Universidade do Porto
Faculdade de Ciências

Estudo Comparativo de Processamento de Dados em Larga Escala em Python

Autores: Gabriela Simon de Cenço e Robert Gleison dos Reis Pereira
Unidade Curricular: Ciência de Dados em Larga Escala
Professores: Inês de Castro Dutra e Zafeiris Kokkinogenis
Ano Acadêmico: 2024/2025
Data: Junho de 2025

Sumário

1 Visão Geral da Estrutura do Projeto e Ferramentas

Este projeto serviu como prática de laboratorio para aplicar e testar boas práticas de desenvolvimento de software, engenharia de dados em sitemas distribudos, ciência de dados com machine learning e de realização de profiling e benchmarkings.

1.1 Linguagem de Programação e Bibliotecas

Versão do Python utilizada no projeto:

- 3.10.13. Especificada no arquivo .python_version para garantir consistência entre os ambientes.
- Todas as ferramentas de processamento foram instaladas sem problemas, mas, em vez de utilizar o Koalas, que foi descontinuado nesta versão do Python, estamos utilizando o pyspark.pandas, o sucessor oficial, que oferece melhor integração com o Apache Spark.

  OBS: durante o documento, iremos utilizar a nomenclatura "koalas", mas toda vez que nos referirmos a "koalas", estamos nos referindo à pyspark.pandas.

1.2 Gerenciamento de Dependências e Pacotes

- Poetry: Utilizado para gerenciar as dependências e empacotamento do projeto. Ele simplifica o controle de versões via o pyproject.toml, arquivos de lock, e consegue resolver conflitos de dependência automaticamente.
- Venv: Utilizado para criar ambientes virtuais, garantindo ambientes isolados e reproduzíveis.
- Makefile: Foi utilizado um Makefile para automatizar a criação do ambiente virtual, instalação das dependências com o Poetry, instalação do python e JDK (para spark e koalas) na máquina virtual.

1.3 Configuração de Infraestrutura

- Terraform (Infraestrutura como Código - IaC): Em vez de criar os recursos do GCP (como máquina virtual e cluster) via linha de comando ou console, adotamos o Terraform como Infraestrutura como Código, para facilitar alterações, controle do ciclo de vida e também com propósito de aprendizado. O código do Terraform está localizado no módulo terraform_infra, e foi utilizado para provisionar:
  - Instância de máquina virtual
  - Cluster Kubernetes no GKS

1.4 Estratégia de Execução de Código

Scripts Python ao invés de Jupyter Notebooks. O processamento foi realizado em arquivos Python, por facilitar a depuração, melhorar a modularidade e aumentar a reutilização do código. Além disso, rodar um arquivo python em uma máquina virtual é mais fácil.

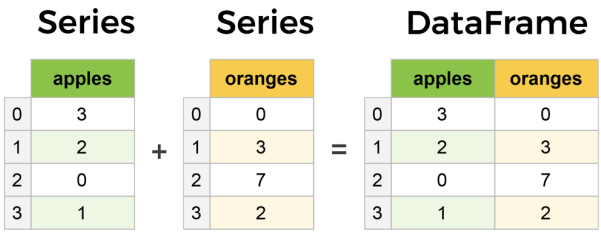
2 Background das Bibliotecas

Nesta seção, apresenta-se uma visão geral das principais bibliotecas utilizadas para processamento distribuído e paralelo de dados em Python.

2.1 Pandas

Visão geral: O Pandas é uma biblioteca de software de código aberto desenvolvida em Python para análise e manipulação de dados. A biblioteca Pandas oferece estruturas de dados projetadas especificamente para lidar com conjuntos de dados tabulares por meio de uma API simplificada em Python. O Pandas é uma extensão do Python voltada para o processamento e manipulação de dados tabulares, implementando operações como carregamento, alinhamento, mesclagem e transformação de conjuntos de dados de forma eficiente.

Funcionamento: No núcleo da biblioteca open-source do Pandas estão as estruturas de dados DataFrame e Series, usadas para lidar com dados tabulares e estatísticos. Um DataFrame do Pandas é uma tabela bidimensional, semelhante a um array, onde cada coluna representa os valores de uma variável específica, e cada linha contém um conjunto de valores correspondentes a essas variáveis.



Especificações de processamento de dados:

- Arquitetura: Single-node (nó único). Processamento de dados em memória (RAM).
- Multithreading: As operações são majoritariamente single-threaded devido ao Global Interpreter Lock (GIL) do CPython.
- Tipo de execução: Eager (execução imediata).
- Resiliência: Não possui mecanismos nativos de tolerância a falhas ou recuperação.
- Otimizações: Utiliza internamente o NumPy, que é rápido e escrito em C, mas pode não ser suficiente para conjuntos de dados muito grandes.
- Suporte a I/O: Parquet, JSON, CSV, texto etc. (mas apenas para nó único/local).

Limitações: Embora o pandas ofereça um conjunto abrangente de funcionalidades e suporte à maioria das operações necessárias para dados tabulares, ele possui limitações de escalabilidade por ser single-node, com execução eager e depender de que todo o conjunto de dados caiba na memória (RAM). Em outras palavras, embora seja uma ferramenta poderosa para o processamento de dados em pequena e média escala, normalmente não é utilizada em contextos de big data, nos quais os conjuntos de dados podem exceder a memória disponível. Para alcançar maior escalabilidade, outras ferramentas foram desenvolvidas para substituir ou complementar o pandas — e discutiremos algumas delas a seguir.

2.2 Joblib

Visão geral: Joblib é uma biblioteca Python de código aberto desenvolvida para fornecer utilitários leves de pipelining e paralelização, principalmente voltados para tarefas de computação numérica.

É especialmente adequada para casos de uso que envolvem operações repetitivas ou intensivas em CPU (CPU-bound), sendo comumente utilizada em conjunto com bibliotecas como o scikit-learn para treinamento eficiente de modelos e validação cruzada. Joblib melhora o desempenho por meio de cache e multiprocessamento, exigindo mudanças mínimas no código do usuário.

Funcionamento: Joblib oferece duas funcionalidades principais: cache transparente baseado em disco e computação paralela. Suas construções Parallel e delayed permitem que o usuário paralelize loops em múltiplos cores usando multiprocessing (ou multithreading, quando apropriado). O Joblib cuida automaticamente da serialização e do envio das tarefas entre os processos, facilitando a escalabilidade de tarefas intensivas em CPU em uma única máquina. Além disso, seu mecanismo de cache armazena os resultados de chamadas de funções custosas e os reutiliza quando os parâmetros de entrada não mudaram, evitando recomputações desnecessárias.

Especificações de processamento de dados:

- Arquitetura: Single-node, com paralelismo entre múltiplos núcleos (multi-core).
- Multithreading/Multiprocessing: Usa o módulo multiprocessing do Python ou o backend loky (baseado em processos). Backend com threading também está disponível, mas é limitado pelo Global Interpreter Lock (GIL).
- Tipo de execução: Execução eager (as tarefas são despachadas e executadas imediatamente).
- Resiliência: Não possui tolerância a falhas embutida, mas conta com pontos de verificação no cache que podem armazenar a última versão de algumas computações.
- Otimizações: Uso eficiente de memória na serialização de tarefas por meio de estratégias personalizadas de pickling; suporta memory mapping de grandes arrays do NumPy para evitar cópia de dados entre processos; execução simultânea de tarefas com delayed().
- Suporte a I/O: Não realiza leitura direta de arquivos, apenas escrita (dump). Portanto, é necessário usar ferramentas como o Pandas para transformar arquivos em DataFrame.

