.1.1.2	Dados reais	28
3.1.1.3	Dados sintéticos	29
3.1.2	Ferramentas	30
3.1.2.1	Algoritmos de Estimativa	30
3.1.3	Parâmetros Avaliados	31
3.1.4	Implementação	32
4	**EXPERIMENTOS**	**34**
4.1	**Conjuntos de dados**	**34**
4.2	**Predicados**	**35**
4.3	**Algoritmos**	**36**
4.4	**Limitações**	**42**
5	**CONCLUSÕES**	**43**
	REFERÊNCIAS	**44**

1 INTRODUÇÃO

Otimização de consulta é uma tarefa realizada por Sistemas de Gerenciamento de Banco de Dados (SGBDs) durante o processamento de uma consulta, cujo objetivo é encontrar o plano de execução ótimo (ou, ao menos, próximo do ótimo) para a consulta em questão. Encontrar um bom plano é crucial para a execução eficiente de uma consulta, tendo em vista que planos subótimos reduzem o desempenho (ou seja, tempo de resposta) em algumas ordens de magnitude (Leis *et al.*, 2018).

Tradicionalmente, duas estratégias de alto nível para otimização de consultas são utilizadas: (i) heurísticas; e (ii) custo. A primeira abordagem consiste em utilizar heurísticas para reescrever consultas utilizando regras de equivalência da álgebra relacional. Dizemos que duas expressões da álgebra relacional são equivalentes se elas geram o mesmo resultado — isto é, produzem a mesma relação de saída. A estratégia de heurísticas é interessante pois permite a otimização de consultas sem ler os dados das relações. A segunda estratégia consiste em utilizar um modelo de custo para estimar o custo (tempo) de diferentes planos de execução de uma consulta, escolhendo ao final aquele que apresenta o menor custo estimado — diferentemente das heurísticas, a estratégia de custo necessita ler dados.

O *System R* foi o primeiro SGBD a utilizar do modelo de custo (Leis *et al.*, 2015), introduzido por Selinger *et al.* (1979). Em alto nível, uma consulta é tipicamente decomposta em blocos de consulta lógica e existem inúmeras sequências de execução desses blocos. Uma das ideias-chave introduzidas por Selinger *et al.* (1979) foi a utilização de um modelo de custo que utiliza estatísticas das relações, armazenadas no catálogo do sistema, para estimar o custo de cada sequência.

O modelo de custo utiliza estimativas de seletividade dos predicados da consulta e estimativas da cardinalidade (isto é, tamanho) da saída dos operadores relacionais como entrada para gerar planos de execução semanticamente equivalentes, selecionando ao final aquele com menor custo. A estimativa de cardinalidade é obtida por meio da seletividade do predicado — a fração de tuplas da relação que qualificam o predicado. Para isso, estatísticas dos atributos das tabelas (por exemplo, cardinalidade e frequência de valor distinto) são armazenadas separadamente.

A literatura apresenta uma variedade de trabalhos que propõem estruturas de dados aproximadas (não exatas) capazes de manter essas estatísticas de forma escalável. Por exemplo, Flajolet *et al.* (2007) apresentam uma estrutura compacta (isto é, utiliza pouco espaço) para estimar o número de valores distintos de um conjunto de dados de grande volume. Cormode e Muthukrishnan (2005) introduzem uma estrutura compacta para estimar a frequência de valores distintos (contagem).

Além do uso de estruturas de dados para armazenar estatísticas, SGBDs frequentemente fazem suposições para estimar a cardinalidade em consultas complexas, como junções (*joins*) e conjunções de predicados. Essas suposições são fundamentais para decompor e otimizar consultas ainda mais complexas.

No caso de *joins*, uma abordagem comum é particionar os dados em grupos com base nos valores dos atributos envolvidos e alinhar esses grupos conforme suas faixas de valores. Para tornar isso viável, assume-se o princípio da inclusão (Selinger *et al.*, 1979), ou seja, que para cada tupla do lado com menos grupos, existe um grupo correspondente no outro lado com pelo menos uma tupla compatível, garantindo uma correspondência entre as tuplas dos dois lados.

Quanto à conjunção de predicados, ou seja, seleções (σ), a estimativa é baseada na suposição da independência: assume-se que os atributos dos predicados são independentes. Dados reais geralmente não satisfazem essa propriedade. Por exemplo: em uma relação `veiculo(fabricante, modelo,...)`, uma conjunção do tipo `fabricante = 'Volkswagen' AND modelo = 'Gol'` assumiria independência entre os atributos — o que é irrealista, pois na prática todo carro “Gol” é fabricado pela Volkswagen. Observe que para cada seleção é necessário estimar a quantidade de tuplas que qualificam aquele valor. Para lidar com esse problema, SGBDs recorrem à suposição da uniformidade: os valores de um atributo são uniformemente distribuídos. Novamente, essa suposição é impraticável sobre dados reais.

Uma solução para o problema de obter a frequência de valores distintos são *sketches* — estruturas de dados sinopse que representam estatísticas sobre a distribuição dos dados de forma compacta e aproximada, construídas com uma única varredura sobre os dados (Cormode *et al.*, 2012). *Sketches* são uma abordagem relativamente mais recente do que as ferramentas estatísticas tradicionais (por exemplo, histogramas e amostragem). Técnicas de *sketch* têm se mostrado cada vez mais adequadas para SGBDs, visto que são capazes de realizar atualizações incrementais sobre estatísticas de um conjunto de dados — isto é, suportam conjuntos de dados dinâmicos. Além disso, *sketches* comumente utilizam o conceito de *k-hashing independence* (Freitag; Neumann, 2019), no qual funções *hash* com probabilidade uniforme de serem escolhidas são selecionadas de forma que seus resultados sejam variáveis aleatórias independentes e identicamente distribuídas. Dada essa natureza, os SGBDs ainda recorrem à suposição de independência, estimando a seletividade de predicados conjuntos como o produto das seletividades individuais. Ambas as suposições representam simplificações que, embora práticas, ainda são fontes significativas de imprecisão na estimativa de cardinalidade.

Além disso, observamos uma tendência que grande parte dos estudos recentes têm se concentrado em consultas do tipo SPJ (select-project-join) (e.g., (Leis *et al.*, 2018)) e em consultas de agrupamento (e.g., (Moerkotte, 2024)). Como exemplo, em consultas SPJ, as etapas do procedimento são: filtrar, juntar tabelas com base em chaves primárias e chaves estrangeiras, e projetar, tendo como principal desafio a ordem de execução dos *joins*. Apesar da relevância dessas consultas, autojunções (denominadas *self-joins* ao longo deste trabalho) permanecem pouco exploradas, criando uma lacuna na compreensão dos limites dos algoritmos do estado da arte aplicados a esse tipo de consulta.

Consultas de *self-join* têm ampla aplicação prática, especialmente em tarefas de *data cleaning* (Pena; Almeida; Naumann, 2021). Um exemplo atual e relevante é o uso de *Denial*

Tabela 1 – Relação `funcionario`.

	ID	Nome	Dept	DataInicio	Salario
t_1	101	V. Costa	Comercial	2012	3000
t_2	102	P. Silva	Comercial	2014	8000
t_3	103	V. Gonçalves	Comercial	2014	6000
t_4	104	M. Oliveira	Comercial	2014	8000

Fonte: Autoria própria (2025).

Constraints (DCs), que modelam regras de qualidade de dados. Cada DC define a negação de uma condição que, quando satisfeita por um par de tuplas em uma base de dados, indica uma violação da integridade semântica dos dados, revelando erros ou inconsistências. SGBDs já implementam restrições básicas, como `UNIQUE` e `NOT NULL`. No entanto, o modelo relacional tradicional não abrange todas as formas possíveis de restrições semânticas. Por exemplo, considere a Tabela 1 com registros fictícios de funcionários de uma empresa. Uma regra implícita (φ_1) pode ser: “entre dois funcionários do mesmo departamento, o funcionário com data de admissão anterior deve possuir salário superior ao daquele com data de admissão posterior”.

Esse tipo de regra pode ser expressa por uma DC, representada por predicados sobre atributos da relação e implementada com uma consulta SQL com *self-joins* para identificar possíveis violações. O exemplo de SQL a seguir ilustra essa estrutura: Teoricamente, apesar de

Listagem 1 – Consulta que retorna pares de funcionários que violam a regra φ_1

```
1 SELECT t.ID, u.ID
2 FROM   funcionario t, funcionario u
3 WHERE  t.Dept = u.Dept
4 AND    t.DataInicio < u.DataInicio
5 AND    t.Salario < u.Salario;
```

Fonte: Autoria própria (2025).

sua implementação ser simples e compatível com mecanismos de otimização dos SGBDs, essa abordagem apresenta desempenho imprevisível e, em alguns casos, falha devido ao consumo excessivo de memória. Isso ocorre porque muitos SGBDs recorrem a algoritmos ineficientes de junção — como o *nested loop join* — ao processar filtros com intervalos. Para esse problema, diversos trabalhos propõem diferentes algoritmos para detecção de violações de restrições com predicados de intervalo, como Pena, Almeida e Naumann (2021) e Liu *et al.* (2024).

Um dos principais custos nos algoritmos do estado-da-arte para a verificação de DCs é, em geral, correlacionada ao número de valores distintos do atributo que participa do predicado. Por exemplo, considere um par de tuplas $t \neq t'$ pertencentes a uma relação R com um atributo A. Uma ideia geral de um algoritmo para identificar pares de tuplas que satisfaçam um predicado da forma $t.A = t'.A$ é dado pelo Algoritmo 1.

Observe que cada entrada da tabela consiste em um conjunto de tuplas, cujo produto cartesiano com ele próprio representa todos os pares de tuplas que satisfazem a DC para um valor distinto v. Esses conjuntos podem ser utilizados como entrada para algoritmos que imple-

Algoritmo 1 – Verificação de DC com um predicado da forma $t.A = t'.A$

requer Relação R, atributo A da DC
inserir Tabela hash H com tuplas agrupadas por $\pi_A(t)$
1: $H \leftarrow \emptyset$
2: **para** cada $t \in R$ **faça**
3: $v \leftarrow \pi_A(t)$
4: **se** $v \notin H$ **então**
5: $H[v] \leftarrow \emptyset$
6: **finaliza se**
7: $H[v] \leftarrow H[v] \cup \{t\}$
8: **finaliza para**
9: **retorna** H

Fonte: Adaptado de Liu *et al.* (2024).

mentam a validação de predicados de diferentes classes conjuntos na mesma DC — processo conhecido como *refinement*. Ferramentas de validação (Pena; Almeida; Naumann, 2021) exploram esse fato para reduzir o resultado intermediário no processamento de DCs, dado que tuplas pertencentes a diferentes conjuntos não podem violar a restrição.

Alon *et al.* (1999), em seu trabalho seminal, apresentaram um algoritmo (denominado *Tug-of-War*) para estimar o tamanho de um *join* entre duas tabelas combinando os tamanhos de seus respectivos *self-joins*. Ele estimou cada *self-join* explorando a assimetria dos dados, medida por meio do "k-ésimo momento da frequência" em fluxos de dados — especificamente para $k = 2$, momento que representa a assimetria dos dados. Dado que o artigo foi publicado há mais de duas décadas, levantamos a seguinte questão: não seria o momento de revisitar esse problema?

Para contextualizar nossa proposta, destacamos algumas contribuições: (Leis *et al.*, 2018) e (Leis *et al.*, 2015) concentram-se em consultas SPJ; (Moerkotte, 2024) aborda consultas de agregação estendendo o modelo proposto por (Selinger *et al.*, 1979); (Acharya *et al.*, 1999) explora o uso de amostras pré-computadas para estimar consultas de *join*; (Ganguly; Garofalakis; Rastogi, 2004) mantém a contagem das maiores frequências em sinopses separadas para aproximar *joins* em *fluxos de dados*; por fim, analogamente, existem trabalhos sobre estimativa do produto interno (e.g., (Daliri *et al.*, 2024; Wang *et al.*, 2023)), equivalente ao tamanho do *join*. Apesar desses avanços, otimizações para a estrutura de *self-joins* permanece uma área pouco explorada. Esses fatores reforçam nossa motivação para investigar técnicas de estimativa de cardinalidade em consultas de *self-join*.

As principais contribuições deste trabalho são:

- Uma revisão das principais técnicas de estimativa de cardinalidade;
- A implementação de um gerador de dados sintéticos, desenvolvido com o objetivo de ser utilizado no benchmark das técnicas de estimativa aplicadas a *self-join*;
- A condução de um experimento comparativo, avaliando o desempenho de diferentes estratégias para estimativa de cardinalidade aplicadas à consultas de *self-join*.

