

Análise Experimental da Escalabilidade e Elasticidade de Bibliotecas de Análise de Dados em Workflows com Alto Custo Computacional

Gabriel R. B. Cirillo¹, Guilherme Galante¹

¹Universidade Estadual do Oeste do Paraná (UNIOESTE)
Centro de Ciências Exatas e Tecnológicas, Cascavel, PR, Brasil

{gabriel.cirillo, guilherme.galante}@unioeste.br

Abstract. *This work experimentally analyzes the scalability and elasticity of Python libraries for data analysis Pandas, Polars, Dask, DuckDB, and PySpark under different data volumes and resource configurations. The results show that none of the evaluated libraries presented full scalability in the tested scenarios: all exhibited sublinear speedup with saturation, with efficiency decreasing as cores were added; DuckDB achieved the lowest absolute execution times and Polars the highest average speedup and efficiency, while PySpark, Pandas, and Dask showed limited gains. Regarding elasticity, the libraries proved horizontally elastic, with PySpark making better use of the dynamically allocated resources (closer to the configured ceiling, with more stable scaling) and Dask achieving the lowest execution times, though a larger worker ceiling reduced execution time for both.*

Resumo. *Este trabalho analisa experimentalmente a escalabilidade e a elasticidade de bibliotecas Python para análise de dados a Pandas, a Polars, a Dask, a DuckDB e a PySpark em diferentes volumes e configurações de recursos. Os resultados mostram que nenhuma das bibliotecas apresentou escalabilidade plena nos cenários testados: todas exibiram speedup sublinear com saturação, com eficiência decrescente conforme novos núcleos eram adicionados; a DuckDB obteve os menores tempos absolutos de execução e a Polars as maiores médias de speedup e eficiência, enquanto a PySpark, a Pandas e a Dask apresentaram ganhos mais limitados. Na elasticidade, as bibliotecas mostraram-se elásticas horizontalmente, com a PySpark aproveitando melhor os recursos alocados dinamicamente (mais próximo do teto configurado e com escalonamento mais estável) e a Dask alcançando os menores tempos de execução, embora, em ambos os casos, um teto maior de workers tenha reduzido o tempo total.*

1. Introdução

O avanço tecnológico tem impulsionado a geração e o uso de grandes volumes de dados provenientes de fontes diversas, especialmente com tecnologias como IoT, computação em nuvem e ferramentas de *big data* [Balusamy et al. 2021]. Entretanto, a heterogeneidade e a variedade das fontes dificultam sua coleta e integração, principalmente devido às dimensões de volume, variedade e velocidade dos dados [Putrama and Martinek 2024].

Devido a isso, cientistas de dados dedicam tempo e recursos consideráveis para realizar a preparação de dados. Normalmente, esse processo fundamental envolve múltiplas operações combinadas em um *pipeline* com o objetivo de explorar, limpar e transformar dados brutos em conjuntos de dados selecionados. Nesse contexto, bibliotecas especializadas oferecem recursos para automatizar essas tarefas e otimizar a manipulação e o processamento de dados em larga escala, favorecendo soluções mais eficientes e escaláveis [Mozzillo et al. 2025].

A biblioteca Pandas destaca-se por sua popularidade e ampla adoção e por simplificar a manipulação de dados em Python [McKinney 2023]. Entretanto, sua arquitetura *single thread*, a ausência de suporte nativo a *clusters* e de mecanismos de otimização de memória limitam seu desempenho no processamento de grandes volumes de dados [Mozzillo et al. 2025]. Consequentemente, à medida que os dados crescem, podem ocorrer gargalos de desempenho e aumento significativo do tempo de processamento [Petersohn 2018]. Essa limitação é especialmente crítica em aplicações de larga escala e tempo real, demandando alternativas que explorem paralelismo e computação distribuída. Além disso, a variabilidade da demanda em ambientes modernos exige não apenas escalabilidade, mas também elasticidade, permitindo ajustar dinamicamente os recursos computacionais.

Neste trabalho busca-se, por meio da comparação sistemática das bibliotecas Pandas, Polars, DuckDB, Dask e PySpark, identificar quais soluções melhor equilibram desempenho absoluto com capacidade de adaptação a cenários reais, onde a disponibilidade de recursos computacionais pode flutuar significativamente. As contribuições deste trabalho são: (i) a comparação experimental de cinco bibliotecas Python em três volumes de dados (SF 1, 10 e 50) e quatro configurações de núcleos (1, 8, 16 e 32), com análise estatística baseada em intervalos de confiança; (ii) a avaliação da elasticidade horizontal das bibliotecas distribuídas (Dask e PySpark) em ambiente Kubernetes com *Horizontal Pod Autoscaler* (HPA); e (iii) a análise conjunta de escalabilidade e elasticidade, distinguindo os dois regimes de provisionamento de recursos.

2. Fundamentação Teórica

Esta seção apresenta os conceitos fundamentais que embasam este trabalho, bem como as ferramentas utilizadas.