Limitações: Embora o Joblib seja altamente eficaz para paralelizar operações em uma única máquina, ele não é projetado para computação distribuída entre clusters ou múltiplos nós. Seu paralelismo é limitado ao número de cores disponíveis e pode ser prejudicado pelo GIL do Python quando se utiliza multithreading. Além disso, o mecanismo de cache do Joblib é baseado em arquivos e pode se tornar um gargalo em conjuntos de dados muito grandes ou com acesso frequente de leitura/gravação. Para cargas de trabalho distribuídas em grande escala, ferramentas como Dask ou Spark são mais adequadas. **Mais indicado para:**

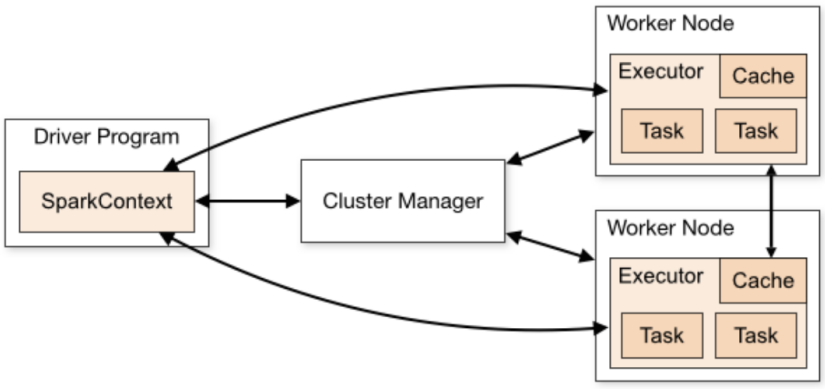
- Paralelizar tarefas CPU-bound em uma única máquina.
- Fluxos de trabalho de aprendizado de máquina para acelerar o treinamento de modelos, a busca de hiperparâmetros (hyperparameter tuning)

2.3 PySpark

Visão geral: O Apache Spark, lançado em 2014 com sua interface em Python chamada PySpark, trouxe o paradigma da computação distribuída para o processamento de dados em larga escala. Ele foi projetado para lidar com conjuntos de dados massivos distribuídos entre clusters de máquinas, sendo ideal para análises de big data. O Spark possui uma grande comunidade e documentação extensa. Embora seja baseado na JVM (Java Virtual Machine), o PySpark permite que usuários Python aproveitem seus poderosos recursos de processamento distribuído. Ele suporta várias linguagens, incluindo Java e Scala, e é amplamente utilizado em setores que exigem processamento e análise de dados em grande escala.

Funcionamento: No núcleo do PySpark está a computação distribuída, onde os dados são divididos em pedaços menores (partições) e processados em paralelo por um cluster de máquinas. O PySpark suporta os RDDs (Resilient Distributed Datasets), que são coleções imutáveis de objetos distribuídas entre os nós. Com a introdução dos DataFrames (a partir do Spark 1.3), o PySpark passou a oferecer uma abstração mais estruturada para o processamento de dados, permitindo operações semelhantes a SQL e aplicando otimizações. Os DataFrames proporcionam uma interface mais simples para trabalhar com dados, mantendo os recursos de computação distribuída e tolerância a falhas.

O **SparkContext** serve como ponto de entrada para executar operações no cluster, e a avaliação preguiçosa (lazy evaluation) permite que o Spark otimize a execução das tarefas antes de realizar os cálculos de fato.



O **Cluster Manager** é responsável por alocar recursos (CPU, memória) e agendar tarefas entre os diversos nós de trabalho (Worker Nodes). Ele garante que os trabalhos sejam distribuídos de forma eficiente e pode reagendar tarefas caso um nó falhe.

Os **Worker Nodes** são as máquinas que realizam efetivamente o processamento dos dados. Cada nó de trabalho executa executores, que são responsáveis por processar tarefas específicas, armazenar dados temporários e relatar o progresso de volta ao Driver.

Especificações de Processamento de Dados:

- Arquitetura: Distribuída, baseada em cluster (arquitetura master-worker)
- Modelos principais de dados:
  - RDDs (Resilient Distributed Datasets): coleções de dados distribuídas e tolerantes a falhas
  - DataFrames: abstração de dados estruturados em formato colunar (introduzidos no Spark 1.3)
  - Datasets: API fortemente tipada, usada principalmente em Scala
- Paralelismo: Processamento de dados massivamente paralelo e distribuído
- Processamento: Em memória, com fallback para disco (em caso de limite de memória)
- Tipo de execução: Avaliação preguiçosa (lazy evaluation), com transformações avaliadas via um Directed Acyclic Graph (DAG)
- Resiliência: Alta — rastreamento de linhagem (lineage) para total tolerância a falhas e capacidade de reconstrução de dados perdidos
- Tolerância a falhas: Rastreamento de linhagem em transformações estreitas (narrow) e checkpointing em transformações amplas (wide)
- Agendador: Baseado em estágios, com otimizações de shuffle
- Agendador: Baseado em estágios, com otimizações de shuffle
- Otimização de consultas: Catalyst Optimizer — faz análise, otimização lógica e planejamento físico das consultas
- Suporte a I/O: Leitura e escrita em formatos como: Parquet, ORC, Avro, JSON, CSV, texto, JDBC, entre outros
- Integrações: Hive, HBase, Cassandra, Kafka e mais
- Processamento em streaming: Structured Streaming com processamento em micro-batches ou contínuo
- Escalabilidade: Lida com petabytes de dados em clusters com milhares de nós

Limitações:

- Sobrecarga para conjuntos de dados pequenos: O Spark introduz uma sobrecarga significativa para conjuntos de dados pequenos ou operações em nó único, tornando-se ineficiente para casos em que a computação distribuída não é necessária.
- Degradação de desempenho: A sobrecarga de serialização entre Python e JVM pode degradar o desempenho, especialmente em transformações complexas ou quando grandes volumes de dados são transferidos entre os ambientes.

- Complexidade operacional: Configurar, gerenciar e ajustar clusters Spark pode ser complexo, especialmente para usuários iniciantes, exigindo ferramentas robustas de gerenciamento de clusters.

Mais indicado para:

- Processamento de dados em larga escala e análises distribuídas, especialmente quando se trabalha com conjuntos de dados massivos que não cabem na memória de uma única máquina.

2.4 PySpark.pandas (Koalas)

Visão geral: O PySpark.pandas, anteriormente conhecido como Koalas, oferece uma API semelhante à do pandas sobre o Apache Spark. Ele foi criado para facilitar a vida de cientistas de dados familiarizados com o pandas que desejam escalar seus fluxos de trabalho para grandes volumes de dados distribuídos em clusters Spark. Ao manter a mesma API do pandas, o PySpark.pandas preenche a lacuna entre o processamento de dados em pequena escala (onde o pandas se destaca) e o processamento distribuído em larga escala (especialidade do Spark).

Enquanto o pandas é excelente para manipulação de dados em um único nó, o PySpark.pandas estende essas capacidades para funcionar em dados distribuídos, aproveitando o mecanismo de execução eficiente do Spark. Isso o torna ideal para quem deseja escalar suas operações sem precisar aprender a API nativa do Spark.

Funcionamento: Ele oferece a mesma sintaxe do pandas, facilitando a transição para usuários que já o conhecem. Internamente, o PySpark.pandas converte as operações do tipo pandas em operações Spark e as executa no cluster, mantendo a compatibilidade.

O DataFrame do PySpark.pandas é baseado nas abstrações de DataFrame e RDD do Spark. As operações são distribuídas, ou seja, os dados são divididos em partições e processados em paralelo em um cluster. A principal vantagem é que o usuário pode escalar seu código pandas para grandes volumes de dados sem precisar reescrevê-lo na API nativa do Spark.

Especificações de Processamento de Dados:

- Arquitetura: Distribuída, baseada em cluster (arquitetura master-worker)
- Modelos principais de dados:
  - PySpark.pandas DataFrame: abstração de DataFrame distribuído semelhante ao pandas
  - Spark DataFrame: estrutura subjacente usada internamente
- Paralelismo: Processamento massivamente paralelo e distribuído
- Processamento: Em memória, com fallback para disco (quando o limite de memória é atingido)
- Tipo de execução: Lazy evaluation com transformações avaliadas por meio de um DAG (Directed Acyclic Graph)

- Resiliência: Alta — rastreamento de linhagem garante tolerância a falhas e reconstrução de dados
- Tolerância a falhas: Utiliza mecanismos do Spark, como lineage e checkpointing
- Agendador: Baseado em estágios com otimizações de shuffle
- Otimização de consultas: Catalyst Optimizer — análise de consulta, otimização lógica e planejamento físico
- Suporte a I/O: Leitura e escrita em formatos como Parquet, ORC, Avro, JSON, CSV, texto, JDBC, etc.
- Integrações: Hive, HBase, Cassandra, Kafka e outros
- Processamento em streaming: Structured Streaming para fluxos de dados contínuos (como Kafka e sockets)
- Escalabilidade: Lida com petabytes de dados em clusters com milhares de nós

Limitações:

- Sobrecarga para conjuntos de dados pequenos: Assim como o PySpark, o PySpark.pandas tem uma sobrecarga considerável para tarefas em pequena escala, tornando-se ineficiente quando a computação distribuída não é necessária.
- Sobrecarga de serialização: Apesar de imitar o pandas, a conversão entre Python e JVM pode causar perda de desempenho, especialmente em transformações complexas ou quando há troca de grandes volumes de dados entre os ambientes.
- API limitada: Nem todas as funções do pandas estão disponíveis ou são otimizadas para computação distribuída. Algumas operações podem não ser suportadas ou apresentar limitações.
- Complexidade operacional: Mesmo com a familiaridade da API do pandas, gerenciar clusters Spark e otimizar o desempenho distribuído ainda pode ser desafiador para novos usuários, especialmente sem ferramentas adequadas de gerenciamento e ajuste de recursos.