Para a clareza deste trabalho, é importante distinguirmos os conceitos de cardinalidade, estimativa de cardinalidade e estimativa de seletividade. A cardinalidade é o número de elementos distintos em um conjunto — embora comumente confundida com o tamanho. Estimativa de cardinalidade é uma das tarefas realizadas pelo otimizador para estimar o tamanho da saída (i.e., o número de tuplas) do processamento de uma expressão de consulta. A seletividade é a fração de tuplas que qualificam um predicado de uma expressão, no qual se multiplica sua estimativa pelo tamanho da entrada para obter a estimativa de cardinalidade. Com base nessas distinções, o termo "estimativa" será utilizado neste texto de forma que o contexto esclareça a qual conceito se refere.

O restante deste trabalho está organizado da seguinte forma: no Capítulo 2 revisamos os principais mecanismos atualmente utilizados por SGBDs, bem como as técnicas apresentadas pelos estudos mais relevantes relacionados à nossa pergunta de pesquisa. Os materiais e métodos necessários são apresentados no Capítulo 3. No Capítulo 4 analisamos os resultados obtidos. Por fim, no Capítulo 5 sumarizamos nossas conclusões.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Processamento e Otimização de Consultas

Devido à sua natureza declarativa, consultas SQL com estruturas textuais distintas podem produzir o mesmo resultado. Otimizadores buscam explorar esse fato por meio de diversas estratégias. Uma tarefa em comum consiste em representar a consulta como uma árvore de expressões da álgebra relacional (estendida). Cada nó da árvore representa uma operação a ser processada e a ordem de execução inicia a partir das folhas. Cada nó é substituído pelo resultado de sua execução e o processamento da consulta acaba quando a raiz finaliza sua execução. Além disso, ramos da árvore podem ser transformados por regras de equivalência.

Para gerar o plano de execução físico, o otimizador decide o método de acesso e o algoritmo que implementa o operador algébrico de cada expressão da árvore. Essa escolha é baseada em um modelo de custo. O modelo introduzido por (Selinger *et al.*, 1979) utiliza diferentes fórmulas para avaliar o custo de diferentes tipos de consulta. Por exemplo: consultas que acessam uma única tabela utilizam a fórmula de custo:

$$\text{custo} = \#\ \textit{acesso a blocos em disco} + W * \textit{RSI CALLS}, \tag{1}$$

onde W é um fator de balanceamento entre operações de E/S (blocos lidos) e CPU (instruções executadas). RSI CALLS corresponde ao número estimado de tuplas retornadas. Na otimização de consultas, essa estimativa é geralmente conhecida como estimativa de cardinalidade, que se refere à previsão do número de linhas (ou tuplas) que um operador de consulta ou uma consulta completa irá produzir.

Para estimar essa cardinalidade, utilizam-se estatísticas armazenadas no catálogo. Entretanto, um dos principais desafios do modelo de custo ocorre quando essas estatísticas não estão disponíveis. Nesses casos, são feitas suposições sobre a distribuição dos dados, como uniformidade e independência. Essas suposições representam limitações do modelo devido à ausência de alternativas mais precisas (Moerkotte, 2024). Com isso, este trabalho tem como foco uma revisão da literatura sobre algoritmos e técnicas de estimativa de cardinalidade.

2.1.1 Estatísticas

Enumerar exaustivamente todos os planos de execução semanticamente equivalentes não é viável para o otimizador, pois o tempo gasto na otimização pode superar o da execução da consulta. Para estimar o custo das consultas e decidir quais planos explorar, os SGBDs mantêm estatísticas no catálogo do sistema. Essas estatísticas podem descrever tanto aspectos físicos (e.g., número de páginas) quanto lógicos (e.g., cardinalidade de um atributo e valores mínimo e máximo). Como este trabalho foca na estimativa de cardinalidade, a seguir consideramos

apenas os aspectos lógicos. Tradicionalmente (Selinger *et al.*, 1979), para uma relação R e atributos com índices, as estatísticas armazenadas são:

- **Cardinalidade da relação**: o número de tuplas;
- **Cardinalidade de um atributo**: o número de valores distintos de um atributo;
- **Valor mínimo de um atributo**: o menor valor de um atributo;
- **Valor máximo de um atributo**: o maior valor de um atributo.

Além disso, as estatísticas são essenciais por capturarem a assimetria na distribuição dos dados, eliminando a necessidade, em alguns casos, de recorrer a suposições tradicionais como uniformidade dos dados, independência e princípio da inclusão (Leis *et al.*, 2018).

Outro aspecto importante relacionado às estatísticas é sua manutenção. Como os dados de uma tabela estão sujeitos a atualizações frequentes, as estatísticas armazenadas podem rapidamente se tornar obsoletas. Atualizá-las a cada modificação resultaria em uma sobrecarga significativa. Por isso, os SGBDs costumam adotar estratégias baseadas em limiares: as estatísticas são atualizadas apenas quando a proporção de dados modificados ultrapassa um determinado limite ou durante períodos de baixa carga no sistema.

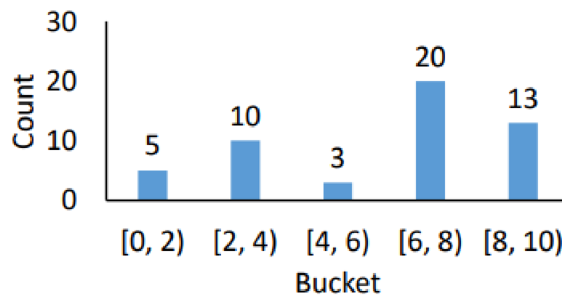
2.1.1.1 Histogramas

Devido à sua simplicidade, histogramas são uma das técnicas mais utilizadas para representar a distribuição da frequência de valores (Ioannidis, 2003). Na prática, eles armazenam a distribuição de valores de atributos de uma relação. No entanto, manter um histograma de todos os valores únicos para cada atributo pode ser custoso, por exemplo: o histograma de um atributo chave primária A em uma relação R com n tuplas, teria n colunas. Para mitigar esse custo, uma das alternativas mais utilizadas é o agrupamento dos valores do histograma tradicional em subconjuntos disjuntos (partições). Cada subconjunto, também conhecido como *bucket*, armazena a frequência acumulada dos valores contidos no grupo, além do número de valores distintos. Diversas estratégias podem ser adotadas para definir esses agrupamentos. A seguir, apresentamos algumas das mais comuns:

- Histograma de classes com mesma amplitude (*equi-width histogram*): os *buckets* possuem intervalos de mesmo tamanho, onde cada *bucket* representa uma faixa contínua de valores. A Figura 1 ilustra um exemplo desse histograma com 5 *buckets*.
- Histograma de classes com mesma altura (*equi-depth histogram* ou *equi-height histogram*): os *buckets* são formados de forma que a contagem da frequência de cada *bucket* seja aproximadamente a mesma, o que pode resultar em intervalos de tamanhos diferentes. Um exemplo desse histograma é apresentado na Figura 2.

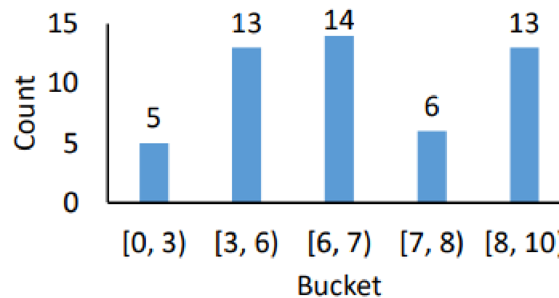
- Histograma com viés (*end-biased histogram*): define-se um número N de *buckets*, onde $N-1$ armazenam os valores de maior frequência (conhecidos como *heavy hitters*), enquanto o *bucket* restante representa a média da frequência dos demais valores. A Figura 3 ilustra um exemplo na qual a classe R representa essa média.

Figura 1 – Exemplo de *equi-width histogram*



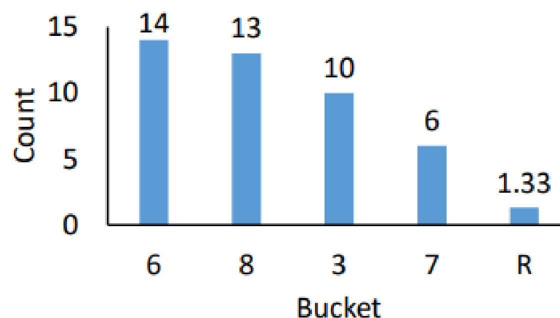
Fonte: Ding, Narasayya e Chaudhuri (2024).

Figura 2 – Exemplo de *equi-depth histogram*



Fonte: Ding, Narasayya e Chaudhuri (2024).

Figura 3 – Exemplo de *end-biased histogram*



Fonte: Ding, Narasayya e Chaudhuri (2024).

2.1.1.2 Amostragem

Outra estatística comumente utilizada na estimativa de cardinalidade e na construção de histogramas é a amostragem (*sampling*), que, assim como os histogramas, destaca-se pela

simplicidade. No entanto, diversos estudos consolidados apontam que a amostragem tende a ser ineficiente, especialmente em conjuntos de dados grandes (Harmouch; Naumann, 2017). O desempenho inferior da amostragem, em comparação com outros métodos, deve-se principalmente à sua dificuldade em garantir que a amostra inclua os elementos mais relevantes para a estimativa. Em outras palavras, a amostragem é sensível à assimetria e à presença de valores atípicos, podendo ter uma representação excessiva de partes irrelevantes da relação para a consulta, resultando em estimativas imprecisas. Para mitigar o problema, podem ser utilizadas técnicas de amostragem aleatória.

2.1.1.3 *Sketches*

Estruturas de dados sinopse pertencem a uma classe de estruturas de dados que são utilizadas para adquirir propriedades sobre um grande volume de dados. *Sketches* são *sinopses* com um custo de espaço substancialmente menor ao conjunto de dados original. Existe uma perda de informação pelo fato de serem compactas — não fornecem respostas exatas — entretanto, esse relaxamento é aceitável, pois as estatísticas adquiridas possuem uma acurácia aproximada na maioria dos casos (Cormode *et al.*, 2012). *Sketches* são relevantes no contexto de SGBDs pois, além de fornecerem as vantagens descritas, permitem que suas estruturas sejam incrementais, i.e., processam atualizações sem a necessidade de recomputar todos os dados.

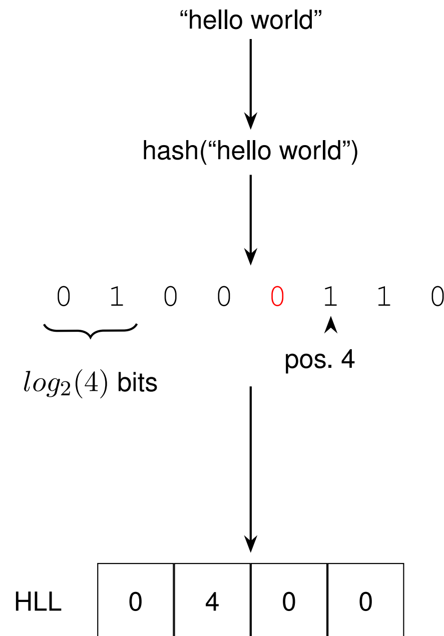
HyperLogLog. É uma estrutura de dados probabilística quase ótima — alcança uma estimativa precisa enquanto utiliza assintoticamente a menor quantidade de memória necessária para o problema—, voltada à estimativa do número de elementos distintos (cardinalidade) para conjuntos de dados de grande volume. Foi inicialmente proposta para lidar com problemas relacionados a fluxos de dados (Flajolet *et al.*, 2007), mas também é altamente adequada para aplicações em SGBDs.

O algoritmo possui um parâmetro M que define o número de registradores (*buckets*) e um parâmetro $b \in \mathbb{Z}^+$ que define a quantidade de bits iniciais para representar *hashes* dos valores distintos, onde $M = 2^b$. Cada elemento do conjunto passa por uma função *hash* que mapeia valores à strings binárias de forma $\{0, 1\}^\infty$. Os primeiros b bits da string são utilizados para identificar o *bucket* m correspondente ao elemento. Em seguida, determina-se a posição p do primeiro bit igual a 1, considerando apenas os bits subsequentes a b, isto é, p corresponde ao número de zeros à esquerda após b bits, mais 1. Ao final, atribui-se ao bucket correspondente $M[m] := max(M[m], p)$. A Figura 4 demonstra o funcionamento do algoritmo por meio de um exemplo com 4 *buckets* e a entrada “`hello world`”.

Após todos elementos serem processados, o algoritmo computa a função *indicadora*:

$$Z = \left(\sum_{i=1}^{M} 2^{-M[i]} \right)^{-1} \tag{2}$$

Figura 4 – Diagrama ilustrativo do funcionamento do HyperLogLog com M = 4 *buckets*



Fonte: Autoria própria (2025).