2.1. Bibliotecas

A seguir, descrevem-se as bibliotecas *Pandas*, *DuckDB*, *Polars*, *PySpark* e *Dask*, que são analisadas ao longo desse trabalho.

A biblioteca **Pandas** para Python destaca-se pela ampla adoção em tarefas de manipulação e análise de dados estruturados. Entretanto, opera em modo *single-thread*, executando operações sequencialmente em um único núcleo, o que limita seu desempenho em cenários de alto volume de dados [McKinney 2023]. Suas principais estruturas, *Series* e *DataFrame*, permitem organizar e processar dados tabulares de forma intuitiva.

O **DuckDB**¹ é um sistema de banco de dados analítico embarcado (*in-process*), otimizado para cargas OLAP. Opera em modo *multithread*, paralelizando consultas SQL

¹Site oficial do DuckDB. Disponível em: <https://duckdb.org>

diretamente sobre arquivos CSV e Parquet. Sua arquitetura utiliza processamento vetorizado e armazenamento colunar, resultando em execução eficiente. É capaz de operar tanto em memória quanto em disco, mantendo desempenho consistente mesmo em *datasets* que excedem a memória disponível.

A **Polars** é uma biblioteca de processamento de dados implementada em *Rust*, desenvolvida como alternativa de alto desempenho à Pandas. Opera em modo *multithread*, distribuindo operações entre múltiplos núcleos, e utiliza *lazy evaluation*, permitindo otimizar e executar operações apenas quando o resultado é requisitado [Nahrstedt et al. 2024]. Além disso, adota armazenamento colunar e estruturas de dados imutáveis, contribuindo para um gerenciamento eficiente da memória e para a redução do tempo de execução.

O **PySpark** é a *interface* Python para o *Apache Spark*, plataforma de processamento distribuído voltada ao tratamento de grandes volumes de dados [Drabas and Lee 2017]. Entre as bibliotecas analisadas, é a que opera em modelo genuinamente distribuído, suportando implantação em *clusters* com múltiplos nós. Sua arquitetura utiliza um *driver* central para coordenar *workers* distribuídos e *Resilient Distributed Datasets* (RDDs), que permitem operações tolerantes a falhas [Zaharia et al. 2012].

A **Dask** é uma biblioteca Python desenvolvida para superar a limitação da Pandas de operar restrita à memória de uma única máquina [Rocklin 2015]. Suporta implantação em *clusters* e também pode operar em modo *multithread* localmente, funcionando como alternativa intermediária entre bibliotecas locais e *frameworks* distribuídos. Seu funcionamento baseia-se em grafos de tarefas acíclicos dirigidos e utiliza *lazy evaluation*, com APIs compatíveis com Pandas e NumPy [Dask Development Team 2026].

A tabela 1 sintetiza as principais características de cada biblioteca. A Pandas não oferece *multithreading* nem *lazy evaluation*, enquanto a Polars e a DuckDB suportam ambos. A PySpark e a Dask destacam-se pelo suporte à implantação em *cluster*.

Tabela 1. Características das bibliotecas analisadas.					
Característica	Pandas	DuckDB	Polars	PySpark	Dask
Multithreading		✓	✓	✓	✓
Lazy evaluation		✓	✓	✓	✓
Deploy em cluster				✓	✓
Linguagem nativa	Python	C++	Rust	Scala	Python
Licença	BSD-3	MIT	MIT	Apache 2.0	BSD-3
Versão considerada	2.2.2	0.10.1	0.20.21	3.5.1	2024.4.1

2.2. Polars Decision Support (PDS)

O *Polars Decision Support* (PDS)² é um *benchmark* desenvolvido em Python pela equipe da biblioteca Polars que adapta o TPC-H³ para execução sobre bibliotecas com estruturas de *dataframes*: Polars, Modin, Dask, PySpark, Pandas e DuckDB. O TPC-H simula um ambiente de negócios com consultas analíticas complexas sobre um esquema fictício de vendas composto por oito tabelas [Transaction Processing Performance Council 2023]. O

²Repositório disponível em: <https://github.com/pola-rs/polars-benchmark>

³Site oficial do TPC. Disponível em: <https://www.tpc.org/tpch/>

volume de dados é escalável por meio do fator de escala (*Scale Factor*, SF), em que SF 1 corresponde a aproximadamente 1 GB e SF 50 a cerca de 50 GB no formato CSV. São definidas 22 consultas padronizadas, projetadas para testar operações como junções, agregações, ordenações e filtros complexos.

Cabe destacar que a especificação oficial do TPC-H impõe requisitos rigorosos para validação, como sessões simultâneas e a métrica *QphH* [Transaction Processing Performance Council 2023], que estão além do escopo deste trabalho. Por essa razão, adotou-se o PDS como alternativa, preservando as 22 consultas originais e o esquema de dados do TPC-H, mas adequando a execução ao contexto de análise comparativa entre bibliotecas Python.