Mais indicado para:

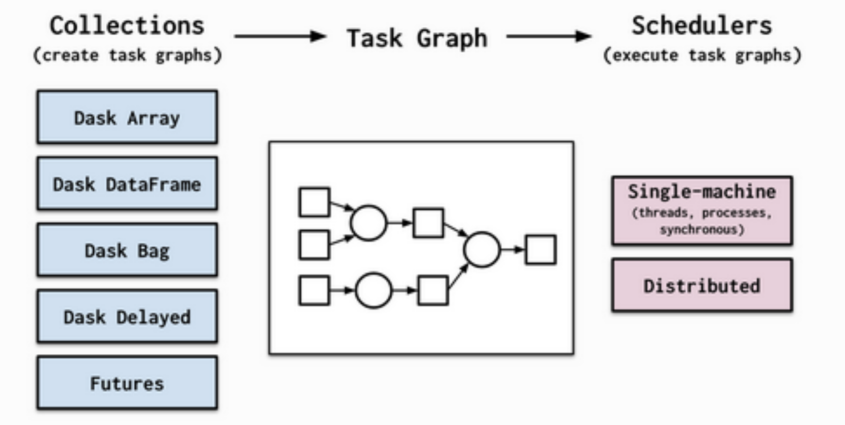
- Escalar fluxos de trabalho do pandas para processamento distribuídos.
- Aproveitar a computação distribuída do Apache Spark mantendo a familiaridade com a API do pandas. É uma excelente opção para quem quer escalar suas operações sem precisar mergulhar na complexidade da API nativa do Spark.

2.5 Dask

Visão geral: Dask é uma biblioteca open-source em Python projetada para computação paralela e distribuída, permitindo processamento e análise de dados escaláveis. Integra-se perfeitamente ao ecossistema PyData (como NumPy, Pandas, scikit-learn) e permite que usuários trabalhem com conjuntos de dados maiores que a memória, dividindo os cálculos em tarefas menores e gerenciáveis. Dask é especialmente indicado para cargas de trabalho de big data, oferecendo abstrações de alto nível (como DataFrames e Arrays paralelos) e agendamento de tarefas de baixo nível para fluxos de trabalho personalizados.

Funcionamento: Dask cria grafos de tarefas que representam os cálculos de forma lazy (adiada). Esses grafos são executados em paralelo, seja em múltiplos núcleos de uma única máquina, seja distribuídos em clusters multi-nós. Componentes principais:

- Agendamento de tarefas: Dask agenda dinamicamente as tarefas usando multithreading, multiprocessamento ou computação distribuída (via dask.distributed).
- Coleções paralelas: Oferece dask.array, dask.dataframe e dask.bag para operações similares a NumPy, Pandas e PySpark, com paralelismo automático.



Especificações de processamento de dados:

- Arquitetura: Nó único (multi-core) ou distribuída (clusters multi-nós).
- Multithreading/Multiprocessing: Suporta ambos; paralelismo baseado em threads para operações que liberam o GIL (ex: NumPy/Pandas) e paralelismo baseado em processos para tarefas pesadas em CPU (evitando o GIL, mas com overhead de serialização).
- Tipo de execução: Avaliação preguiçosa (lazy evaluation). Os cálculos só são executados quando explicitamente solicitados (ex: .compute() ou .persist()), permitindo otimizações como fusão de tarefas e gerenciamento eficiente de memória.
- Resiliência: Tolerância a falhas embutida no modo distribuído; suporta recuperação de falhas em workers e otimiza a execução via agendamento inteligente.
- Opções de agendador:
  - Threads: Para cargas que liberam o GIL (ex: operações NumPy/Pandas).

  - Processes: Para tarefas pesadas em CPU e que envolvem o GIL (evita o GIL, mas adiciona overhead de serialização).
  - Distributed: Para clusters, com tolerância a falhas e agendamento avançado (dask.distributed).
- Otimizações:
  - Processamento de dados em blocos (chunked).
  - Fusão de tarefas e otimização do grafo para minimizar overhead.
  - Integração com serialização rápida (ex: Cloudpickle, Arrow).
- Suporte a I/O: Parquet, JSON, CSV, texto, JDBC, etc.
- Escalabilidade: Escala adicionando mais nós em um cluster conforme aumenta a necessidade de processamento.

Limitações:

- Apesar de escalar bem, o desempenho pode não igualar frameworks especializados (ex: Spark para ETL distribuído muito grande), devido ao overhead do Python e mecanismos de consulta menos refinados.
- Depuração de grafos de tarefas complexos pode ser difícil.
- Requer ajuste manual (ex: tamanho dos blocos, escolha do agendador) para desempenho ideal.
- Não é ideal para workloads em tempo real/streaming (para isso, prefira Apache Flink, Faust, etc.).

Mais adequado para:

- Processamento de dados em larga escala (workflows out-of-core com Pandas/NumPy).
- Machine learning distribuído (ex: ajuste paralelo de hiperparâmetros com dask-ml).
- Pipelines ETL batch e algoritmos paralelos customizados.
- Escalar cargas de trabalho Python de um laptop para um cluster sem precisar reescrever o código.

2.6 Modin

Visão geral: Modin é uma biblioteca open-source em Python desenvolvida para acelerar e escalar fluxos de trabalho com Pandas com o mínimo de alterações no código. Ela oferece uma substituição direta ao Pandas, permitindo um processamento de dados mais rápido ao paralelizar automaticamente as operações entre vários núcleos de CPU ou até em clusters distribuídos. Modin é especialmente útil ao lidar com conjuntos de dados grandes que excedem a memória ou que exigem alto desempenho computacional.

Funcionamento: Modin substitui o mecanismo de execução do Pandas por um engine paralelo (inicialmente baseado em Ray, e agora também compatível com Dask e Unidist). As principais características incluem:

- Paralelismo transparente: divide automaticamente os DataFrames em partições e os processa em paralelo. As partições podem ser:
  - Particionamento em blocos (2D): Ideal para operações que exigem acesso por linha e coluna (ex: df.iloc, df.apply).
  - Particionamento por linha: Otimizado para operações baseadas em linhas (ex: df.apply(axis=1), df.dropna()).
  - Particionamento por coluna: Indicado para operações em colunas (ex: df[col].mean(), df.drop(columns=...)).

Especificações de processamento de dados:

- Arquitetura: Nó único (multi-core) ou distribuída (usando Ray ou Dask).
- Multithreading/Multiprocessing: Utiliza Ray/Dask para paralelismo baseado em processos, contornando as limitações do GIL.
- Tipo de execução: Principalmente imediata (eager, como no Pandas).
- Resiliência: Herdada do Dask no modo distribuído (reexecução de tarefas, recuperação de falhas em workers).
- Otimizações:
  - Paralelismo baseado em partições para operações como read_csv, groupby, join.
  - Agendamento inteligente de tarefas (via Ray ou Dask).
  - Uso de memória reduzido graças ao processamento particionado.
- Suporte a I/O:
  - Leitura/gravação em CSV, Parquet, SQL e outros formatos (os mesmos do Pandas).
  - Mais rápido que o Pandas na leitura de arquivos grandes devido ao carregamento paralelo.
- Escalabilidade:Funciona em máquinas únicas (multi-core) ou clusters distribuídos via Ray ou Dask.