Finalmente, a estimativa de cardinalidade é dada por:

$$E = \alpha_m \cdot m^2 \cdot Z \tag{3}$$

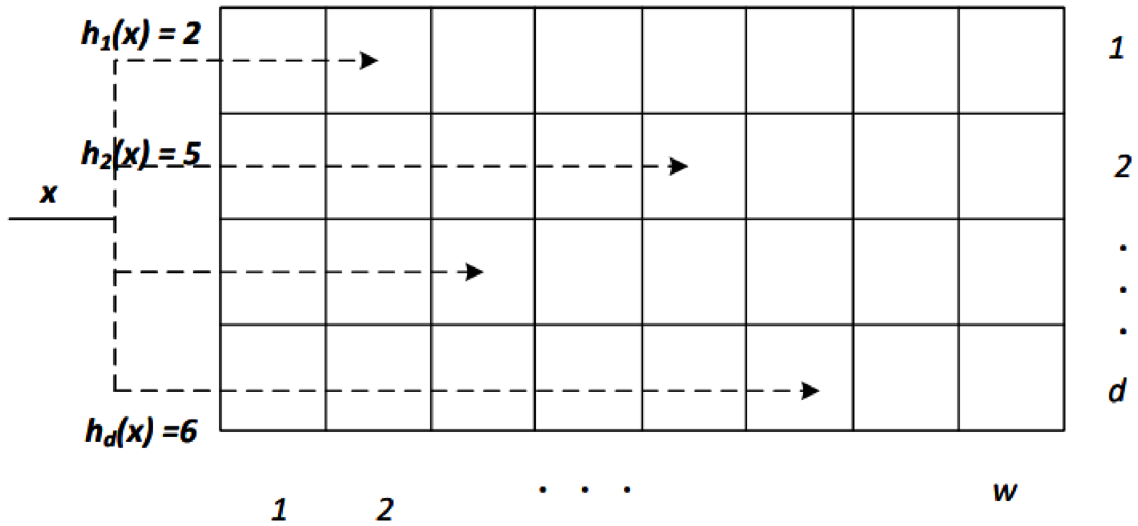
Uma descrição detalhada sobre a Equação 3 pode ser encontrada em Flajolet *et al.* (2007).

Count-Min Sketch. Diferentemente do *HyperLogLog*, o *Count-Min Sketch* é utilizado para estimar a frequência de um determinado elemento. A estrutura é implementada através de uma matriz A de tamanho $d \times w$, onde cada linha d possui uma *função hash independente* que mapeia um valor de entrada para $v \in \{1, 2, ..., w\}$. As células da matriz são contadores, inicializados em 0, e são incrementados quando elementos mapeados pela função resultam em sua posição. Computa-se, para todos os elementos, suas posições em todas as linhas da matriz, i.e., um elemento passa por d funções hash, como ilustrado na Figura 5. A estimativa de frequência de um valor x é dada por $min\ A_{j,h_j(x)}$, $j \in [1,d]$. O valor mínimo é escolhido pois a célula com o menor valor representa a estimativa com o menor “ruído” — onde houveram menos colisões.

Primordialmente, o *Count-min Sketch* foi projetado para estimar a frequência de um determinado valor distinto. Entretanto, no trabalho seminal de Cormode e Muthukrishnan (2005), os autores descrevem um procedimento para estimativa de produto interno entre duas estruturas *Count-min Sketch*. Como mencionado na introdução (Capítulo 1), o produto interno entre dois fluxos de dados é equivalente à estimativa de tamanho de *join* em SGBDs. Especificamente, o tamanho de *join* entre duas relações sobre um determinado atributo corresponde ao número de pares no produto cartesiano que compartilham o mesmo valor desse atributo.

O procedimento de estimativa pode ser resumido da seguinte forma: dadas duas estruturas *Count-Min Sketch* com dimensões $d \times w$, multiplicam-se as linhas correspondentes de

Figura 5 – Estrutura generalizada de um *Count-Min Sketch*



Fonte: Ding, Narasayya e Chaudhuri (2024).

cada estrutura e, em seguida, seleciona-se o produto de linha com o menor valor. A intuição por trás desse procedimento é semelhante à da estimativa de frequência de um valor distinto — sabendo que a *sketch* sempre superestima os valores devido à colisões, então, o menor valor representa a estimativa mais próxima do valor real.

JoinSketch. O *JoinSketch* é a *sketch* mais recente estudada neste trabalho. Introduzida por Wang *et al.* (2023), os autores descrevem que *sketches* como *Count-min Sketch* possuem baixo desempenho em dados reais que possuem alta assimetria. A partir disso, o *JoinSketch* é projetado com base na suposição de que dados do mundo real apresentam poucos valores frequentes e muitos valores infrequentes. Trabalhos anteriores apresentaram soluções com base nessa suposição, entretanto, nenhum obteve uma solução que exigisse apenas uma varredura sobre o conjunto de dados. Quando a distinção entre valores frequentes e infrequentes é realizada corretamente, a variância da estimativa do produto interno é reduzida substancialmente.

O *JoinSketch* é composto por três componentes: (i) uma estrutura para armazenar contagens exatas dos valores frequentes; (ii) uma estrutura para armazenar contagens exatas da frequência dos valores "intermediários"; e (iii) uma estrutura para armazenar contagens dos valores infrequentes. A parte intermediária é o componente chave do *JoinSketch*. Ela mantém a contagem dos valores que ainda não se enquadraram como frequentes ou infrequentes. Quando a contagem de um valor nessa parte ultrapassa um limiar T, o valor é adicionado no componente de valores frequentes e removido do componente intermediário. Caso o componente intermediário esteja cheio e um novo valor é inserido, o algoritmo busca e move o valor com menor frequência no componente intermediário para o componente de valores infrequentes, dealocando espaço para esse novo valor.

Os componentes frequente e intermediário são implementados com tabelas *hash* aumentadas que armazenam as contagens exatas de valores distintos. A parte infrequente utiliza *Fast-AGMS Sketches* (uma variação do *Tug-of-War*) — como apenas itens infrequentes são in-

seridos, o *JoinSketch* requer substancialmente menos memória do que uma solução baseada exclusivamente em *Fast-AGMS Sketches*.

Por fim, o *JoinSketch* também permite estimar a frequência de valores distintos.

2.1.2 Técnicas para Estimativa

Antes de abordarmos as principais técnicas, é importante classificá-las em três categorias, que são baseadas nas expressões lógicas que avaliam: (i) seleções (σ) em uma única relação; (ii) estimativa de valores distintos; e (iii) estimativa do tamanho de *join*. Essas categorias são utilizadas como blocos de construção para a otimização de consultas mais complexas.

Quadro 1 – Relação R_1

boat	id	color	years_in_service
t_1	1	yellow	1
t_2	2	blue	2
t_3	3	red	3
t_4	4	red	3
...	...	...	...
$t_{1\ 000\ 000}$	1 000 000	rusty	10

Fonte: Autoria própria (2025).

2.1.2.1 Estimativa de Seleção (σ)

Há diferentes formas de aplicar seleções sobre um atributo, e a estimativa de cardinalidade varia conforme o tipo de expressão. Por exemplo, consideremos predicados que possuam o seguinte padrão: $\sigma_{A\ op\ c}\ (R)$, em que A é um atributo de R; $op\ \in\ \mathcal{O}$, onde $\mathcal{O}\ \in\ \{=\ ,\ \neq\ ,\ \leq\ ,\ \geq\ ,\ <\ ,\ >\}$; e c é uma constante. A seguir, descrevemos diferentes formas por meio de exemplos sobre a relação apresentada na Tabela 1.

$\sigma_{A\ =\ c}$ (`boat`). Se nenhuma estatística sobre a contagem de c estiver disponível no catálogo, a estimativa é computada com base na suposição de que os valores seguem uma distribuição uniforme. Assim, a estimativa resulta da divisão entre o tamanho da relação R_1 e o número de valores distintos de A. No entanto, essa suposição raramente se sustenta em cenários do mundo real.

Caso contrário, a frequência aproximada de c está armazenada utilizando algum método de Seção 2.1.1 e é utilizada como estimativa resultante.

Em um exemplo, a Tabela 1 apresenta uma relação que armazena registros de barcos de uma empresa. Todos os barcos com 10 anos de serviço ou mais possuem cor enferrujada. Considere que apenas 1 000 barcos tenham ao menos 10 anos de serviço e que existam 6 valores únicos no atributo `color`. Então, a expressão $\sigma_{color\ =\ 'rusty'}$ seria estimada em $\frac{|R_1|}{6} \approx 166\ 667$, sendo uma superestimação do valor real, que é 1 000.

$\sigma_{A \le c}$ (`boat`). Se nenhuma estatística sobre a distribuição de A estiver disponível no catálogo, a solução é tomada a partir de uma suposição de distribuição uniforme dos dados e utilizando a seguinte função:

$$CardEst(\sigma_{A \le c}) = \begin{cases} 0 & \text{se } c < \min(A) \\ |R_1| & \text{se } c \ge \max(A) \\ |R_1| \times \left(\frac{c - \min(A)}{\max(A) - \min(A)} \right) & \text{caso contrário} \end{cases}$$

De modo similar, se o operador for $\ge$, estimamos o resultado com:

$$CardEst(\sigma_{A \ge c}) = \begin{cases} 0 & \text{se } c > \max(A) \\ |R_1| & \text{se } c \le \min(A) \\ |R_1| \times \left(\frac{\max(A) - c}{\max(A) - \min(A)} \right) & \text{caso contrário} \end{cases}$$

Em contrapartida, caso haja um histograma para o atributo A, somamos as frequências dos valores em $[min(A), c)$ para $<$ e em $(c, max(A)]$ para $>$.

Múltiplos predicados (predicados complexos). Estimamos seleções com múltiplos predicados considerando dois casos: conjunções e disjunções.

Uma seleção conjuntiva segue a forma:

$\sigma_{\theta_1 \wedge \theta_2 \wedge ... \wedge \theta_n}$, onde θ é um predicado de seleção.

A seletividade para essa seleção é calculada através de um produtório entre a seletividade de cada predicado. Para isso, supõe-se a independência dos atributos, i.e., os atributos entre quaisquer predicados da conjunção não possuem correlações. Por fim, a estimativa é dada por:

$$n_r \,.\, \frac{sel(\sigma_{\theta_1}) \times sel(\sigma_{\theta_2}) \times ... \times sel(\sigma_{\theta_n})}{n_r^n}$$, onde n_r é o tamanho da relação r.

Uma seleção disjuntiva segue a forma:

$\sigma_{\theta_1 \vee \theta_2 \vee ... \vee \theta_n}$, onde θ é um predicado de seleção.

A seletividade para uma seleção disjuntiva é calculada através da união entre as seletividades de cada predicado. Para isso, utiliza-se o Princípio da inclusão-exclusão. Novamente,

assume-se independência dos atributos. Por exemplo, uma seleção conjuntiva com dois predicados é calculada através de:

$$sel(\sigma_{\theta_1 \vee \theta_2}) = sel(\sigma_{\theta_1}) + sel(\sigma_{\theta_2}) - sel(\sigma_{\theta_1 \wedge \theta_2})$$
$$\equiv 1 - (1 - sel(\sigma_{\theta_1})) \times (1 - sel(\sigma_{\theta_2}))$$

Observe que a suposição de independência pode levar a estimativas de cardinalidade que diferem em várias ordens de magnitude do valor real. Evidenciamos o problema por meio de um exemplo sobre a relação da Tabela 1: considere que não existem histogramas sobre os atributos da tabela e que os valores de `years_in_service` pertencem à $\{1, 2, 3, \ldots, 10\}$, então, a seleção $\sigma_{color='rusty' \wedge years_in_service=10}$ seria estimada em $166\ 667 \times \frac{1\ 000\ 000}{10}$, sendo um valor consideravelmente superior ao esperado.

2.1.2.2 Estimativa do Número de Valores Distintos

A estimativa do número de valores distintos de atributos em um resultado intermediário nos ajuda a estimar o número de valores distintos de atributos nos resultados intermediários do próximo nível. Embora *sketches* possam ser armazenadas, é necessário que sejam calculadas previamente, o que implica que a relação precisa ser varrida pelo menos uma vez. Considerando que não existem sinopses disponíveis, a estimativa é efetuada através dos seguintes procedimentos.

Seleção. Suponha V(A, σ_θ (r)).