2.3. Escalabilidade, Elasticidade e Kubernetes

A escalabilidade pode ser definida como a capacidade de um sistema manter níveis adequados de desempenho à medida que a demanda aumenta [Lehrig et al. 2015], seja pela adição de recursos a um único nó (vertical) ou pela adição de novos nós (horizontal). A elasticidade vai além, tratando da capacidade de ajustar dinamicamente os recursos provisionados em resposta a variações na demanda, sem intervenção manual [Lehrig et al. 2015], frequentemente automatizada por mecanismos de *autoscaling* [Herbst et al. 2013].

O *Kubernetes* é uma plataforma de código aberto para orquestração de contêineres [Medel et al. 2024]. As aplicações são executadas em *pods*, com recursos de CPU e memória delimitados por *request* e *limit*⁴. O *Horizontal Pod Autoscaler* (HPA) monitora métricas como CPU e memória e ajusta o número de réplicas: quando a utilização ultrapassa o *threshold*, executa um *scale-up*; quando diminui, realiza o *scale-down* [Kubernetes 2026]. No cenário de elasticidade deste trabalho, a orquestração foi realizada pelo Kubernetes, com o HPA configurado para escalar o número de *workers* das bibliotecas Dask e PySpark, conforme apresentado na Seção 4.

3. Trabalhos Relacionados

Diversos estudos comparam bibliotecas de análise de dados, porém de forma pontual e em escalas fixas. [Petersohn 2018] compara a Pandas e a Modin, ambas projetadas para um único nó, mostrando que ferramentas capazes de lidar com grandes conjuntos de dados exigem conhecimentos de computação distribuída, como o particionamento dos dados. [Piñeiro 2022] avalia a Pandas e a Polars no processamento de grandes conjuntos de dados, concluindo que a Polars apresenta desempenho superior por acessar a memória de forma mais eficiente e reduzir cópias redundantes. [Gu et al. 2022] destaca que, embora amplamente utilizada, a Pandas não é escalável: quando os dados excedem a memória, é necessário migrar para sistemas que suportem dados em larga escala. [Mozzillo et al. 2025] avalia bibliotecas de *dataframe* para preparação de dados em uma única máquina, cobrindo Pandas, Polars, DuckDB, Dask e PySpark com o benchmark TPC-H, e conclui que nenhuma domina em todos os cenários. O presente trabalho expande essa comparação ao introduzir configurações com múltiplos núcleos (escalabilidade) e orquestração distribuída via Kubernetes (elasticidade), dimensões não contempladas por esse estudo.

⁴De acordo com o site oficial do Kubernetes disponível em: <https://kubernetes.io/docs>.

No campo da elasticidade em computação em nuvem, [Herbst et al. 2013] propõe métricas quantitativas para avaliar a elasticidade de plataformas, como tempo de reação e precisão do provisionamento, destacando a ausência de métricas padronizadas que permitam comparações objetivas. [Lehrig et al. 2015] complementa essa visão com uma revisão sistemática de definições para escalabilidade, elasticidade e eficiência em nuvem. Já [Medel et al. 2024] analisa o comportamento do Kubernetes e do HPA na orquestração de aplicações distribuídas, avaliando o impacto de diferentes políticas de *autoscaling* sobre o desempenho e o uso de recursos.

Embora esses estudos ofereçam uma base sólida, a maioria se restringe à comparação entre pares de bibliotecas ou à avaliação da escalabilidade em isolamento, sem investigar conjuntamente a elasticidade horizontal. Este trabalho avança nessa direção ao avaliar simultaneamente cinco bibliotecas Python sob dois regimes distintos de provisionamento de recursos, combinando escalabilidade vertical (variação de núcleos) e elasticidade horizontal (alocação dinâmica de *workers* via HPA) em um mesmo ambiente experimental.

4. Avaliação Experimental

4.1. Ambiente Computacional

Os experimentos foram executados em uma máquina virtual hospedada em um servidor Dell PowerEdge R7625, equipado com dois processadores AMD EPYC 9334 (32 núcleos/64 *threads* a 2,70 GHz), totalizando 64 núcleos físicos e 128 *threads*, 256 GB de memória RAM e aproximadamente 53 TB de armazenamento NVMe SSD U.2 Gen4. A máquina virtual foi configurada com 32 núcleos de processamento (vCPUs), 60 GB de memória RAM e sistema operacional Ubuntu 24.04 LTS. Cabe ressaltar que a execução em uma única máquina virtual, embora garanta controle experimental, limita a validade externa dos resultados do cenário distribuído, que em ambientes de produção operaria com múltiplos nós físicos.