Limitações:

- Não substitui o Pandas totalmente: Algumas funções específicas do Pandas podem não estar otimizadas ou ainda não são suportadas.
- Overhead de memória: A divisão em partições pode aumentar o uso de memória em relação ao Pandas em conjuntos de dados pequenos.
- Escalabilidade distribuída limitada vs. Spark: Embora Modin + Dask funcione bem em uma máquina ou pequeno cluster, para implantações em larga escala, o Spark ainda oferece um ecossistema mais maduro.
- Complexidade no modo distribuído: Configurar um cluster (Ray ou Dask) adiciona uma camada de complexidade em comparação ao uso local com Pandas.

Mais indicado para:

- Operações em grandes DataFrames (mais rápido que o Pandas em datasets volumosos).
- Usuários de Pandas que precisam de escalabilidade sem reescrever o código.
- Pipelines ETL, limpeza de dados e engenharia de atributos onde o Pandas se torna lento.
- Máquinas multi-core onde o processamento paralelo acelera os fluxos de trabalho.

2.7 RapidsAI (cuDF)

Visão geral: RAPIDS introduziu o cuDF, uma implementação de DataFrame que executa na GPU (não na CPU), radicalmente acelerando o processamento através do paralelismo massivo oferecido pelo hardware especializado de inúmeros cores que uma GPU oferece. Mas usar esse tipo de processamento GPU bound requer que o utilizador tenha uma GPU, em especial da Nvidia, que criar os drivers para cuDF.

Funcionamento:
Características técnicas:

- Arquitetura: GPU-accelerated, single-node ou multi-GPU
- Hardware: Requer GPUs NVIDIA com suporte a CUDA
- API: Similar ao Pandas, focada em compatibilidade
- Modelo de execução: Principalmente eager, com algumas capacidades lazy
- Estrutura de dados: Colunar, baseada na biblioteca CUDA primitiva libcudf
- Paralelismo:
  - SIMT (Single Instruction, Multiple Threads) da GPU
  - Milhares de threads em paralelo para processamento de dados
- Escalabilidade:
  - Vertical: Multi-GPU via Dask-cuDF
  - Horizontal: Através da integração com Dask para clusters GPU
- Integração:
  - Interoperabilidade com PyArrow
  - cuPy para computação numérica em GPU
  - XGBoost, cuML para machine learning acelerado
- Operações otimizadas:
  - Joins, sorts, aggregations acelerados por GPU

  - Filtro e transformação vetorizada
  - Computação de strings em GPU
- Formato de memória: Colunar compatível com Arrow
- Transferências: Zero-copy entre CPU e GPU quando possível
- Componentes do ecossistema RAPIDS:
  - cuDF: DataFrames na GPU
  - cuML: Machine learning na GPU
  - cuGraph: Análise de grafos na GPU
  - cuSpatial: Análise geoespacial na GPU
- Escala: Limitada pela memória da GPU (tipicamente 16-80GB por GPU)

Limitações técnicas: Requer hardware NVIDIA específico, limitações de memória da GPU, nem todas as operações Pandas são igualmente eficientes em GPU, overhead de transferência CPU-GPU para datasets que não cabem inteiramente na memória da GPU. Não funciona bem para tarefas complexas.

Mais indicado para:

- Tarefas repetitivas e com instruções simples
- Pre processamento intensivo de dados par treino de Machine learning
- Computações intensivas que tiram proveito de processamento em GPU

3 Especificações de Hardware e Software

- Máquina de testes local (computador dos integrantes):
  - CPU: 11th Gen Intel i7-1165G7 2.80Ghz
  - Cores: 8 cores (2 threads per core)
  - RAM: 16 (8+8) RAM DDR4
  - GPU: Intel Corporation TigerLake-LP GT2 [Iris Xe Graphics]
  - Disk: 500Gb SSD
  - SO: Ubuntu 22.04 x86
- Máquina virtual para benchmarking single node:
  - CPU: Intel Haswell 2.30GHz
  - Cores: 8 vCPUs
  - RAM: 52GB DDR4
  - GPU: none
  - Disk: 70GB SSD Persistent Disk

  - SO: Ubuntu 22.04 x86

- Cluster para benchmarking distribuído:
  - Nodes: 3
  - CPU: Intel Skylake 2.20GHz (per node)
  - Cores: 4 vCPUs (per node)
  - RAM: 16GB DDR4 (per node)
  - GPU: none
  - Disk: 10GB SSD Persistent Disk (per node)
  - SO: Ubuntu 22.04 x86

4 Descrição dos Datasets

4.1 NYC Taxi Driver Dataset

Para este projeto utilizamos os datasets NYC Taxi Driver, disponibilizados no site oficial da Comissão de Taxis e Limuosines da cidade de Nova York (`https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page`). Os datasets apresentam dados sobre corridas realizadas por mais de um tipo de veículo. Neste projeto, utilizamos apenas os datasets de taxis amarelos, os famosos taxis de Nova York.

O site reúne datasets de todos os meses desde janeiro de 2009 até o março de 2025, no momento presente. Para a nossa tarefa de benchmark utilizamos os datasets dos anos de 2009 e 2010, já para a tarefa de machine learning, utilizamos os datasets de 2011 e 2012.

Os dados do NYC Yellow Taxi de 2009 e 2010 têm uma estrutura diferente em relação aos de 2011 e 2012. A partir de 2011, novos campos foram incluídos que capturam mais detalhes de cada viagem, como localização de pickup/dropoff, por exemplo.

Colunas comuns em todos os datasets:

- **VendorID ou vendor_name:** ID do fornecedor dos dados (em formato de string para 2009 e 2010, e int64 para 2011 e 2012)
- **tpep_pickup_datetime:** a data e horário em que a corrida iniciou (em formato de string para 2009 e 2010, e timestamp[us] para 2011 e 2012)
- **tpep_dropoff_datetime:** a data e horário em que a corrida terminou (em formato de string para 2009 e 2010, e timestamp[us] para 2011 e 2012)
- **passenger_count:** o número de passageiros no taxi
- **trip_distance:** a distância percorrida (calculada pelo taxímetro)
- **RatecodeID ou Rate_Code:** o tipo de tarifa da corrida
- **store_and_fdw_flag ou store_and_forward:** indica se os dados da corrida foram guardados localmente na memoria do veiculo antes de enviar ao vendedor por falta de conexão ao servidor no momento da corrida (em formato de double para 2009 e 2010, e string para 2011 e 2012)

- **payment_type:** o método de pagamento utilizado pelo passageiro
- **fare_amount ou Fare_Amt:** o valor base da corrida, calculado a partir do tempo e distância percorridos
- **mta_tax:** valor fixo de $0.50 de taxa da Metropolitan Transport Authority
- **tip_amount ou Tip_Amt:** valores de gorgeta dados pelo passageiro
- **tolls_amount ou Tolls_Amt:** valores pagos em pedágios
- **improvement_surcharge ou surcharge:** taxa obrigatória de $0.30, implementada a partir de 2015
- **total_amount ou Total_Amt:** valor total pago na corrida

Colunas exclusivas dos datasets de 2009 e 2010:

- **Start_Lon:** longitude do início da corrida
- **Start_Lat:** latitude do início da corrida
- **End_Lon:** longitude do final da corrida
- **End_Lat:** latitude do final da corrida

Colunas presentes a partir dos datasets de 2011:

- **PULocationID:** ID da zona onde a corrida iniciou
- **DOLocationID:** ID da zona onde a corrida terminou
- **extra:** valores extras cobrados
- **congestion_surcharge:** taxa adicional, implementada em 2019
- **airport_fee:** taxa para corridas de/para aeroportos