- Se θ força A assumir um valor específico, e.g., $\sigma_{A=2}(r)$, então, V(A, $\sigma_\theta(r)$) = 1.
- Se θ força A assumir um conjunto de valores, e.g., $\sigma_{A=1 \vee A=2 \vee A=3}$, então, V($A$, $\sigma_\theta(r)$) é a cardinalidade do conjunto.
- Se θ apresenta o padrão $A\ op\ v$, onde op é um operador de comparação, então, V(A, $\sigma_\theta(r)$) é $V(A, r) \times sel(\sigma_\theta(r))$.
- Para outros casos, quando θ não força nada sobre A, assumimos que a distribuição de valores A é independente da distribuição dos valores na qual a seleção especifica, e a estimativa é dada por $min(V(A, r), n_{\sigma_\theta(r)})$.

Join. Suponha V(A, $r \bowtie s$), onde A é um conjunto de atributos.

- Sem perda de generalidade, se todos os atributos de A pertencem exclusivamente à r, então, a estimativa é dada por $min(V(A, r), n_{r \bowtie s})$.
- Se A contém atributos das duas relações, e.g., A_1 e A_2 para r e s, respectivamente, então, a estimativa é dada por $min(V(A_1, r) \times V(A_2 - A_1, s), V(A_2, s) \times V(A_1 - A_2, r), n_{r \bowtie s})$.

Relação com *self-join*. Em algoritmos de *join*, como o *Hash Join* e o *Sort-Merge Join*, o número de valores distintos presentes em uma coluna é um fator determinante para o desempenho da operação. Quando a coluna utilizada na junção possui poucos valores distintos, a operação tende a possuir maior valor de seletividade, resultando em um conjunto de resultados maior. No caso do *Hash Join*, por exemplo, uma baixa cardinalidade leva à construção de uma tabela hash mais compacta, com maior chance de se manter na cache do processador. Isso reduz o número de acessos à memória principal e contribui para uma execução mais eficiente. De forma semelhante, no *Sort-Merge Join*, uma menor cardinalidade pode acelerar as etapas de ordenação e mesclagem.

Assim, conhecer a quantidade de valores distintos permite estimar a seletividade da junção, o que auxilia o SGBD a escolher um plano de execução mais adequado para a consulta.

2.1.2.3 Estimativa do Tamanho de *Join*

Sejam R e S relações, e A e B atributos respectivos de cada uma delas. O *join* $R \bowtie_{A=B} S$ pode ser estimado baseado nas seguintes definições:

- Se $A \cap B = \emptyset$, então, a estimativa é o produto cartesiano entre as relações R e S.
- Sem perda de generalidade, se $A \cap B$ é uma chave para A, ou seja, identificam unicamente tuplas em R, então, sabemos que cada tupla em S se juntará somente com uma tupla de R, consequentemente, a estimativa é no máximo o número de tuplas em S.
- Se $A \cap B$ não são chaves em nenhuma das relações, consideramos as seguintes suposições: (i) *uniformidade*, i.e., a frequência dos valores dos atributos da junção são uniformemente distribuídos; e (ii) *princípio da inclusão*, i.e., para cada valor no atributo com o menor domínio — o atributo que possui o menor número de valores distintos —, haverá no mínimo uma tupla correspondente na outra relação.

Como exemplo, considere o último caso. Estimamos que cada tupla $t \in R$ irá produzir

$$\frac{n_S}{V(\mathsf{B}, S)}$$

tuplas em $R \bowtie_{A=B} S$, onde $V(B, S)$ representa a cardinalidade do atributo B. Então, a estimativa resultante do *join* é dada por:

$$n_R \cdot \frac{n_S}{V(\mathsf{B}, S)}$$

Observe que, invertendo-se R por S na expressão anterior, a fórmula resultante é semelhante:

$$n_S \cdot \frac{n_R}{V(\mathsf{A}, R)}$$

É interessante notarmos esse aspecto, pois caso a cardinalidade entre os dois atributos seja diferente, haverá tuplas que não vão participar da junção (*dangling tuples*). Nesse cenário, a estimativa de menor valor tende a ser mais exata. Assim, a fórmula generalizada é dada por:

$$\frac{n_R \times n_S}{max(V(\mathsf{B}, S), V(\mathsf{A}, R))} \tag{4}$$

Se histogramas para os atributos estão disponíveis, a suposição de uniformidade não precisa ser mantida. Para fins de simplicidade, consideraremos um exemplo onde, neste primeiro caso, os histogramas dos dois atributos estão alinhados, ou seja, possuem o mesmo número de *buckets* e que *buckets* correspondentes têm a mesma largura. A estimativa de junção entre dois buckets alinhados é calculada por meio da Equação 4. Então, a estimativa final é obtida pela soma das estimativas de junção entre todos os pares de *buckets*. Caso os histogramas não estejam alinhados, uma das abordagens consiste em alinhar o histograma com maior número de buckets aos limites dos buckets do histograma com menor número.

2.1.2.4 Estimativa do Tamanho de *self-join*

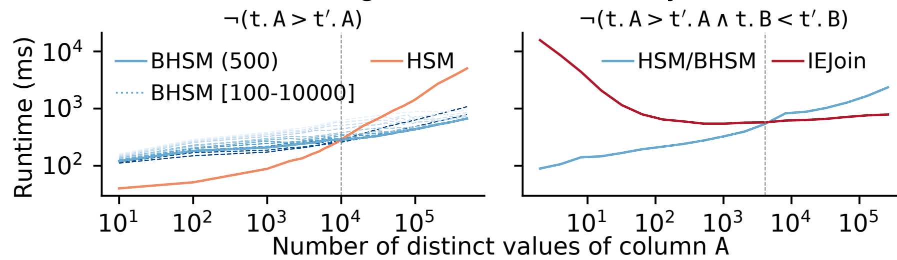
A estimativa do tamanho de *self-join* é realizada utilizando as técnicas apresentadas na Seção 2.1.2.3. Adicionalmente, outra abordagem é apresentada na Seção 3.1.2.1. Contudo, é importante ressaltarmos que as técnicas se aplicam apenas para *equi self-joins*.

Observamos que há uma lacuna de trabalhos sobre estimativa de tamanho de *join* para operadores não equitativos, i.e., $<, >, \leq, \geq$, e $\neq$. Em uma pesquisa na literatura, encontramos somente um estudo (Repas *et al.*, 2022) em acesso antecipado (pré-publicação) no arXiv que aborda a estimativa de *joins* com operadores de desigualdade. Os autores apresentam uma solução baseada em histogramas de classes com mesma altura que realiza a estimativa utilizando da função de densidade de probabilidade e função de distribuição acumulada dos atributos do *join*. Por fim, *self-joins* também são explorados em outros contextos. Por exemplo, (Rafiei; Deng, 2020) estudam a similaridade de *joins* e *self-joins* em *streaming* de dados.

Relação com *self-join* com operadores de desigualdade. *Self-joins* com operadores de desigualdade, como $<, >, \leq, \geq$, representam um desafio adicional para a estimativa de tamanho e ordenação dos predicados, especialmente quando há múltiplas condições envolvidas. A Figura 6, extraída de (Pena; Almeida; Naumann, 2021), ilustra o impacto da cardinalidade das colunas no tempo de execução de três algoritmos: *HSM*, *BHSM* e *IEJoin* (visto em (Khayyat *et al.*, 2015)).

O algoritmo *HSM* (*Hash-Sort-Merge*) realiza junções de intervalo utilizando uma combinação de hashing e ordenação para processar os dados de forma eficiente, especialmente quando a cardinalidade das colunas é baixa. Já o *BHSM* (Binned HSM) estende o *HSM* ao aplicar uma técnica de *binning*, agrupando os valores da coluna em intervalos discretos (*bins*), o que se mostra vantajoso em cenários com alta cardinalidade. Por fim, o *IEJoin* é um algoritmo utiliza ordenação e *bitsets* para avaliar pares de tuplas de forma eficiente, sendo limitado a apenas dois predicados por execução.

Figura 6 – Os diferentes impactos da cardinalidade das colunas no tempo de execução dos algoritmos hsm, bhsm e iejoin



Fonte: Pena, Almeida e Naumann (2021).

Os resultados apresentados na figura mostram que o *HSM* é mais eficiente que o *BHSM* em colunas com até aproximadamente 10.000 valores distintos, pois a sobrecarga associada ao gerenciamento de *bins* no *BHSM* não compensa nesse intervalo. Acima desse limiar de cardinalidade, o *BHSM* passa a ter melhor desempenho, já que sua estrutura permite um controle mais eficiente da distribuição dos dados e da verificação de candidatos. O *IEJoin*, por sua vez, não é diretamente afetado pela cardinalidade da mesma forma, mas sua limitação a dois predicados reduz sua aplicabilidade em junções com múltiplas desigualdades.

A implicação prática é que, ao se realizar *self-joins* com desigualdades, a escolha do algoritmo deve considerar cuidadosamente a cardinalidade das colunas envolvidas. Em especial, para colunas com baixa cardinalidade, o *HSM* se mostra mais eficiente, enquanto que o *BHSM* é preferível para colunas mais diversas. Além disso, os autores sugerem uma abordagem gananciosa, denominada *GreedyHLL*, para ordenar os predicados a partir de estimativas de cardinalidade e custo de avaliação, priorizando a redução de intermediários e a eficiência do plano de execução.

Esses achados são especialmente relevantes para sistemas que precisam lidar com *self-joins* complexos e de alta dimensionalidade, como consultas de similaridade, filtragens temporais e operadores analíticos baseados em desigualdades.

3 DESENVOLVIMENTO

3.1 Materiais e métodos

Um dos objetivos deste trabalho é investigar, empiricamente, o comportamento de estimadores sobre diferentes distribuições e tamanhos de dados. Nesta seção, descrevemos todos os materias e métodos necessários para a condução dos experimentos.

3.1.1 Conjuntos de Dados

Para a realização do experimento, selecionamos conjuntos de dados de diferentes origens. Essa diversidade é importante em nosso contexto, pois *benchmarks* (e.g., *TPC-H*) frequentemente utilizados em experimentos com SGBDs apresentam a inconveniência de terem seus dados uniformemente distribuídos (Freitag; Neumann, 2019).

3.1.1.1 *Benchmark*

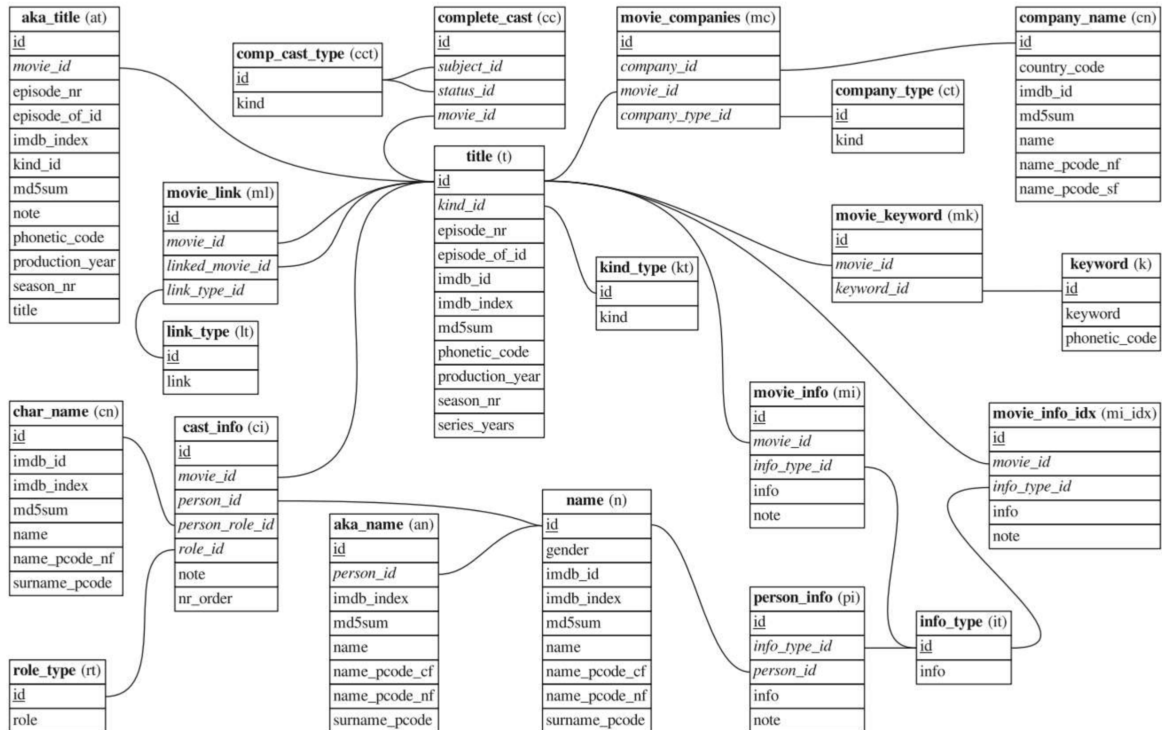
O *TPC-H* é um *benchmark* que modela o negócio de uma empresa de produtos. Ele utiliza um esquema composto por oito tabelas. Os tamanhos (GB) dessas tabelas variam proporcionalmente ao parâmetro de fator de escala, com valores: 1, 10, 30, 100, 300, 1000, 3000, 10000, 30000 e 100000. As tabelas são populadas através de um gerador de dados sintéticos disponibilizado pela própria ferramenta. Por fim, o *benchmark* possui uma carga de trabalho que consiste em 22 consultas que representam consultas analíticas (Thanopoulou; Carreira; Galhardas, 2012).