4.2. Cenários de Teste

Na avaliação de escalabilidade, os experimentos foram executados em contêineres *Docker* gerenciados pelo *Docker Compose*. Para garantir a reprodutibilidade do ambiente experimental, foi criada uma imagem *Docker* baseada no Ubuntu 22.04 LTS, contendo todas as bibliotecas, dependências e ferramentas necessárias para a execução do benchmark PDS. Essa imagem foi construída para executar a versão de um *fork* desenvolvido pelos autores sobre a implementação oficial do PDS, estendido com a implementação das consultas ausentes do benchmark TPC-H, uma vez que, na versão original, as bibliotecas Polars, Pandas e Dask disponibilizam apenas 7 das 22 consultas especificadas pelo benchmark. As 22 consultas do *fork* foram validadas pela comparação dos resultados com a saída de referência produzida pelo TPC-H em SQL.

A partir dessa imagem, o *Docker Compose* foi utilizado para instanciar quatro configurações de contêineres, diferenciadas exclusivamente pela quantidade de núcleos de processamento alocados (1, 8, 16 e 32 núcleos), executadas de forma sequencial. Em todas as configurações, a memória RAM disponível foi mantida fixa em 60 GB, de modo a isolar o impacto da variação dos recursos computacionais sobre o desempenho das bibliotecas. Para mitigar efeitos de *cache*, cada consulta foi executada em um processo

independente, e a ordem de execução das 22 consultas foi mantida fixa em todas as baterias.

Na avaliação de elasticidade, os contêineres foram orquestrados por meio do *Kubernetes*, também implantado na mesma máquina virtual. Para esse cenário, foi desenvolvida, sobre o *fork* do PDS, a implementação necessária para suportar a execução distribuída do benchmark. Entretanto, apenas as bibliotecas Dask e PySpark ofereciam suporte à execução distribuída, razão pela qual somente essas bibliotecas foram consideradas nessa etapa dos experimentos. Ressalta-se que a elasticidade horizontal resulta da combinação entre o mecanismo de *autoscaling* do Kubernetes (HPA), responsável pelo provisionamento e remoção de pods, e a capacidade das bibliotecas de incorporar novos *workers* dinamicamente durante a execução, sem interrupção do processamento em curso.

Em todos os experimentos foi utilizada a linguagem Python 3.11. Na avaliação de escalabilidade foram analisadas as bibliotecas Pandas, Polars, Dask, DuckDB e PySpark, enquanto na avaliação de elasticidade foram avaliadas apenas Dask e PySpark. As versões das bibliotecas empregadas são apresentadas na tabela 1.

As métricas de desempenho do cenário de escalabilidade foram coletadas por meio de uma ferramenta de monitoramento desenvolvida pelos autores deste trabalho, que coletaram estatísticas de memória e processador com um intervalo de 500 milissegundos. Na avaliação de elasticidade, utilizou-se o *Prometheus*, em conjunto com o *Pushgateway*, para a coleta e centralização das métricas dos contêineres e da infraestrutura orquestrada pelo *Kubernetes*⁵ com um intervalo maior para o cenário de coleta, com o valor de 15 segundos.

As tabelas usadas em ambos os cenários foram armazenadas em arquivos no formato *Parquet*. A escolha desse formato deve-se ao seu armazenamento colunar e aos mecanismos de compressão, que reduzem significativamente o espaço em disco em relação ao tamanho nominal (1, 10 e 50) de cada fator de escala (SF) apresentado na tabela 2. Dessa forma, em Parquet, SF1 tem tamanho aproximado de 255 MB, SF10 2,6 GB e SF50 15,3 GB. Os valores originais 1, 10 e 50 referem-se aos arquivos no formato CSV.

Tabela 2. Número de linhas e tamanho dos arquivos Parquet por fator de escala TPC-H

Tabela	SF 1		SF 10		SF 50	
	Linhas	Tamanho	Linhas	Tamanho	Linhas	Tamanho
lineitem	6.001.215	176 MB	59.986.052	1,8 GB	300.005.811	11 GB
orders	1.500.000	38 MB	15.000.000	393 MB	75.000.000	2,1 GB
partsupp	800.000	29 MB	8.000.000	293 MB	40.000.000	1,5 GB
customer	150.000	8,0 MB	1.500.000	80 MB	7.500.000	400 MB
part	200.000	4,1 MB	2.000.000	40 MB	10.000.000	217 MB
supplier	10.000	532 KB	100.000	5,1 MB	500.000	26 MB
nation	25	4 KB	25	4 KB	25	4 KB
region	5	4 KB	5	4 KB	5	4 KB
Total	8.661.252	≈ 255 MB	86.586.089	≈ 2,6 GB	433.005.848	≈ 15,3 GB

Para avaliar a escalabilidade, adotou-se o plano de execução apresentado na tabela 3, contemplando a execução de todas as 22 consultas de cada biblioteca por, no

⁵Disponível em : <https://kubernetes.io/docs>

mínimo, 30 vezes, considerando três escalas (SF 1, 10 e 50) e quatro configurações de contêineres.