5 Dificuldades durante a execução do trabalho

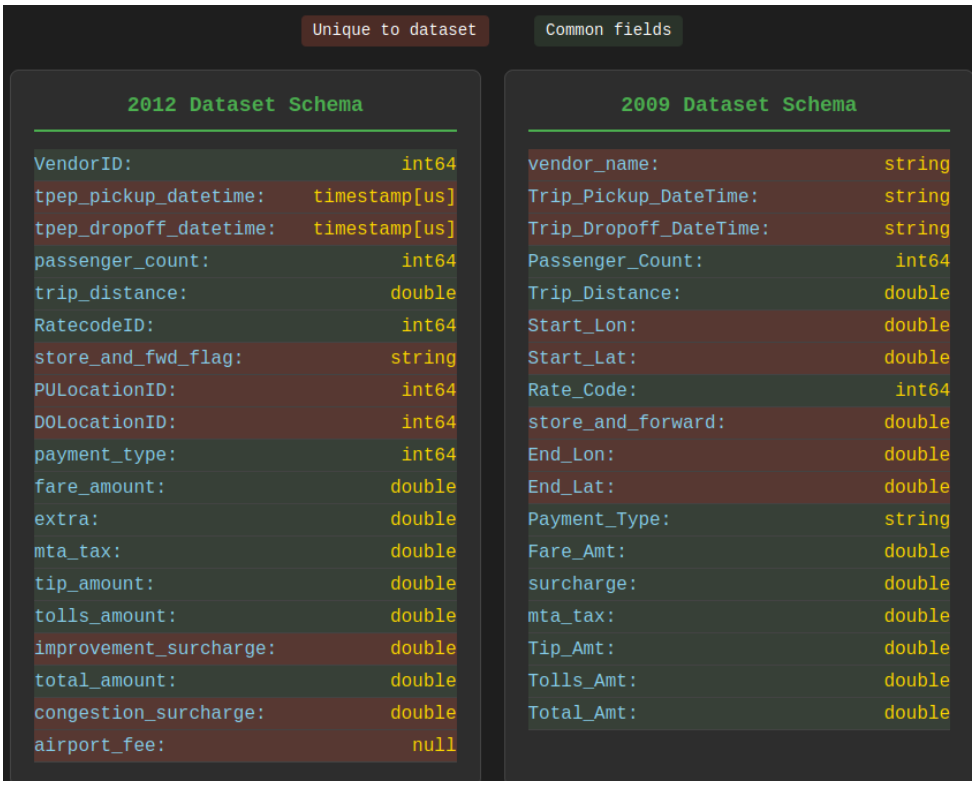
Durante a execução deste projeto, cometemos erros de implementação e de overengineering que nos custaram tempo de implementação. A grande realidade é que subestimamos este trabalho e durante o prazo final, algumas coisas podem não ter sido feitas conforme o que os professores esperavam, mas tratamos de relatar todos os processos feitos e o porquê de terem se tornado um gargalo para a finalização. Dentre eles:

- **Diferentes estruturas de dataset:** Os dados de táxis apresentam inconsistências estruturais entre diferentes períodos. Os datasets de 2009 e 2010 possuem conteúdo idêntico, mas com nomenclaturas de colunas divergentes, que foram tratadas durante a execução ao invés de pré-processamento.

  Os dados de 2011 em diante seguem uma estrutura diferente dos anos anteriores. Como a base de código foi desenvolvida inicialmente para o padrão de 2009, a adaptação para os formatos posteriores demandaria refatoração considerável.

Decisão de implementação: Mantivemos os datasets de 2009/2010 para os benchmarks devido ao maior volume de dados (500MB/mês vs. 150MB/mês dos anos posteriores). Para o treinamento do modelo, utilizamos os dados de 2011 e 2012 por incluírem a coluna de distância da viagem.

Limitação técnica: Os datasets de 2009/2010 contêm apenas coordenadas de longitude e latitude dos pontos de origem e destino, exigindo cálculo manual da distância, enquanto os datasets posteriores fornecem essa informação diretamente.



- **Modularização da codebase:** Optamos por desenvolver uma arquitetura modular com funções reutilizáveis e um ponto único de execução para todos os benchmarks, em contraste com abordagens mais simples como notebooks Jupyter independentes ou um notebook monolítico.

Desafios da modularização:

- **Complexidade arquitetural:** A implementação modular introduziu maior complexidade de design comparada a soluções mais diretas, exigindo planejamento maior da estrutura de módulos e interfaces.
- **Gerenciamento de dependências:** Conflitos entre versões de bibliotecas e dependências transitivas foram solucionados através do uso do Poetry como gerenciador de pacotes e ambientes virtuais, proporcionando isolamento e reprodutibilidade.
- **Orquestração de múltiplos clients:** A coordenação simultânea de diferentes clientes (Spark, Dask) e ferramentas de processamento através de uma única interface de comando mostrou-se difícil, especialmente no gerenciamento de recursos e estados das sessões.

- **Execução unificada:** Implementar um sistema que execute todos os benchmarks de forma integrada, mantendo isolamento entre ferramentas gerou uma manutenção mais complicada.

Benefícios obtidos: Apesar da complexidade inicial, a arquitetura modular facilitou significativamente os processos de debugging e de rodar os benchmarking num local centralizado.

O código-fonte completo está disponível em:

`https://github.com/RobertGleison/cdle-assignment`

Integração com Google Cloud Storage (GCS): A estratégia de armazenar datasets em um bucket GCS e salvar resultados em outro bucket trouxe desafios técnicos significativos relacionados ao acesso de dados remotos.

- **Compatibilidade de protocolos:** Algumas ferramentas (Modin, Joblib) não conseguem interpretar arquivos Parquet diretamente do GCS como arquivos locais tradicionais. Estes são tratados como objetos remotos que requerem URLs específicas com prefixo `gs://` para serem processados adequadamente.
- **Suporte nativo heterogêneo:** Enquanto ferramentas como Spark e Koalas possuem suporte nativo para GCS, outras bibliotecas necessitam de interfaces adicionais como `fsspec` ou `gcsfs` para estabelecer comunicação adequada com o sistema de arquivos remoto.
- **Problemas de autorização:** Configurações de permissões entre a VM do GCE e os buckets GCS geraram falhas de leitura e escrita, exigindo ajustes nas credenciais de acesso.

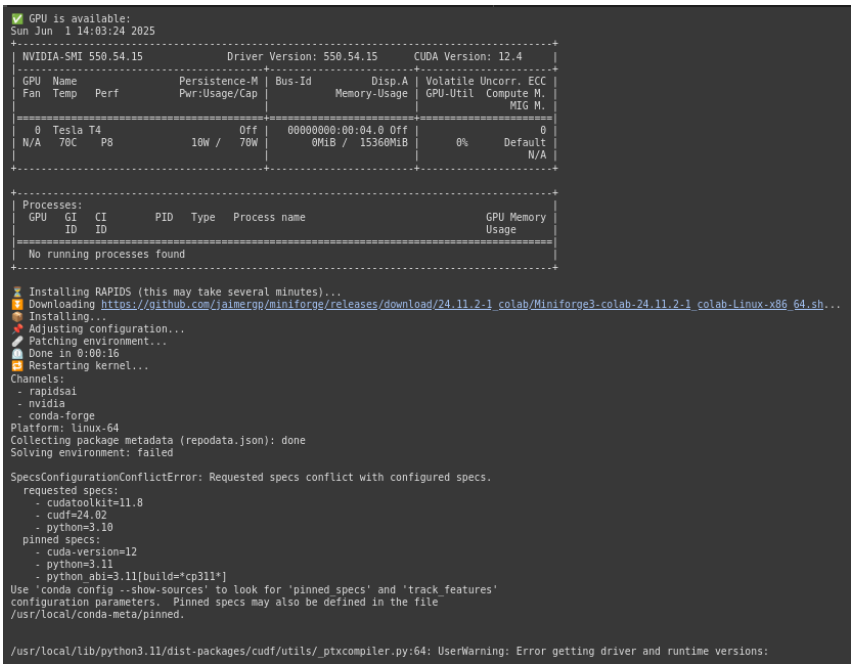
Tivemos que adicionar leitura de arquivos locais para testes e do GCP bucket para o benchmarking modificando o prefixo da URL.

Limitações de hardware para utilização de RapidsAi: A biblioteca RAPIDS AI requer o uso da lib cuDF e, consequentemente, uma GPU NVIDIA para utilizar CUDA. Nem nossa VM nem nossos computadores locais possuem o hardware e drivers necessários. Utilizamos o Google Colab, que oferece um kernel com GPU NVIDIA, para testar o RAPIDS AI, mas a infraestrutura limitada da versão gratuita (RAM e disco) impossibilita ter as mesmas condições do benchmarking. Por isso, não incluímos o RAPIDS AI no benchmarking final, embora reconheçamos seu potencial de ser extremamente rápido utilizando computação baseada em GPU. A foto abaixo mostra a instalação do cudf no kernel NVIDIA oferecido pelo google colab:

Deploy no Google Kubernetes Engine (GKE): O deployment no GKE representou um dos maiores desafios técnicos do projeto devido à inexperiência da equipe com orquestração de containers e clusters distribuídos.