3.1.1.2 Dados reais

Trabalhos anteriores ((Leis *et al.*, 2015) e (Leis *et al.*, 2018)) apresentam que a estimativa de cardinalidade é, possivelmente, o fator primordial do modelo de custo. Consequentemente, se as estimativas são substancialmentes distantes do valor real, o otimizador pode selecionar uma solução não ótima (ou distante da ótima). Nessa perspectiva, utilizar um *benchmark* com dados uniformemente distribuídos pode mascarar o desempenho relativo sobre estimadores em cargas de trabalho do mundo real. Isso ocorre devido às suposições sobre co-dependência apresentadas na Seção 2.1.2. Adicionalmente, conjuntos de dados do mundo real frequentemente apresentam distribuições assimétricas e correlações entre os atributos (Leis *et al.*, 2018), propriedades que os dados sintéticos não conseguem replicar.

Inspirado em Leis *et al.* (2018), selecionamos o conjunto de dados *Internet Movie Database* (IMDb) para ilustrar uma instância de dados do mundo real em nossos experimentos. A Figura 7 mostra o esquema do conjunto de dados; ademais, a Seção 4.1 especifica as tabelas utilizadas.

Figura 7 – Esquema do conjunto de dados IMDb



Fonte: Leis *et al.* (2018).

3.1.1.3 Dados sintéticos

Empiricamente, Lee *et al.* (2023) demonstrou que existe uma alta variabilidade no impacto das estimativas sobre a qualidade dos planos em diferentes cargas de trabalho. Isso decorre de uma escassez de consultas que sejam capazes de evidenciar a distribuição enviesada (i.e., assimétrica) e a correlação dos atributos. Por fim, Lee *et al.* (2023) estabelece que o conjunto de consultas de *benchmarks* frequentemente utilizados precisa ser aumentado.

Fundamentado nessa conclusão, levantamos a hipótese de que um conjunto de dados com parâmetros configuráveis (e.g., família de distribuições, média, desvio padrão, assimetria, valores mín./máx., etc.) pode facilitar a elaboração de consultas que enfatizem a distribuição assimétrica ou a correlação dos atributos. A necessidade de controlar parâmetros sobre a distribuição dos atributos motivou o desenvolvimento de um gerador de dados sintéticos — disponível como artefato deste[1].

O gerador suporta atributos numéricos (discretos ou contínuos) e categóricos. Existem duas classes de atributos: contagem por agrupamento e convencional. Para cada conjunto de dados, só é possível gerar um atributo de contagem por agrupamento. Esse atributo é configurado por um dicionário, onde as chaves representam o tamanho de cada grupo e os valores indicam a frequência de grupos daquele tamanho. Por exemplo, o dicionário `{2:2, 3:4}` gera uma coluna com dois grupos de tamanho 2 e quatro grupos de tamanho 3, totalizando $(2*2)+(4*3)=16$ tuplas.

[1] https://github.com/atenokida/tcc

Tabela 2 – Configurações disponíveis no gerador de dados

Tipo do Atributo	Distribuição	Parâmetros
Numérico	Uniforme	-
	Normal	Média, Desvio Padrão
	Zipf	Assimetria
Categórico	Uniforme	-
	Normal	Desvio Padrão

Fonte: Autoria própria (2025).

O gerador permite um número ilimitado de atributos convencionais (numéricos e categóricos). Para cada distribuição, é possível definir sua família de distribuição (e.g., normal, uniforme, zipf), bem como os parâmetros específicos dessa distribuição. A Tabela 2 apresenta a lista completa de distribuições suportadas e seus respectivos parâmetros.

O número de registros para os atributos convencionais é limitado pelo número de tuplas gerado na contagem por agrupamento, i.e., $\sum_{i=1}^{n} k_i \cdot v_i$, em que k_i e v_i representam as chaves e os valores, respectivamente. Por fim, o conjunto de dados é montado pela concatenação de todas as colunas.

Evidenciamos que o gerador não está finalizado e pretendemos aumentá-lo em trabalhos futuros. Por exemplo, a criação de variáveis dependentes é uma *feature* importante ainda não implementada. Outra *feature* importante seria a incorporação de técnicas de injeção de erros, uma vez que uma das nossas motivações para estimativa de cardinalidade é a validação de regras de qualidade de dados (como, por exemplo, DCs).

3.1.2 Ferramentas

3.1.2.1 Algoritmos de Estimativa

Na Seção 2.1.1.3 apresentamos aplicações de *sketches* para estimativas de diferentes operadores da álgebra relacional (cardinalidade, seleção e junção). Nesta seção, focamos em *sketches* para estimativa de cardinalidade de *self-joins*. Descrevemos a *sketch* seminal *Tug-of-War* de Noga Alon (1999), que foi a primeira estrutura na literatura especificamente projetada para estimativa de *joins* (Harmouch; Naumann, 2017). Por fim, detalhamos as *sketches* e implementações utilizadas neste trabalho na Seção 3.1.4.

Tug-of-War. Para estimar o tamanho de um join entre duas relações, Alon *et al.* (1999) propuseram uma solução que utiliza a frequência dos momentos. A frequência dos momentos, para um vetor de frequências $A = (a_1, a_2, ..., a_m)$ onde a_i é a frequência de cada valor distinto em um conjunto de dados, é definida por:

$$F_k = \sum_{i=1}^{m} a_i^k \tag{5}$$

Observe que:

- o momento 0, i.e., $k = 0$, é a cardinalidade do conjunto;
- o primeiro momento, i.e., $k = 1$, é o tamanho do conjunto;
- o segundo momento, i.e., $k = 2$, é o tamanho do *self-join* sobre o atributo A do conjunto.

No entanto, construir vetores de frequência sobre grandes volumes de dados é uma tarefa computacionalmente custosa. Para isso, Alon *et al.* (1999) propuseram uma solução baseada em *sketches*. O pseudoalgoritmo 2 sumariza a ideia por trás da solução introduzida.

Algoritmo 2 – Tug-of-War Sketch

requer Conjunto de dados $x_1, x_2, \ldots, x_n$
inserir Estimativa de F_2
1: Seja $h : [n] \to \{\pm 1\}$ uma função escolhida de uma família *hash* universal 4-independente $\mathcal{H}$
2: $z \leftarrow 0$
3: **para** $i = 1$ até n **faça**
4: $\quad z \leftarrow z + h(i)$
5: **finaliza para**
6: **retorna** z^2

Fonte: Alon *et al.* (1999).

Uma das limitações sobre o *Tug-of-War* é sua alta variância, que pode resultar em estimativas de baixa acurácia. Para mitigar o problema, Alon *et al.* (1999) realizaram a execução de múltiplas instâncias paralelas, combinada com a técnica de estimativa "*mediana das médias*". A técnica consiste em: (i) particionar as variáveis aleatórias (i.e., cada z_2) em grupos de mesmo tamanho; (ii) computar a média de cada grupo; por fim, (iii) obter a mediana desses valores.

3.1.3 Parâmetros Avaliados

Utilizamos o *q-error* para avaliar a distância entre as estimativas e os valores reais. O *q-error* é uma métrica de erro amplamente utilizada para avaliar técnicas de estimativa de cardinalidade (e.g., (Leis *et al.*, 2015; Leis *et al.*, 2018; Lee *et al.*, 2023; Moerkotte, 2024)).

Introduzida por (Moerkotte; Neumann; Steidl, 2009), é defendida pelos autores como a mais adequada para a avaliação do desempenho de estimadores. Uma das principais características do *q-error*, quando comparado com outras métricas, é a garantia de correlação entre seu resultado e a qualidade do plano de execução gerado pelo otimizador. Através de resultados teóricos, (Moerkotte; Neumann; Steidl, 2009) provam que o custo do melhor plano de execução é obtido desde que o *q-error* não ultrapasse um limiar. Consequentemente, é crucial que as soluções de estimativa busquem minimizar o *q-error*.

Sua definição é dada pela fórmula:

$$\text{q-error}(x, \hat{x}) = \begin{cases} 1 & \text{se } x = 0 \wedge \hat{x} = 0 \\ max(\frac{x}{\hat{x}}, \frac{\hat{x}}{x}) & \text{se } x \neq 0 \wedge \hat{x} \neq 0 \\ \infty & \text{caso contrário} \end{cases} \quad (6)$$

onde x é a cardinalidade real, estimada em $\hat{x}$.

Outra métrica utilizada para calcular o erro de uma estimativa é a norma l_2 (euclidiana). A seguir, apresentamos sua aplicação por meio de um exemplo: considere dois vetores de frequências $b = (f_1, f_2, ..., f_m)$ e $\hat{b} = (\hat{f}_1, \hat{f}_2, ..., \hat{f}_m)$, da cardinalidade real e estimada, respectivamente. A fórmula para o erro l_2 é dada por:

$$\sqrt[n]{\sum_{i=1}^{m} |f_i - \hat{f}_i|^n} = \sqrt{|f_1 - \hat{f}_1|^2 \; + \; |f_2 - \hat{f}_2|^2 \; + \; ... \; + |f_m - \hat{f}_m|^2}$$

Além disso, considere as estimativas 1 ($\hat{e}_1$) e 100 ($\hat{e}_2$) sobre uma expressão (e) com cardinalidade 10. Então, os erros l_2 são: $\hat{e}_1 = 9$ e $\hat{e}_2 = 90$.

No entanto, independente do fator de diferença entre as duas estimativas, subestimativas podem ser tão catastróficas quanto superestimativas. Ao contrário da norma l_2, o *q-error* trata subestimações e superestimações de forma simétrica. A intuição de utilizar a razão no lugar da diferença absoluta ou quadrática é suficiente, pois somente as diferenças relativas importam nas escolhas dos planos de execução (Leis *et al.*, 2015).

3.1.4 Implementação

Para a execução dos experimentos avaliados neste estudo, priorizamos implementações existentes de trabalhos anteriores. Nossa abordagem busca garantir a reprodutibilidade dos experimentos e focar na análise dos resultados — considerando as restrições de tempo.

A lista a seguir apresenta as implementações existentes utilizadas em nosso estudo:

- *Tug-of-War* e *JoinSketch*: O trabalho de Wang *et al.* (2023) apresenta um experimento comparativo entre as *sketches*: *JoinSketch*, *AMGS* (*Tug-of-War*), *Fast AGMS*, *Skimmed Sketch* e *Count-min Sketch*. Com exceção ao *Count-min Sketch*, o código-fonte dos estimadores encontram-se disponíveis no repositório do GitHub dos autores[2].

- *Count-min Sketch*: A implementação utilizada foi desenvolvida pelos próprios autores da *sketch* e está disponível no sítio do Laboratório de Análise de Dados Massivos da Universidade de Rutgers[3].

[2] https://github.com/JoinSketch/JoinSketch
[3] https://www.dimacs.rutgers.edu/~graham/code/

Denotamos que foram realizadas adaptações específicas para garantir que as *sketches* fossem capazes de lidar com diferentes tipos de predicados (veja Seção 4.3).

4 EXPERIMENTOS

Neste capítulo, avaliamos os algoritmos em diferentes conjuntos de dados. Especificamente, comparamos os desempenhos (tempo de execução e estimativa) dos algoritmos sobre diferentes famílias de distribuições.

Inicialmente, descrevemos os conjuntos de dados e os predicados utilizados nos experimentos. Em seguida, de forma breve, categorizamos as diferentes cláusulas de *self-joins*, com base na combinação da quantidade de predicados e operadores de comparação. Por fim, ressaltamos os procedimentos dos algoritmos previamente descritos, os parâmetros adotados em cada um deles e as adaptações necessárias para sua aplicação neste estudo.

Os experimentos deste trabalho foram executados em um *desktop* equipado com um processador Intel Core i5-10400F (6 núcleos, 12 *threads*, 2.9 GHz, 12 MB cache), 16 GB de memória RAM DDR4-2666 e um dispositivo de armazenamento NVMe SSD de 1 TB. A sequência de execução dos experimentos segue a enumeração dos predicados utilizados (veja Tabela 4).