Tabela 3. Plano de execução do experimento

Bibliotecas	Queries	Escalas	Núcleos
Pandas, Polars, PySpark, Dask, DuckDB	Q1–Q22	SF 1/ SF 10 / SF 50	1
Pandas, Polars, PySpark, Dask, DuckDB	Q1–Q22	SF 1/ SF 10 / SF 50	8
Pandas, Polars, PySpark, Dask, DuckDB	Q1–Q22	SF 1/ SF 10 / SF 50	16
Pandas, Polars, PySpark, Dask, DuckDB	Q1–Q22	SF 1/ SF 10 / SF 50	32

A tabela 4 apresenta o plano de execução adotado na avaliação da elasticidade. Para cada uma das 22 *queries*, foram realizadas no mínimo 30 execuções (o número exato de execuções n é apresentado na tabela de resultados), considerando os fatores de escala 10 e 50. A escala 1 foi descartada, pois os resultados da avaliação de escalabilidade já indicam baixos valores de *speedup* e eficiência. Além disso, nessa escala, os tempos de sincronização da infraestrutura distribuída são superiores ao tempo médio de execução da maioria das consultas, comprometendo a coleta precisa das métricas de desempenho.

Tabela 4. Plano de execução do experimento de elasticidade

Bibliotecas	Consultas	Cenários	Núcleos	Memória	Workers	Escalas	Threshold
Dask, PySpark	Q1–Q22	A (2 fixos)	2	4–8 GB	2 (fixo)	SF 10 e SF 50	70% CPU, 80% Memória
Dask, PySpark	Q1–Q22	B (HPA 2→4)	2–4	4–8 GB	2–4	SF 10 e SF 50	70% CPU, 80% Memória
Dask, PySpark	Q1–Q22	C (HPA 2→6)	2–4	4–8 GB	2–6	SF 10 e SF 50	70% CPU, 80% Memória

O plano experimental contempla três cenários (A, B e C), nos quais é avaliada a elasticidade horizontal. Inicialmente, cada *worker* é provisionado com 2 núcleos de CPU e 4 GB de memória RAM como *request*, podendo trabalhar verticalmente com um *limit* de até 4 núcleos e 8 GB de memória. O número mínimo de 2 *workers* foi definido pelo requisito de memória da SF 50: com 4 GB de *request* por *worker*, um único *worker* disporia de no máximo 8 GB (no *limit*), insuficiente para as consultas mais intensivas em memória nessa escala. Quando algum *threshold* baseado nesses *requests* é atingido, ocorre a elasticidade horizontal por meio da alocação de novos *workers*. A diferença entre os cenários está na quantidade máxima de *workers* permitida pelo HPA:

- **Cenário A (2 fixos):** mantém exatamente 2 *workers* durante toda a execução, sem escalonamento horizontal, servindo como *baseline*.
- **Cenário B (HPA 2→4):** inicia com 2 *workers* e permite o escalonamento horizontal até o máximo de 4 *workers*, conforme a demanda.
- **Cenário C (HPA 2→6):** inicia com 2 *workers* e permite o escalonamento horizontal até o máximo de 6 *workers*, conforme a demanda.

Os tetos de 4 e 6 *workers* foram escolhidos para avaliar o impacto incremental da capacidade de escalonamento, mantendo o consumo total de recursos dentro dos limites da infraestrutura disponível (32 vCPUs e 60 GB de memória). O provisionamento automático de recursos nos Cenários B e C é controlado pelo HPA do Kubernetes. A política de expansão (*scale-up*) adiciona até 100% do número atual de *workers* a cada 15 segundos, com janela de estabilização de 0 segundos. Foram definidos dois limiares (*threshold*) de utilização: 70% para CPU e 80% para memória. A política de redução (*scale-down*) é mais conservadora: ambas as métricas devem permanecer abaixo dos limiares durante 300 segundos consecutivos, com no máximo um *worker* desalocado a cada 60 segundos, reduzindo oscilações (*flapping*).

5. Resultados e Discussão

5.1. Escalabilidade

Os resultados do experimento de escalabilidade são apresentados na tabela 5, que mostra o tempo total de execução das 22 consultas com 1 núcleo (referência), acompanhado do número mínimo de execuções por consulta (n) e do intervalo de confiança de 95%, além do *speedup* e da eficiência para as configurações com múltiplos núcleos.

Tabela 5. Escalabilidade: tempo total das 22 consultas com 1 núcleo (n , IC 95%), *speedup* ($S_p = T_1/T_p$) e eficiência ($E_p = S_p/p$).