Dificuldades encontradas:

- **Complexidade da arquitetura:** Inicialmente, assumimos erroneamente que seria necessário containerizar toda a aplicação, criar imagens Docker customizadas,



configurar container registries e desenvolver arquivos de configuração Kubernetes (YAML) complexos para o deployment. Posteriormente descobrimos que era possível executar o código diretamente em nós do cluster, evitando parte da complexidade de containerização.

- **Problemas de conectividade:** Mesmo com a abordagem simplificada, enfrentamos dificuldades significativas na comunicação entre a API do cluster Kubernetes e as sessões distribuídas do Spark e Dask.
- **Logs complexos:** Os logs de erro gerados eram de difícil interpretação para nossa equipe, com mensagens técnicas relacionadas a networking, service discovery e configurações de cluster que não dominávamos.

Esta etapa consumiu tempo desproporcional ao planejado, comprometendo outras atividades do projeto. A falta de expertise prévia em DevOps e infraestrutura de containers evidenciou a necessidade de maior preparação técnica da equipe nessas áreas.

Criação inadequada de clientes/sessões: Inicialmente, optamos por processar um dataset por vez em todas as ferramentas e concatenar os resultados. Durante a implementação, utilizamos um loop que criava novos clientes e sessões (Spark Session, Dask Client, etc.) a cada iteração do processamento. Essa abordagem gerou dois problemas críticos:

- **Esgotamento de portas:** Após várias execuções, as portas dos clientes se esgotavam, impedindo a criação de novas conexões.
- **Memory leak:** O acúmulo de sessões não finalizadas adequadamente causava vazamentos de memória, resultando na interrupção do programa.

Refatoramos o código utilizando o padrão Singleton para manter uma única sessão durante toda a execução do benchmark. Desta forma, todas as ferramentas compartilham a mesma instância de cliente/sessão, eliminando os problemas de recursos.

Adicionalmente, implementamos a proteção `if __name__ == "__main__":` nos módulos para evitar execução de código durante a importação. Sem essa proteção, ao importar módulos de diferentes ferramentas, códigos de inicialização eram executados automaticamente, causando conflitos entre as bibliotecas e inicializações desnecessárias.

Criação de index: A estratégia de armazenar datasets em um bucket GCS e salvar resultados em outro bucket trouxe desafios técnicos significativos relacionados ao acesso de dados remotos.

Cometemos um erro na implementação da primeira versão do benchmarking. Percebemos que os resultados do databricks mostravam resultados bastante divergentes dos nossos, especialmente para as tasks "join"e "join count", que não pareciam fazer sentido. Analisando, vimos que os parquets disponibilizados não possuem a coluna "index"utilizada para fazer os joins (presentes no teste do databricks pois eles já haviam preprocessado os parquets). Até então, estávamos fazendo os joins usando a coluna "passager count"como referência. Tivemos que adicionar a coluna "index"e modificar as task para utilizá-la.

Dilema arquitetural com Koalas/PySpark: Enfrentamos incertezas sobre qual API utilizar para processamento com Koalas: a sintaxe nativa do PySpark (spark.sql) ou a interface pandas-like (pyspark.pandas).

Análise das abordagens:

- **PySpark nativo:** Oferece lazy evaluation, uso de RDDs e fault tolerance, resultando em maior eficiência de memória para datasets grandes.
- **PySpark.pandas:** Carrega o dataset na memória múltiplas vezes durante as operações, causando maior consumo de recursos.

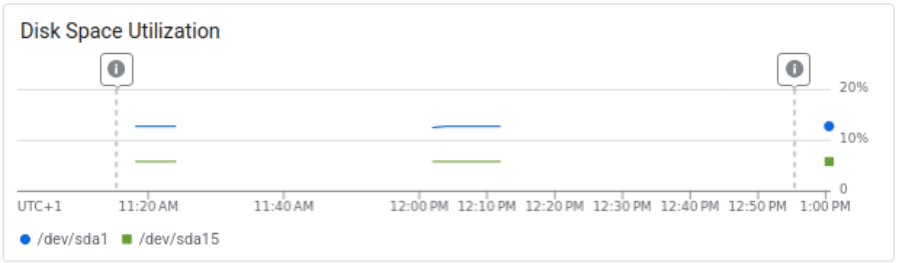
Decisão de implementação: Inicialmente testamos um approach híbrido, processando com Spark nativo e convertendo resultados para pyspark.pandas. Posteriormente, optamos por utilizar exclusivamente pyspark.pandas para manter a a coesão do benchmark, respeitando os princípios de cada ferramenta. Esta decisão resultou em limitações de memória ao executar Koalas em máquinas locais, exigindo o desenvolvimento de um script de redução de datasets para viabilizar os testes com recursos limitados.

Treino do modelo com spill para o disco: Durante o treino do modelo de machine learning, mesmo com 52Gb de RAM, ao utilizar todos os datasets de 2011 e 2012 com modin + dask, enfrentamos problemas de recursos na máquina virtual. A RAM chegava próxima ao uso máximo e o dask iniciava o processo de spill para o disco. Podemos ver na figura abaixo, em diversas vezes tentamos utilizar um dataframe com todos os dados de treino, mas o uso de RAM subitamente desaparece e o consumo do disco aumenta. Isso gerou um processamento extremamente lento.

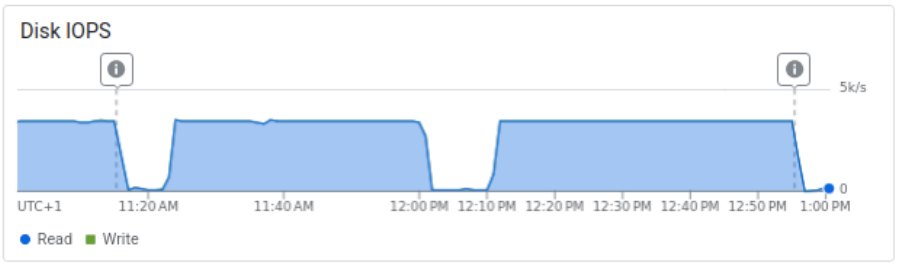
As métricas de Disk IOPS cresceram absurdamente devido aos dados de processamento precisarem de estar no disco, assim como o Disk throughput. Os dados de disk space utilization não possuem logs durante o uso intensivo do disco, o que não nos permite garantir com certeza que houve spill, mas em logs de erros, o spill foi mencionado, portanto, é o mais provável.

Benchmark errado para Joblib: Executamos o benchmarking para joblib, e achamos que o joblib era uma ferramenta extremamente poderosa e superior as outras pelo


Memory Utilization
100%
0
UTC+1
11:20 AM
11:40 AM
12:00 PM
12:10 PM
12:20 PM
12:30 PM
12:40 PM
12:50 PM
1:00 PM
Memory Utilization


Disk Space Utilization
20%
10%
0
UTC+1
11:20 AM
11:40 AM
12:00 PM
12:10 PM
12:20 PM
12:30 PM
12:40 PM
12:50 PM
1:00 PM
/dev/sda1
/dev/sda15


Disk Throughput
200MiB/s
0
UTC+1
11:20 AM
11:40 AM
12:00 PM
12:20 PM
12:40 PM
1:00 PM
Read
Write