Cada algoritmo foi executado cinco vezes sobre cada predicado; a média aritmética das execuções foi utilizada como resultado. Os parâmetros que definem os tamanhos das *sketches Tug-of-War* (quantidade e tamanho dos grupos) e *Count-min Sketch* (profundidade e largura) foram determinados por meio de tentativa e erro — os resultados apresentados correspondem à configuração de parâmetros que obteve o melhor desempenho em nossos experimentos. No caso do *JoinSketch*, foram adotados os mesmos parâmetros descritos no trabalho original de Wang *et al.* (2023). Ressaltamos que uma das principais vantagens das *sketches* é o consumo de memória reduzido, característica que as torna adequadas para cenários de *big data*.

4.1 Conjuntos de dados

Nesta seção, descrevemos os conjuntos de dados empregados nos experimentos.

Sintético uniforme. Geramos um conjunto de dados sintético composto por três atributos que simulam informações corporativas. Os atributos `age`, `salary` e `department` são, respectivamente, discreto, discreto e categórico. Os valores de cada atributo foram distribuídos uniformemente. O conjunto foi produzido em cinco tamanhos: 10K, 100K, 500K, 1M e 2M. Esse dataset é fundamental para nosso experimento, pois permite demonstrar que a estimativa baseada em suposições (Seção 2.1.2.4) possui bom desempenho sob a condição de uniformidade.

Sintético normal. Similarmente ao sintético uniforme, geramos outro conjunto mantendo os mesmos tamanhos e o mesmo número de atributos. Nesse conjunto de dados, os atributos `age` e `salary` seguem distribuições normais, enquanto `department` mantém distribuição uniforme. Os parâmetros, definidos arbitrariamente, são apresentados na Tabela 3.

Sintético Zipf. Foi gerado ainda um conjunto de dados para distribuições Zipf. Para esse, utilizamos os mesmos tamanhos e atributos do dataset sintético uniforme, com a diferença de que

Tabela 3 – Parâmetros da distribuição dos atributos age e salary

Atributo	Tipo	μ	σ	Mín.	Máx.
age	Discreto	41	12	18	65
salary	Contínuo	75.000	15.000	30.000	120.000

Fonte: Autoria própria (2025).

age e salary seguem distribuição de Zipf com $\alpha = 1.8$, enquanto department permanece uniforme.

IMDb. Como mencionado na Seção 3.1.1, selecionamos o conjunto de dados do IMDb para representar um cenário do mundo real. O conjunto de dados original[1] contém sete tabelas; entretanto, com base na abordagem de (Leis *et al.*, 2015; Leis *et al.*, 2018), adotamos o esquema definido pelos autores no *benchmark* JOB[2]. Nos experimentos, utilizamos apenas duas tabelas: (i) movie_companies, que relaciona obras audiovisuais (filmes, séries etc.) às empresas responsáveis por sua produção; e (ii) person_info, que compõe informações sobre pessoas envolvidas nessas obras. Ambas as tabelas, compostas por 5 atributos cada, foram particionadas nos tamanhos 10K, 100K, 500K, 1M e 2M para manter a consistência de tamanho entre os conjuntos de dados.

TPC-H. Utilizamos a extensão TPC-H do DuckDB[3] para gerar o conjunto de dados do *benchmark* TPC-H. O DuckDB é um SGBD analítico embutido. SGBDs embutidos são sistemas que se vinculam a outros programas como bibliotecas e executam sobre seus processos. Para este trabalho, empregou-se somente a tabela lineitem. Por fim, os mesmos tamanhos de particionamento descritos anteriormente foram mantidos.

4.2 Predicados

Antes de apresentarmos as cláusulas de *self-join* utilizadas neste estudo, é necessário definir os diferentes tipos de predicados, uma vez que tais definições são essenciais para a análise comparativa dos resultados dos algoritmos posteriormente.

Seja uma relação R com conjunto de atributos $\{A, B, C\}$ e t uma tupla de R. Uma cláusula de *self-join* sobre R possui predicado no formato $t.X\ op\ t'.Y$, onde $op \in \{=, \neq, <, \leq, >, \geq\}$ é um operador de comparação, $X, Y \in \{A, B, C\}$ são atributos da relação, e $t \neq t' \in R$ duas tuplas distintas.

Um predicado é homogêneo quando o mesmo atributo aparece em ambos os lados do predicado, ou seja, $X = Y$; caso contrário, o predicado é denominado heterogêneo. Se a cláusula de *self-join* contém mais de um predicado, dizemos que a cláusula é um predicado complexo (neste trabalho, consideramos apenas predicados conjuntivos homogêneos); contrariamente, em contextos com um predicado, predicado simples.

[1] https://developer.imdb.com/non-commercial-datasets/
[2] http://event.cwi.nl/da/job/imdb.tgz
[3] https://duckdb.org/docs/stable/core_extensions/tpch

Essas definições são relevantes, pois nem todos os algoritmos deste estudo suportam diferentes combinações de predicados. A Seção 4.3 apresenta os resultados experimentais e descreve as adaptações necessárias nas *sketches* para suportar diferentes tipos de predicados de *self-join*. A Tabela 4 sumariza os predicados utilizados neste estudo.

Tabela 4 – Predicados utilizados nos experimentos

Conjunto de dados	Predicado	Símbolo
Distribuição Normal	$t.age = t'.age$	φ_1
	$t.age = t'.age$ AND $t.department = t'.department$	φ_2
	$t.department = t'.department$	φ_3
Distribuição Uniforme	$t.age = t'.age$	φ_4
	$t.age = t'.age$ AND $t.department = t'.department$	φ_5
	$t.department = t'.department$	φ_6
IMDb (Movies)	$t.company_id = t'.company_id$ AND $t.company_type_id = t'.company_type_id$	φ_7
IMDb (Person)	$t.person_id = t'.person_id$ AND $t.info_type_id = t'.info_type_id$	φ_8
TPC-H	$t.l_quantity = t'.l_quantity$ AND $t.l_partkey = t'.l_partkey$ AND $t.l_shipmode = t'.l_shipmode$	φ_9
	$t.l_quantity = t'.l_quantity$	φ_{10}
Distribuição Zipf	$t.age = t'.age$	φ_{11}
	$t.age = t'.age$ AND $t.salary = t'.salary$	φ_{12}

Fonte: Autoria própria (2025).

Por fim, ressalta-se que uma limitação deste trabalho é o uso exclusivo de predicados com operadores de igualdade. Até o momento, não identificamos estudos anteriores que abordem diferentes operadores. Embora existam pesquisas relacionadas em processamento de consulta (p. ex.: (Khayyat *et al.*, 2015)), a literatura ainda carece de trabalhos dedicados à estimativa de cardinalidade para operadores variados.

4.3 Algoritmos

Esta seção detalha a ampliação dos procedimentos dos algoritmos — permitindo a estimativa de diferentes tipos de predicados — e os resultados obtidos.

Simple Profile. Neste texto, referimos *Simple Profile* aos procedimentos de estimativa seminais introduzidos no System R, apresentados na Seção 2.1.2.3. A equação 4 fornece a fórmula tradicional para estimar o tamanho do *join* entre duas tabelas distintas. Como um *self-join* também constitui um *join*, derivamos essa fórmula para estimar os predicados descritos na seção anterior. Naturalmente, a fórmula original é aplicável apenas a predicados simples do tipo $t.X = t'.Y$.

Para predicados complexos, é possível adaptar a fórmula modificando o denominador, que representa a estimativa (ou valor exato, se disponível) do número de valores distintos do(s) atributo(s) participante(s) na cláusula de *join*. Para um predicado complexo do tipo $t.X_1 =$

$t'.Y_1 \wedge ... \wedge t.X_n = t'.Y_n$, podemos assumir como número de combinações possíveis na relação o valor $max(\prod_{i=1}^{n} V(X_i, R), \prod_{i=1}^{n} V(Y_i, R))$, ou seja, a combinação de valores possíveis para cada lado do *join* sobre n predicados. A equação 7 define a fórmula para predicados complexos.

$$\frac{n_R \times n_R}{max(\prod_{i=1}^{n} V(X_i, R), \prod_{i=1}^{n} V(Y_i, R))} \tag{7}$$

A seção anterior descreveu a família de distribuições adotada para os atributos dos conjuntos de dados sintéticos. Especificamente, a coluna `age` foi avaliada sob distribuições normal e uniforme (consultas φ_1 e φ_4). Adicionalmente, a coluna `l_quantity` do TPC-H (consulta φ_{10}) também apresenta uma distribuição uniforme.

Os resultados confirmam o comportamento esperado: em distribuições uniformes, a suposição tradicional de uniformidade produz estimativas precisas. Em contraste, conforme a distribuição se afasta da uniformidade e o número de predicados aumenta, o *Simple Profile* tende a subestimar a cardinalidade em algumas ordens de magnitude, como observado nas consultas $\varphi_7 - \varphi_9$, φ_{11} e φ_{12}.

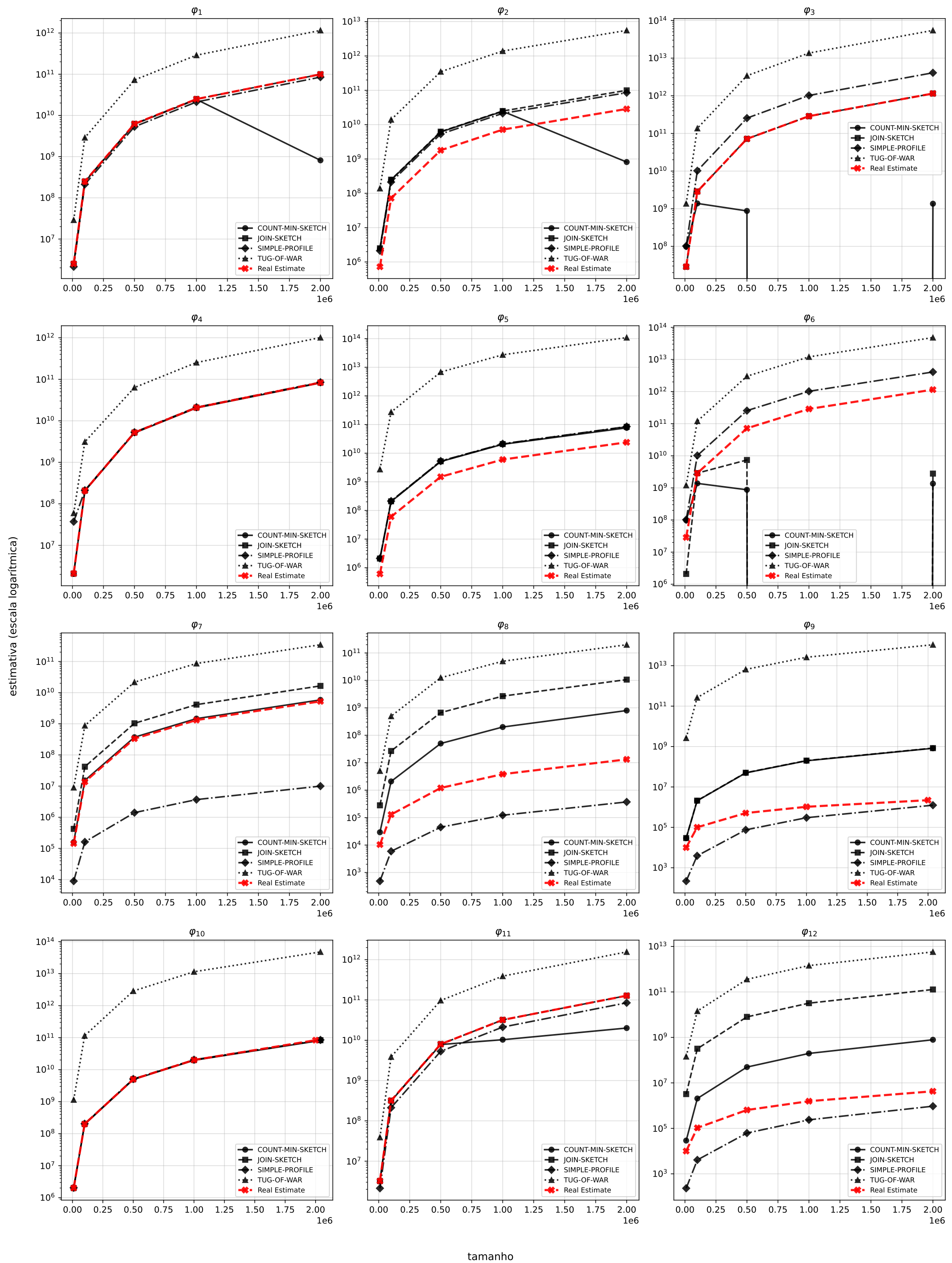
Relembre que a fórmula de estimativa requer o número de valores distintos para todos os atributos envolvidos no *join*. Esse valor varia entre sistemas; por exemplo, o DuckDB realiza essa tarefa por meio de uma estimativa com *sketches* HyperLogLog (Raasveldt; Mühleisen, 2019). Para este estudo, utilizamos o algoritmo da biblioteca XTensor[4] para obter o número de valores distintos. Denotamos que a implementação disponibilizada não é a mais eficiente em termos de complexidade de tempo — como alternativa, poderíamos constriuir um *hashset* com uma única varredura e contar o número de chaves inseridas na estrutura. As Figuras 9 e 10 apresentam, respectivamente, o tempo de construção e o tempo de estimativa das estruturas. Como esperado, o tempo de estimativa permanece praticamente estável com o aumento do tamanho dos dados, já que depende apenas do número de predicados na cláusula do *join* (Equação 7).