SF	Biblioteca	n	Tempo (s), 1 núcleo Média $\pm$ IC _{95%}	Speedup			Eficiência		
				8	16	32	8	16	32
1	DuckDB	32	7,00 $\pm$ 0,04	1,94	2,10	1,91	0,24	0,13	0,06
1	Polars	32	15,35 $\pm$ 0,01	3,22	4,02	3,86	0,40	0,25	0,12
1	Dask	32	35,88 $\pm$ 0,03	1,13	1,13	1,09	0,14	0,07	0,03
1	Pandas	32	94,96 $\pm$ 0,03	1,02	1,02	0,96	0,13	0,06	0,03
1	PySpark	32	149,18 $\pm$ 0,25	2,57	2,74	2,13	0,32	0,17	0,07
10	DuckDB	32	48,15 $\pm$ 1,00	4,13	5,63	6,16	0,52	0,35	0,19
10	Polars	32	191,26 $\pm$ 6,57	3,58	4,65	4,81	0,45	0,29	0,15
10	Dask	30	505,26 $\pm$ 11,07	1,30	1,30	1,24	0,16	0,08	0,04
10	PySpark	30	626,05 $\pm$ 14,92	3,43	4,54	4,33	0,43	0,28	0,14
50	DuckDB	32	902,78 $\pm$ 52,80	1,47	1,44	1,38	0,18	0,09	0,04
50	Dask	27*	7.250,60 $\pm$ 157,98	1,27	1,27	1,27	0,16	0,08	0,04
50	PySpark	29*	8.088,78 $\pm$ 130,89	3,89	5,26	6,41	0,49	0,33	0,20

Fonte: Elaborado pelo autor. n : menor número de execuções entre as consultas; IC_{95%}: margem do intervalo de confiança via distribuição t de Student; *: $n < 30$. Pandas excluída em SF 10 (Q5, Q8, Q9 não concluídas por estouro de memória) e em SF 50 (máximo 8 das 22 consultas concluídas). Polars excluída em SF 50 (Q12 e Q14 não concluídas).

A DuckDB apresentou o menor tempo absoluto de execução em todas as escalas e configurações avaliadas: 7,00 s em SF 1, 48,15 s em SF 10 e 902,78 s em SF 50, todas com 1 núcleo. Em SF 10, a DuckDB obteve o melhor *speedup* entre todas as bibliotecas (4,13 com 8 núcleos, 5,63 com 16 e 6,16 com 32), acompanhado da melhor eficiência nessa escala (0,52, 0,35 e 0,19). O desempenho inferior de *speedup* em SF 1 (máximo 2,10) e SF 50 (máximo 1,47) decorre, respectivamente, do baixo tempo base, que limita o ganho absoluto, e da alta variabilidade nos tempos em volumes maiores.

A Polars foi a segunda mais rápida em tempo absoluto em SF 1 (15,35 s) e SF 10 (191,26 s), superada apenas pela DuckDB. Seu *speedup* foi o melhor em SF 1 (3,22 a 4,02) e o segundo melhor em SF 10, com eficiência máxima de 0,45. Na SF 50, a Polars não concluiu as consultas Q12 e Q14, sendo excluída da comparação nessa escala.

A Pandas e a Dask apresentaram *speedup* próximo de 1,0 em todas as escalas avaliadas. Na Pandas (somente SF 1), os valores entre 0,96 e 1,02 confirmam que a biblioteca operou com um único núcleo devido à sua arquitetura *single-thread*; nas escalas 10 e 50, a Pandas foi excluída por não concluir todas as consultas (Q5, Q8, Q9 em SF 10; máximo 8 de 22 em SF 50). A Dask obteve *speedup* de 1,09 a 1,30 e eficiência máxima de 0,16, os menores valores do estudo, resultado do *Global Interpreter Lock* (GIL) do Python, que restringe o paralelismo de *threads* no modo local.

A PySpark apresentou os piores tempos absolutos em SF 1 (149,18 s) e SF 50 (8.088,78 s), mas foi a biblioteca com melhor aproveitamento dos recursos adicionais em SF 50, com *speedup* de 3,89 (8 núcleos), 5,26 (16 núcleos) e 6,41 (32 núcleos), e eficiência de 0,49, 0,33 e 0,20, as maiores dessa escala.

Com base nesses resultados, conclui-se que nenhuma das bibliotecas avaliadas apresentou escalabilidade plena nos cenários testados: o aumento do número de núcleos não se traduziu em ganho proporcional de desempenho. A eficiência máxima observada foi de 0,52 (DuckDB, SF 10, 8 núcleos) e, na maioria das configurações, manteve-se abaixo de 0,40. A PySpark foi a única a demonstrar tendência de ganho crescente com o volume, atingindo *speedup* de 6,41 em SF 50; contudo, a eficiência de 0,20 com 32 núcleos indica aproveitamento parcial dos recursos. Os resultados indicam, portanto, escalabilidade sublinear e limitada para todas as bibliotecas avaliadas.

5.2. Elasticidade

Nesta seção, apresentam-se os resultados da avaliação de elasticidade das bibliotecas Dask e PySpark nos três cenários descritos na Seção 4.2. A tabela 6 consolida os resultados dos Cenários A, B e C. Em ambas as escalas, as comparações consideram 19 consultas (excluídas Q7, Q18 e Q21, não concluídas pela Dask), com a PySpark recorrida para o mesmo conjunto.