Disk IOPS
5k/s
0
UTC+1
11:20 AM
11:40 AM
12:00 PM
12:10 PM
12:20 PM
12:30 PM
12:40 PM
12:50 PM
1:00 PM
Read
Write

baixíssimo tempo de processamento. Entretanto, ao rodarmos o profiler, percebemos o seguinte:

```
         2 function calls in 0.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/joblib/parallel.py:666(delayed_function)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

As funções joblib estava retornando apenas objetos delayed, e não resultados reais, Portanto, o tempo de processamento era 0, por isso a eficiência tão grande. Tivemos que modificar o código joblib para que retornasse resultados reais.

6 Benchmarking

6.1 Objetivo

Reproduzir o estudo de desempenho conforme descrito no blog, utilizando o dataset de corridas de táxi de Nova York nas ferramentas de processamento: joblib, dask, dask+modin, spark, pyspark.pandas.

6.2 Metodologia Benchmarking

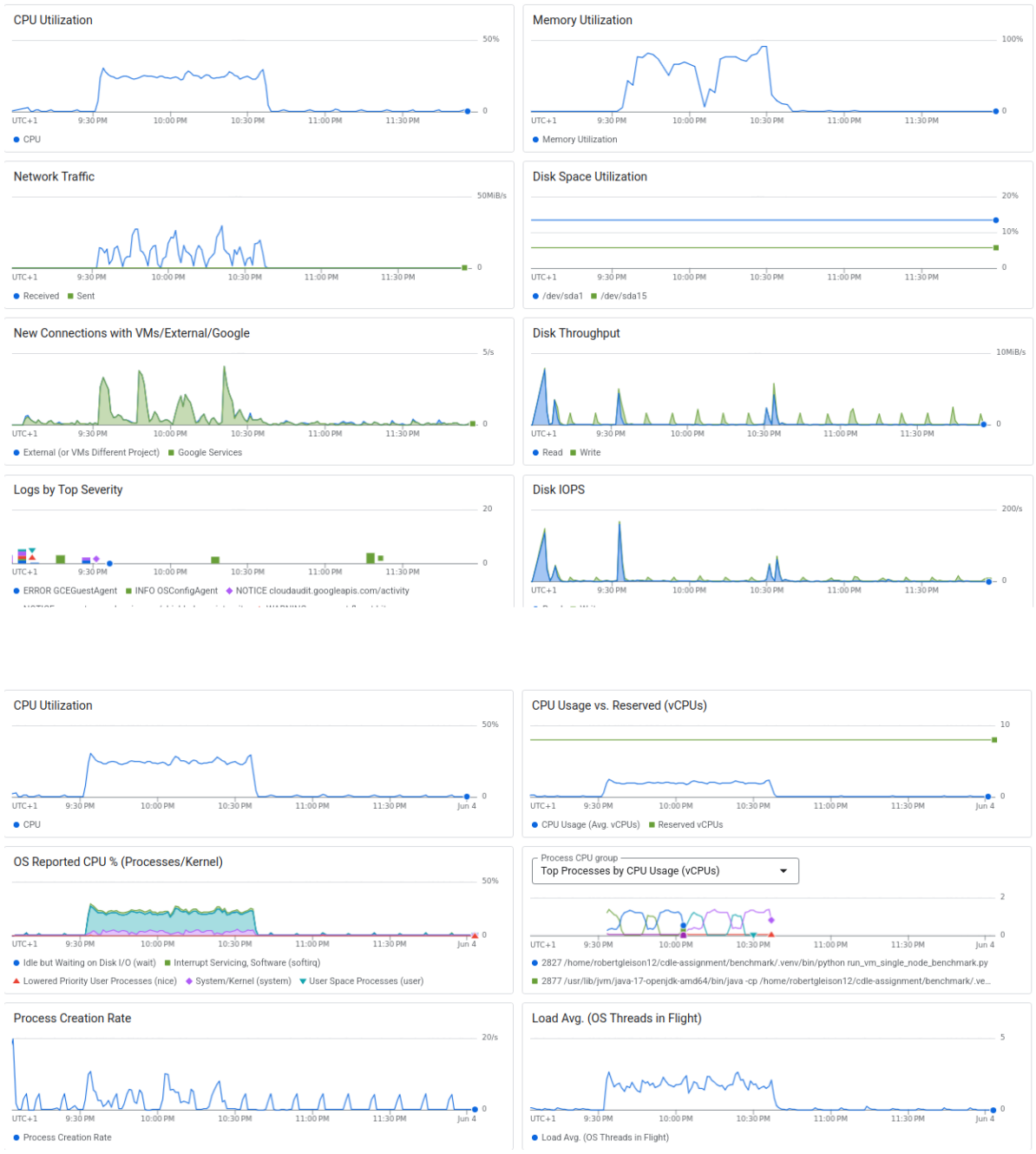
Realizamos o benchmarking apenas dos datasets de 2009 e 2010 devido aos motivos citados em 5. Foram feitos benchmarkings utilizando as seguintes variações (especificações em 3):

- **1 worker single node (VM)**
- **8 workers single node (VM)**
- **1 worker in 3 nodes (Cluster)**

Procuramos igualar os recursos para cada ferramenta. Dentro da VM, mesmo que com 52Gb RAM, limitamos para 40Gb RAM que seria dividida entre os workers (1 ou 8). o limite de memória é feito pelos clientes de cada ferramenta, exceto joblib que apenas consegue limitar o número de workers (parallel tasks).