Tug-of-War. Conforme discutido na Seção 3.1.2.1, a finalidade do Tug-of-War consiste em estimar o segundo momento da frequência. Primordialmente, o algoritmo admite apenas a estimativa de *self-join* sob predicados simples de colunas homogêneas, isto é, $t.X = t'.X$. Entretanto, o procedimento pode ser estendido para predicados complexos de colunas homogêneas, da forma $t.X_1 = t'.X_1 \wedge \ ... \wedge \ t.X_n = t'.X_n$.

A modificação do algoritmo consiste em concatenar, para cada tupla, os valores dos atributos presentes no predicado complexo, preservando a ordem que aparecem. Então, o valor resultante (concatenado) é tratado como uma única entrada para a *sketch*, de modo análogo ao predicado simples. Conceitualmente, é possível assemelhar essa abordagem com índices compostos em SGBDs. No entanto, o método apresenta uma limitação significativa: para cada predicado, é necessário a construção de uma *sketch* distinta. Por exemplo, uma *sketch* cons-

[4] O procedimento do algoritmo ocorre em duas etapas: (i) ordenação dos valores; e (ii) deleção das duplicatas.

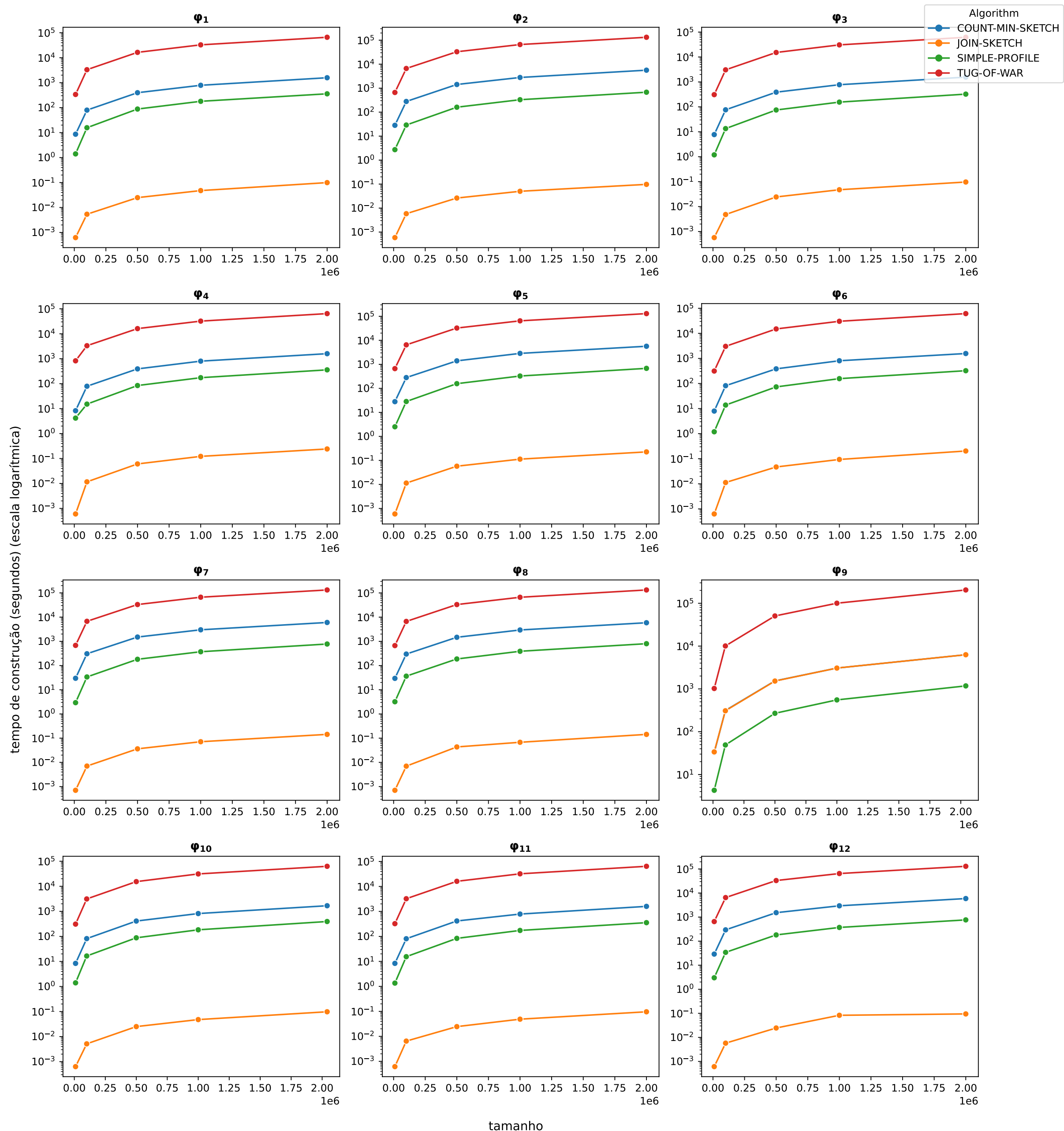
Figura 8 – Estimativas dos algoritmos sobre as consultas $\varphi_1 - \varphi_{12}$



Fonte: Autoria própria (2025).

truída para o predicado $t.X_1 = t'.X_1 \wedge t.X_2 = t'.X_2$ não pode ser reutilizada para o predicado $t.X_1 = t'.X_1$ (contrariamente de um índice composto).

Figura 9 – Tempo de construção das estruturas sobre as consultas $\varphi_1 - \varphi_{12}$

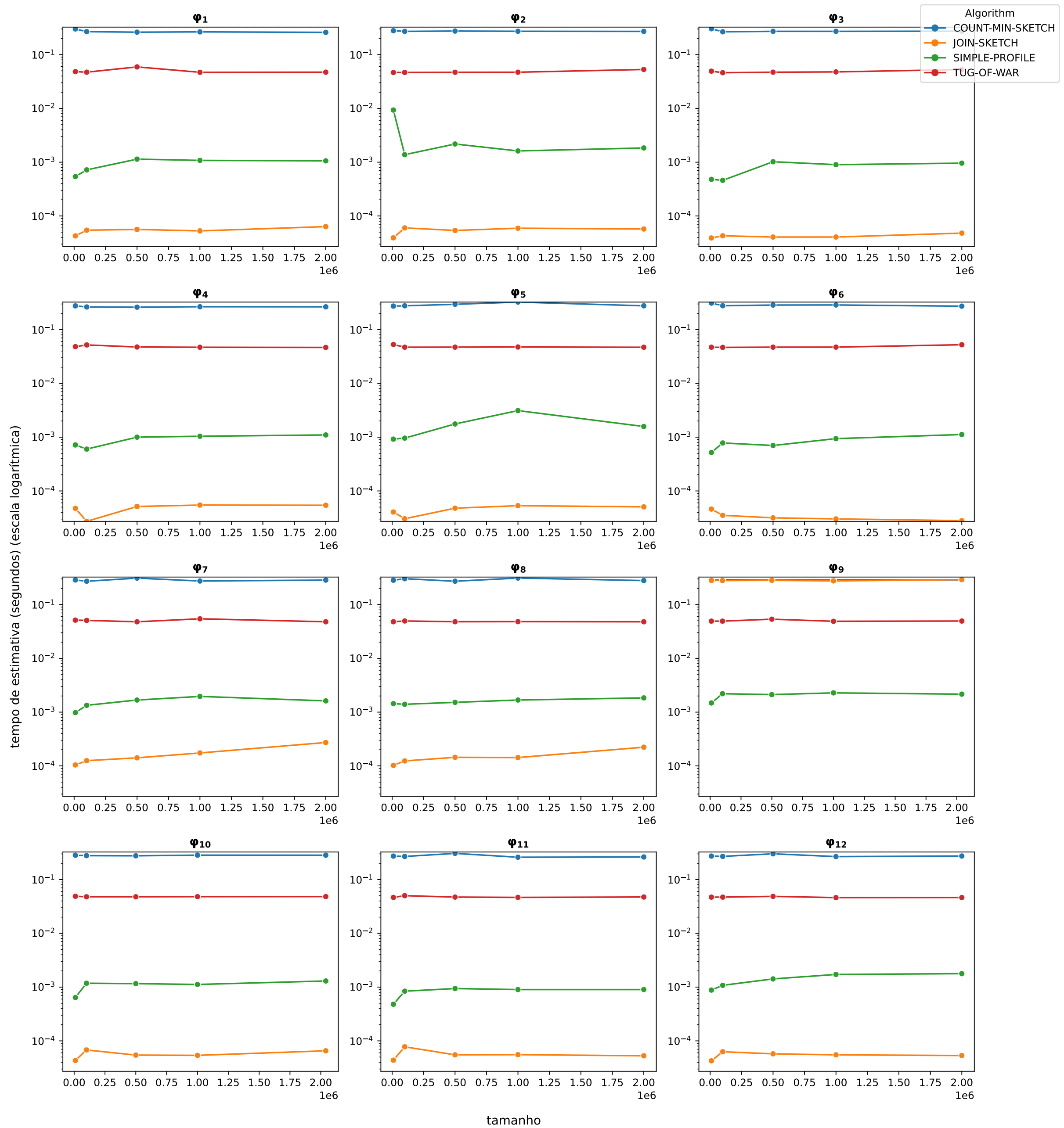


Fonte: Autoria própria (2025).

Utilizamos a técnica da mediana das médias para evitar a variância elevada decorrente do uso de um único contador (conforme descrito na Seção 3.1.2.1). Utilizamos cinco grupos com 512 contadores cada, totalizando aproximadamente 50 KB de memória alocada para a estrutura.

Alguns resultados obtidos foram atípicos: o *Simple Profile* produziu estimativas substancialmente mais precisas — em alguns casos, por várias ordens de magnitude — do que o *Tug-of-War* em todas as consultas. Esse comportamento é inesperado, pois a *sketch* deveria apresentar menor sensibilidade à assimetria da distribuição do que o *Simple Profile*, especialmente para distribuições não uniformes. Além disso, um resultado notável foi o *Count-min*

Figura 10 – Tempo de estimativa dos algoritmos sobre as consultas $\varphi_1 - \varphi_{12}$



Fonte: Autoria própria (2025).

Sketch apresentar desempenho significativamente superior ao *Tug-of-War*, embora, conforme descrito pelos próprios autores, o *Count-min Sketch* talvez não seja eficiente para estimativas de normas em fluxos de dados (Cormode; Muthukrishnan, 2005) — em outras palavras, estimativa de produto interno. Por fim, os valores dos *q-errors* obtidos (Tabela 5) evidenciam a elevada variância apresentada pelo *Tug-of-War*.

Quanto ao tempo de construção da estrutura, com exceção da consulta φ_9, o *Tug-of-War* apresentou o pior desempenho em todos os cenários. Acreditamos que esse desempenho decorre da função *hash* utilizada na implementação selecionada[5]. Por fim, como esperado, o

[5] Adaptado de: https://github.com/JoinSketch/JoinSketch

tempo de estimativa manteve-se constante, uma vez que o procedimento se limita apenas a computar as médias de cada grupo e selecionar a mediana.

Count-min Sketch. O projeto original do *Count-min Sketch* permite a estimativa de predicados simples com colunas homogêneas e heterogêneas, ou seja, $t.X = t'.Y$, onde X, Y são atributos da relação. Uma vantagem relevante é que, no caso de predicados homogêneos, é possível utilizar apenas uma *sketch* para estimativa de *self-join*. Nesse contexto, como o *join* ocorre na mesma tabela utilizando o mesmo atributo, basta construir a *sketch* sobre esse atributo, somar os quadrados das células de cada linha, e, selecionar a linha de valor mínimo. Além disso, adotamos a mesma abordagem do *Tug-of-War* para estender a aplicabilidade da *sketch* para predicados complexos. Dessa forma, o *Count-min Sketch* foi capaz de suportar todos os tipos de predicados considerados: simples ou complexo, de coluna(s) homogênea(s) ou heterogênea(s).