Tabela 6. Resultados dos experimentos de elasticidade: Cenários A (2 fixos), B (HPA 2→4) e C (HPA 2→6).

Cen.	SF	Biblioteca	n	Tempo total (s)	CPU (%)	Memória (GB)	Workers	Up/Down
				Média ± IC _{95%}	Méd/Máx	Méd/Máx	Méd/Máx	
A	10	Dask	36	187,27 ± 0,91	24,0 / 53,3	2,5 / 6,0	2,0 / 2,0	0 / 0
A	10	PySpark	30	368,26 ± 0,97	19,1 / 34,6	0,7 / 2,1	2,0 / 2,0	0 / 0
A	50	Dask	29*	2.641,28 ± 33,99	37,4 / 58,6	3,6 / 6,9	2,0 / 2,0	0 / 0
A	50	PySpark	29*	6.154,89 ± 32,06	25,5 / 64,9	3,6 / 6,1	2,0 / 2,0	0 / 0
B	10	Dask	31	163,50 ± 2,15	22,4 / 54,8	0,9 / 4,3	2,3 / 3,0	8 / 8
B	10	PySpark	32	377,73 ± 0,79	19,0 / 68,8	0,7 / 3,8	2,0 / 2,0	0 / 0
B	50	Dask	31	1.627,42 ± 19,63	33,0 / 65,4	2,9 / 7,5	3,7 / 4,0	40 / 30
B	50	PySpark	29*	2.198,05 ± 38,92	42,2 / 82,8	3,2 / 8,3	3,8 / 4,0	26 / 23
C	10	Dask	32	164,46 ± 2,14	19,7 / 54,1	1,4 / 5,9	2,6 / 4,0	16 / 16
C	10	PySpark	34	380,08 ± 0,90	19,0 / 68,6	0,7 / 2,6	2,0 / 2,0	0 / 0
C	50	Dask	32	1.418,47 ± 27,28	32,7 / 71,3	2,4 / 7,9	5,3 / 6,0	54 / 55
C	50	PySpark	32	1.538,67 ± 8,48	42,3 / 84,6	3,5 / 8,1	5,9 / 6,0	19 / 18

Fonte: Elaborado pelo autor. Tempo total das 19 consultas comparáveis em ambas as escalas (excluídas Q7, Q18 e Q21 por falha da Dask). n: menor número de execuções por consulta. *: $n < 30$.

No Cenário A (*baseline*, 2 *workers* fixos), a Dask foi mais rápida que a PySpark em ambas as escalas: 187 s contra 368 s em SF 10 (1,97×) e 2.641 s contra 6.155 s em SF 50 (2,33×). Em SF 50, a memória média atingiu 3,6 GB para ambas as bibliotecas, acima do limiar de 80% do *request* (3,2 GB), configurando pressão persistente que, neste cenário sem HPA, não pôde ser aliviada por escalonamento horizontal.

No Cenário B (HPA 2→4), a permissão de escalonamento reduziu substancialmente os tempos em SF 50 em relação ao *baseline*: a Dask caiu de 2.641 s para 1.627 s (redução de 38,4%) e a PySpark de 6.155 s para 2.198 s (64,3%). Em SF 10, a PySpark permaneceu estática em 2 *workers* (0/0), enquanto a Dask escalou de forma limitada (média 2,3, pico 3,0, com 8/8 eventos). Em SF 50, ambas pressionaram o teto de 4 *workers*: a Dask com média de 3,7 (40/30 eventos) e a PySpark com média de 3,8 (26/23).

No Cenário C (HPA 2→6), o teto ampliado foi efetivamente atingido em SF 50: a Dask operou com média de 5,3 *workers* (54/55 eventos) e a PySpark com 5,9 (19/18), mantendo a alocação próxima do máximo. A memória média atingiu 2,4 GB e 3,5 GB, com máximos de 7,9 GB e 8,1 GB, aproximando-se do limite de 8 GB e confirmando que o escalonamento respondeu à pressão conjunta de CPU e memória. A Dask mostrou-se mais reativa, com maior frequência de transições, enquanto a PySpark manteve platôs mais estáveis.

A progressão entre os três cenários em SF 50 evidencia o impacto crescente da elasticidade. A razão entre os tempos da PySpark e da Dask reduziu de $2,33 \times$ (A) para $1,35 \times$ (B) e $1,08 \times$ (C), indicando que a PySpark é mais sensível à configuração do HPA. A redução total do Cenário A para o C foi de 46,3% para a Dask e 75,0% para a PySpark. Em SF 10, o efeito da elasticidade foi limitado: a PySpark não acionou o HPA em nenhum cenário, e a Dask escalou de forma restrita, confirmando que o *autoscaling* só se torna efetivo quando o volume de dados sustenta o cruzamento dos *thresholds*.

Com base nos resultados, conclui-se que as bibliotecas avaliadas conseguem explorar a elasticidade horizontal de modo satisfatório quando o volume de dados é suficiente para acionar o HPA. A elasticidade observada foi orientada principalmente pela pressão de memória, enquanto a CPU permaneceu com ampla margem. Diferenças no número de execuções por consulta (n) entre cenários decorrem de variações nos tempos das consultas mais longas: em configurações com menor capacidade de escalonamento, consultas intensivas eventualmente atingem limites de tempo ou memória, resultando em menor número de execuções concluídas.

6. Conclusão

Os experimentos revelaram que nenhuma das bibliotecas avaliadas apresentou escalabilidade plena nos cenários testados: para todas elas, a eficiência caiu conforme novos núcleos eram adicionados, caracterizando escalabilidade sublinear. A DuckDB e a Polars apresentaram os melhores resultados em tempo absoluto e, em SF 10, também os melhores *speedups*; em SF 50, a PySpark obteve o melhor *speedup* (6,41) e a maior eficiência (0,49 com 8 núcleos), embora com os maiores tempos absolutos.

Entre as bibliotecas avaliadas no cenário de elasticidade, a PySpark foi a que melhor aproveitou os recursos dinâmicos alocados pelo Kubernetes via HPA, mantendo a alocação média de *workers* mais próxima do teto configurado, com maior utilização de CPU e memória e menor oscilação de *scale-ups/scale-downs*; a Dask, por sua vez, foi a mais rápida em tempo de execução, embora com utilização média inferior e comportamento mais reativo de escalonamento. A comparação entre o *baseline* sem escalonamento (Cenário A) e o cenário com maior teto (Cenário C) evidencia reduções de 46% (Dask) e 75% (PySpark) no tempo total em SF 50, confirmando o ganho efetivo proporcionado pela elasticidade horizontal.

Como trabalhos futuros, sugere-se avaliar fatores de escala maiores e ambientes com múltiplos nós físicos, explorar novos *thresholds* e gatilhos de escalonamento, incorporar métricas alternativas de *autoscaling* além de CPU e memória, testar cenário de elasticidade vertical, e analisar resultados por consulta individual para identificar os fatores que limitam o *speedup*, ampliando a generalização dos resultados obtidos.

Referências

- Balusamy, B., Abirami R, N., Kadry, S., and Gandomi, A. H. (2021). *Big Data: Concepts, Technology, and Architecture*. Wiley.
- Dask Development Team (2026). Dask documentation. Acessado em: 5 de agosto de 2026.
- Drabas, T. and Lee, D. (2017). *Learning PySpark*. Packt Publishing, Birmingham, UK.
- Gu, R., Shi, J., Chen, X., Wang, Z., Che, Y., Zhang, K., and Huang, Y. (2022). Octopusdf: Unified dataframe-based cross-platform data analytic system. *Parallel Computing*, 110:102879. Acessado em: 5 de agosto de 2026.
- Herbst, N. R., Kounev, S., and Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not. *Proceedings of the 10th International Conference on Autonomic Computing (ICAC '13)*, pages 23–27.
- Kubernetes (2026). Kubernetes documentation. Acesso em: 1 Ago 2026.
- Lehrig, S., Eikerling, H., and Becker, S. (2015). Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA '15)*, pages 83–92.
- McKinney, W. (2023). *Python para Análise de Dados: Tratamento de Dados com pandas, NumPy & Jupyter*. Novatec Editora, São Paulo, 3 edition.
- Medel, V. et al. (2024). Characterising resource management performance in kubernetes. *arXiv preprint arXiv:2401.17125*.
- Mozzillo, A., Zecchini, L., Gagliardelli, L., Aslam, A., Bergamaschi, S., and Simonini, G. (2025). Evaluation of dataframe libraries for data preparation on a single machine.
- Nahrstedt, F., Karmouche, M., Bargiel, K., Banijamali, P., Nalini Pradeep Kumar, A., and Malavolta, I. (2024). An empirical study on the energy usage and performance of pandas and polars data analysis python libraries. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE '24*, page 58–68, New York, NY, USA. Association for Computing Machinery.
- Petersohn, D. (2018). Scaling interactive data science transparently with modin. Master's thesis, EECS Department, University of California, Berkeley.
- Piñeiro, Á. M. (2022). Big data para la comprensión de métricas de e-sports usando tecnologías web. Master's thesis, E.T.S. de Ingenieros Informáticos, Universidad Politécnica de Madrid, Madrid. Acessado em: 5 de agosto de 2026.
- Putrama, I. M. and Martinek, P. (2024). Heterogeneous data integration: Challenges and opportunities. *Data in Brief*, 56:110853.
- Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. *Proceedings of the 14th Python in Science Conference*, pages 130–136.
- Transaction Processing Performance Council (2023). Tpc-h benchmark specification. TPC Official Documentation.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Proc. 9th USENIX NSDI*, pages 15–28.