Os parâmetros selecionados para a *sketch* foram 24 e 5000, correspondentes a d e w, respectivamente, resultando em aproximadamente 468 KB de memória alocada.

O *Count-min Sketch* teve estimativas significativamente melhores que o *Tug-of-War* em diversas consultas. Além disso, para dados reais (IMDb), o *Count-min Sketch* apresentou o melhor desempenho na consulta φ_7. A Figura 8 apresenta os resultados.

No entanto, o *Count-min Sketch* foi a estrutura que mais apresentou problemas. Diversos resultados inesperados foram observados nas consultas executadas nos conjuntos de dados sintéticos. As consultas φ_{1-3}, φ_6 e φ_{11}, por exemplo, apresentaram uma redução abundante na estimativa no aumento de tamanho. Indicamos possíveis causas para essas ocorrências na Seção 4.4.

O tempo de construção da *sketch* (Figura 9) foi consistentemente mais rápido do que o *Tug-of-War* em algumas ordens de magnitude, porém mais lento que o *JoinSketch* e o *Simple Profile* na maioria dos cenários (o *Count-min Sketch* e o *JoinSketch* tiveram praticamente o mesmo tempo de construção na consulta φ_9).

Por fim, novamente o tempo de estimativa (Figura 10) permaneceu estável com o aumento do tamanho dos dados. Essa característica é esperada pois a estimativa depende do tamanho (fixo) da *sketch*, e não do volume total de dados.

JoinSketch. As mesmas adaptações realizadas no *Count-min Sketch* foram realizadas para o *JoinSketch*, permitindo o suporte para todos os tipos de predicados. A implementação original da *sketch*[6] permite configurar o tamanho da estrutura por meio de um parâmetro via linha de comando. Seguindo a mesma abordagem de (Wang *et al.*, 2023), utilizamos 80 KB para a estrutura em nossos experimentos.

O JoinSketch apresentou o melhor desempenho geral, pois obteve as estimativas mais precisas (com exceção de φ_8 e φ_{12}) e obteve os menores tempos de construção e estimativa, superando o segundo melhor resultado por várias ordens de magnitude em todos os experimentos.

Os resultados de estimativa são consistentes com os achados de (Wang *et al.*, 2023). Especificamente, o *Count–min Sketch* requer substancialmente mais memória do que o *JoinS-*

[6] https://github.com/JoinSketch/JoinSketch

ketch para atingir uma precisão comparável na estimativa do produto interno. Em nossos experimentos, essa tendência se manteve em quase todos os predicados: o *Count–min Sketch* utilizou quase seis vezes mais memória que o *JoinSketch*, mesmo produzindo estimativas menos precisas.

Tabela 5 – Média do *q-error* sobre os tamanhos: 10K, 100K, 500K, 1M e 2M

	Simple Profile	Tug-of-War	Count-min Sketch	JoinSketch
φ_1	1.19	11.15	24.80	**1.0**
φ_2	**2.90**	186.22	9.67	3.47
φ_3	3.48	45.64	183.09	**1.00**
φ_4	4.28	15.36	**1.00**	**1.00**
φ_5	3.46	4360.27	**3.42**	**3.42**
φ_6	**3.47**	40.21	183.15	87.34
φ_7	249.41	62.01	**1.13**	3.07
φ_8	**27.98**	8216.62	35.26	458.40
φ_9	1.70	1.73	**1.38**	**1.38**
φ_{10}	**1.00**	549.43	**1.00**	**1.00**
φ_{11}	1.53	11.69	2.45	**1.00**
φ_{12}	18.45	571276.72	83.65	13224.21

Fonte: Autoria própria (2025).

4.4 Limitações

Este estudo apresenta limitações — algumas ainda sem explicações — como ilustrado na Figura 8. As consultas φ_1, φ_2, φ_3, φ_6 e φ_{11} exibem uma queda acentuada nas estimativas do *Count-min Sketch* entre 500 mil e 2 milhões de registros. Esse comportamento é inesperado, dado que o *Count-min Sketch* é um estimador enviesado que sempre superestima os valores reais. O *JoinSketch* apresentou comportamento similar apenas na consulta φ_6.

Outro resultado inesperado foram as estimativas obtidas da consulta φ_2: o *Simple Profile* teve o melhor desempenho. Esse resultado é atípico pois o *Simple Profile* realiza a suposição de uniformidade; e os valores dos atributos do predicado φ_2 formam uma distribuição normal. De forma análoga, consideramos os resultados de φ_{11} e φ_{12} inesperados.

Nossa hipótese é que essas discrepâncias estejam associadas à geração dos dados sintéticos — uma vez que as consultas φ_7 e φ_8, obtidas a partir do mundo real, produziram os resultados mais precisos. Por fim, as implementações das *sketches* por diferentes autores utilizadas neste trabalho pode ter causado diferença nos resultados observados.

5 CONCLUSÕES

Este trabalho teve como objetivo principal investigar experimentalmente o desempenho de diferentes técnicas de estimativa de cardinalidade aplicadas a consultas de *self-join*, uma área ainda pouco explorada na literatura de otimização de consultas. Demonstramos que, em conjuntos de dados do mundo real que apresentam não uniformidade e assimetria, métodos baseados em *sketches*, como o *JoinSketch*, superam consideravelmente o modelo tradicional baseado nas suposições de distribuição. Por fim, uma direção para trabalhos futuros é investigar técnicas alternativas para estimar predicados que envolvam intervalos (ou seja, operadores de comparação de desigualdades).

REFERÊNCIAS

ACHARYA, S. *et al.* Join synopses for approximate query answering. *In*: PROCEEDINGS OF THE 1999 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA. 1999, New York, NY, USA. **Anais [...]** New York, NY, USA: Association for Computing Machinery, 1999. p. 275–286. ISBN 1581130848. Disponível em: https://doi.org/10.1145/304182.304207.

ALON, N. *et al.* Tracking join and self-join sizes in limited storage. *In*: PROCEEDINGS OF THE EIGHTEENTH ACM SIGMOD-SIGACT-SIGART SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS. 1999, New York, NY, USA. **Anais [...]** New York, NY, USA: Association for Computing Machinery, 1999. p. 10–20. ISBN 1581130627. Disponível em: https://doi.org/10.1145/303976.303978.

CORMODE, G. *et al.* Synopses for massive data: Samples, histograms, wavelets, sketches. **Found. Trends Databases**, Now Publishers Inc. Hanover, MA, USA v. 4, n. 1–3, p. 1–294, jan. 2012. ISSN 1931-7883. Disponível em: https://doi.org/10.1561/1900000004.

CORMODE, G.; MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. **Journal of Algorithms**, v. 55, n. 1, p. 58–75, 2005. ISSN 0196-6774. Disponível em: https://www.sciencedirect.com/science/article/pii/S0196677403001913.

DALIRI, M. *et al.* Sampling methods for inner product sketching. **Proc. VLDB Endow.**, VLDB Endowment v. 17, n. 9, p. 2185–2197, maio 2024. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/3665844.3665850.

DING, B.; NARASAYYA, V.; CHAUDHURI, S. **Extensible Query Optimizers in Practice**. Foundations and Trends® in Databases, 2024. v. 14. Disponível em: https://www.microsoft.com/en-us/research/publication/extensible-query-optimizers-in-practice/.

FLAJOLET, P. *et al.* Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. **Discrete mathematics & theoretical computer science**, Episciences. org, n. Proceedings„ 2007.

FREITAG, M. J.; NEUMANN, T. Every row counts: Combining sketches and sampling for accurate group-by result estimates. *In*: CONFERENCE ON INNOVATIVE DATA SYSTEMS RESEARCH. 2019. **Anais [...]** [*s.n.*], 2019. Disponível em: https://api.semanticscholar.org/CorpusID:53654435.

GANGULY, S.; GAROFALAKIS, M.; RASTOGI, R. Processing data-stream join aggregates using skimmed sketches. *In*: BERTINO, E. *et al.* (Ed.). ADVANCES IN DATABASE TECHNOLOGY - EDBT 2004. 2004, Berlin, Heidelberg. **Anais [...]** Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 569–586. ISBN 978-3-540-24741-8.

HARMOUCH, H.; NAUMANN, F. Cardinality estimation: an experimental survey. **Proc. VLDB Endow.**, VLDB Endowment v. 11, n. 4, p. 499–512, dez. 2017. ISSN 2150-8097. Disponível em: https://doi.org/10.1145/3186728.3164145.

IOANNIDIS, Y. The history of histograms (abridged). *In*: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES - VOLUME 29. 2003. **Anais [...]** [*S.l.*]: VLDB Endowment, 2003. p. 19–30. ISBN 0127224424.

KHAYYAT, Z. *et al.* Lightning fast and space efficient inequality joins. **Proc. VLDB Endow.**, VLDB Endowment v. 8, n. 13, p. 2074–2085, set. 2015. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/2831360.2831362.

LEE, K. *et al.* Analyzing the impact of cardinality estimation on execution plans in microsoft sql server. **Proc. VLDB Endow.**, VLDB Endowment v. 16, n. 11, p. 2871–2883, jul. 2023. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/3611479.3611494.

LEIS, V. *et al.* How good are query optimizers, really? **Proc. VLDB Endow.**, VLDB Endowment v. 9, n. 3, p. 204–215, nov. 2015. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/2850583.2850594.

LEIS, V. *et al.* Query optimization through the looking glass, and what we found running the join order benchmark. **The VLDB Journal**, Springer-Verlag Berlin, Heidelberg v. 27, n. 5, p. 643–668, out. 2018. ISSN 1066-8888. Disponível em: https://doi.org/10.1007/s00778-017-0480-7.

LIU, Z. *et al.* Rapidash: Efficient detection of constraint violations. **Proc. VLDB Endow.**, VLDB Endowment v. 17, n. 8, p. 2009–2021, abr. 2024. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/3659437.3659454.

MOERKOTTE, G. Cardinality estimation for having-clauses. **Proc. VLDB Endow.**, VLDB Endowment v. 18, n. 1, p. 28–41, set. 2024. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/3696435.3696438.

MOERKOTTE, G.; NEUMANN, T.; STEIDL, G. Preventing bad plans by bounding the impact of cardinality estimation errors. **Proc. VLDB Endow.**, VLDB Endowment v. 2, n. 1, p. 982–993, ago. 2009. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/1687627.1687738.

PENA, E. H. M.; ALMEIDA, E. C. de; NAUMANN, F. Fast detection of denial constraint violations. **Proc. VLDB Endow.**, VLDB Endowment v. 15, n. 4, p. 859–871, dez. 2021. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/3503585.3503595.

RAASVELDT, M.; MüHLEISEN, H. Duckdb: an embeddable analytical database. *In*: PROCEEDINGS OF THE 2019 INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA. 2019, New York, NY, USA. **Anais [...]** New York, NY, USA: Association for Computing Machinery, 2019. p. 1981–1984. ISBN 9781450356435. Disponível em: https://doi.org/10.1145/3299869.3320212.

RAFIEI, D.; DENG, F. Similarity join and similarity self-join size estimation in a streaming environment. **IEEE Transactions on Knowledge and Data Engineering**, Institute of Electrical and Electronics Engineers (IEEE) v. 32, n. 4, p. 768–781, abr. 2020. ISSN 2326-3865. Disponível em: http://dx.doi.org/10.1109/TKDE.2019.2893175.

REPAS, D. *et al.* **Selectivity Estimation of Inequality Joins In Databases**, . 2022. Disponível em: https://arxiv.org/abs/2206.07396.

SELINGER, P. G. *et al.* Access path selection in a relational database management system. *In*: PROCEEDINGS OF THE 1979 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA. 1979, New York, NY, USA. **Anais [...]** New York, NY, USA: Association for Computing Machinery, 1979. p. 23–34. ISBN 089791001X. Disponível em: https://doi.org/10.1145/582095.582099.

THANOPOULOU, A.; CARREIRA, P.; GALHARDAS, H. Benchmarking with tpc-h on off-the-shelf hardware - an experiments report. *In*: INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS. 2012. **Anais [...]** [*s.n.*], 2012. Disponível em: https://api.semanticscholar.org/CorpusID:18615645.

WANG, F. *et al.* Joinsketch: A sketch algorithm for accurate and unbiased inner-product estimation. **Proc. ACM Manag. Data**, Association for Computing Machinery New York, NY, USA v. 1, n. 1„ maio 2023. Disponível em: https://doi.org/10.1145/3588935.