Para o cluster, optamos por criar 3 nós distintos que possuem menor capacidade que a VM, mas juntos, possuem recursos semelhantes. Cada nó foi limitado a 13Gb RAM, totalizando 39Gb. Importante ressaltar também que a ferramenta Joblib não foi testada no cluster por não suportar processamento distribuído.

Não houve casos onde se necessitou do uso total de RAM, spill para o disco ou redução de datasets. O benchmarking usando 8 workers teve a média de tempo de execução ligeramente menor que usando 1 worker, mas na prática, foram similáres, portanto, optamos por usar o profiler com 1 worker apenas.



Na figura acima, podemos ver os picos de uso de recursos de nossa máquina virtual. Chegamos a utilizar até 92% de toda a nossa memória RAM para processamentos complexos. Tivemos picos de tráfico de network e disk throghput devido a estarmos lendo e escrevendo arquivos entre nossa máquina virtual e buckets.

Embora nossa máquina virtual tenha 8 vCPUS, em nosso processamento, chegamos a utilizar no máximo 2 vCPUs. Isso mostra que existiram alguns gargalos durante o benchmarking ou as tarefas foram divididas de um modo que não permitiram a extração máxima do potencial de nossas ferramentas que utilizam até 8 workers. Não conseguimos identificar o motivo e corrigir a tempo.

7 Single Node Benchmarking (1 worker):

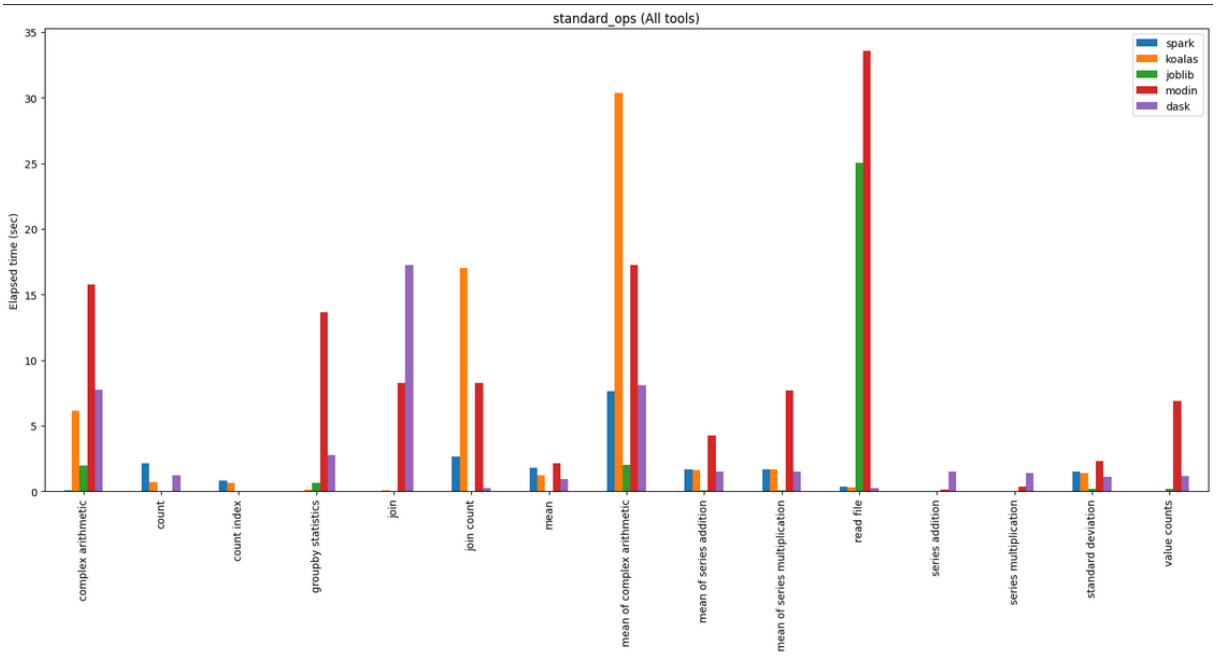


Figura 1: Standard Ops - Single Node 1 worker

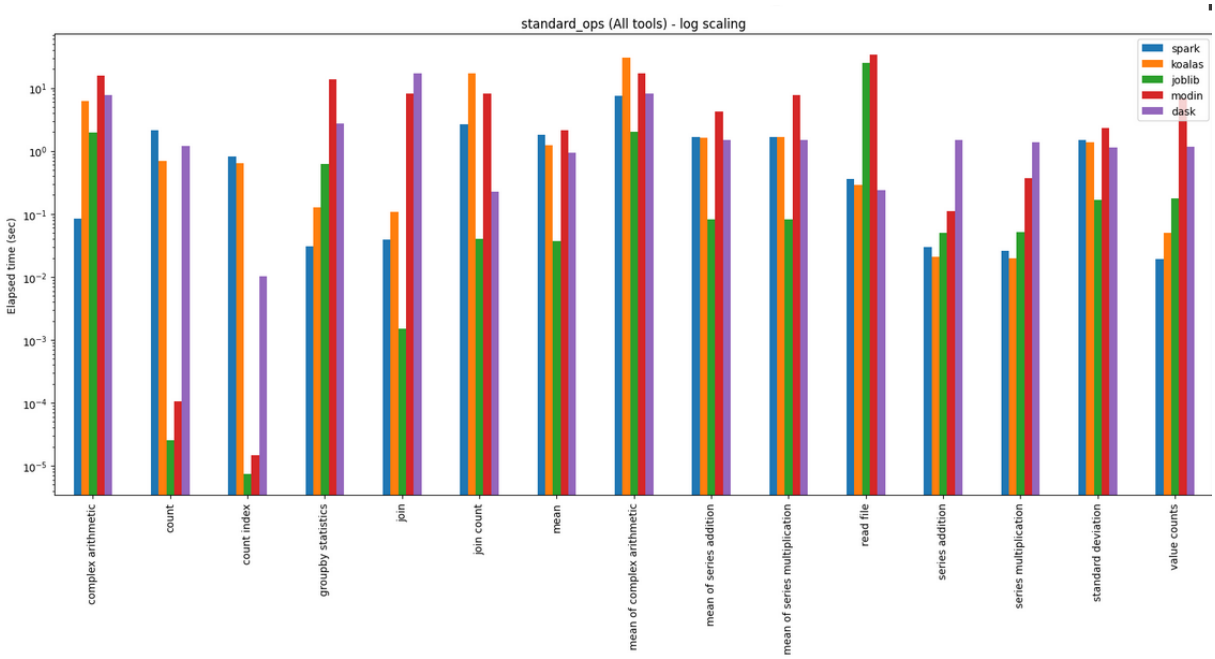


Figura 2: Standard Ops Scaled - Single Node 1 worker

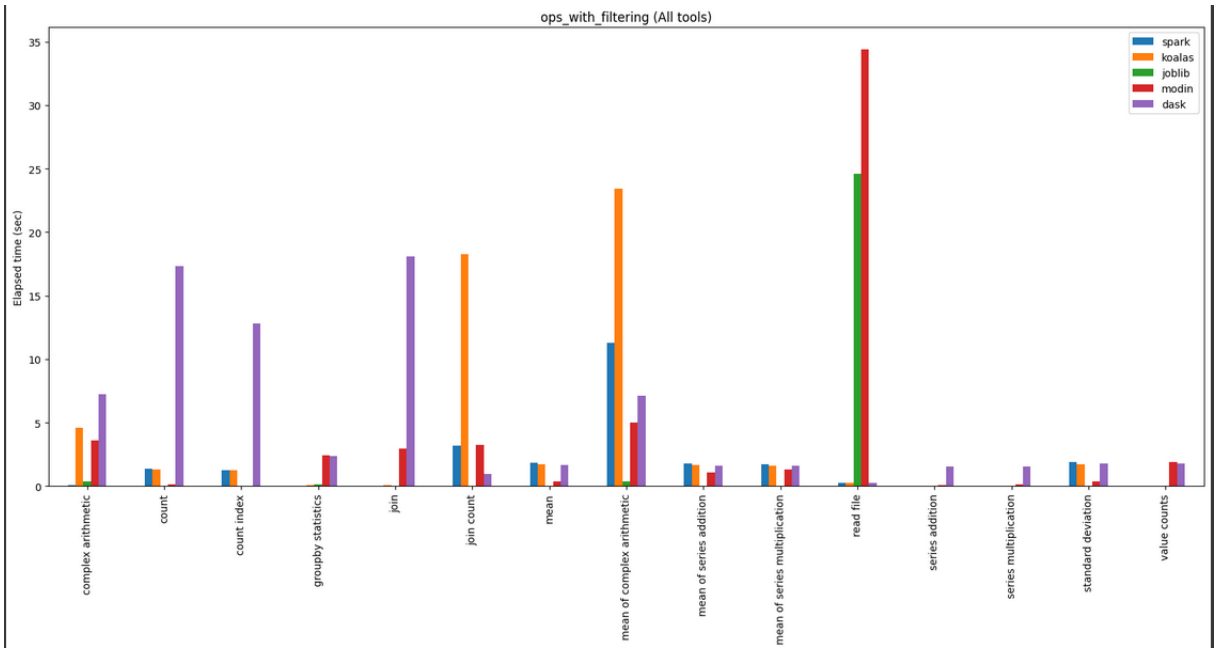


Figura 3: Standard Ops with Filtering- Single Node 1 worker

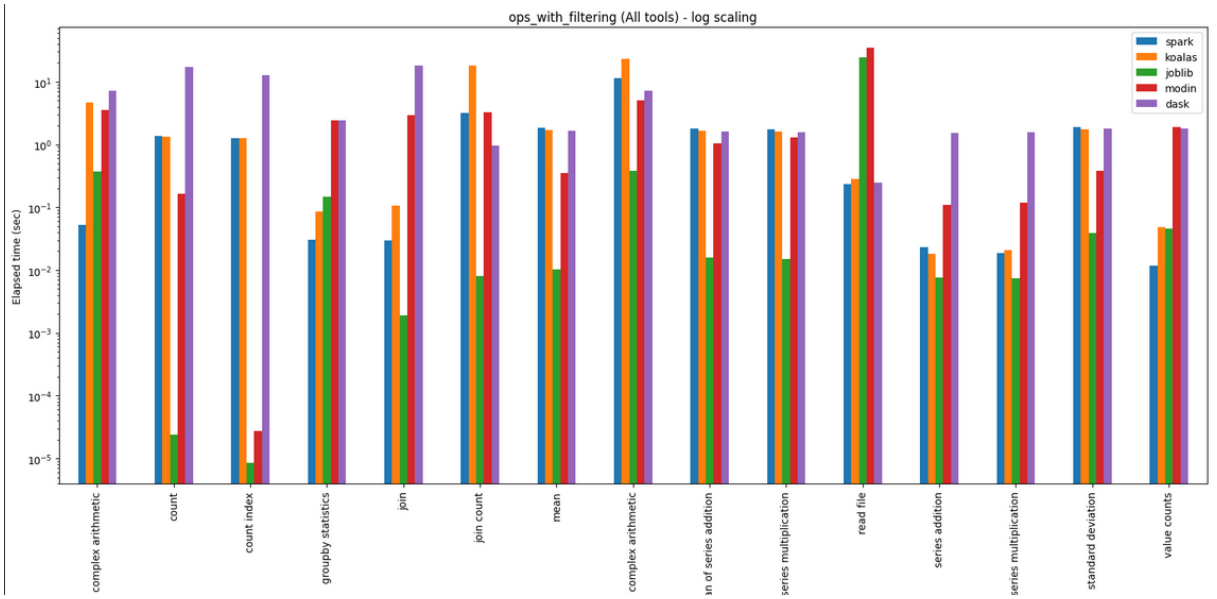


Figura 4: Standard Ops with Filtering Scaled- Single Node 1 worker

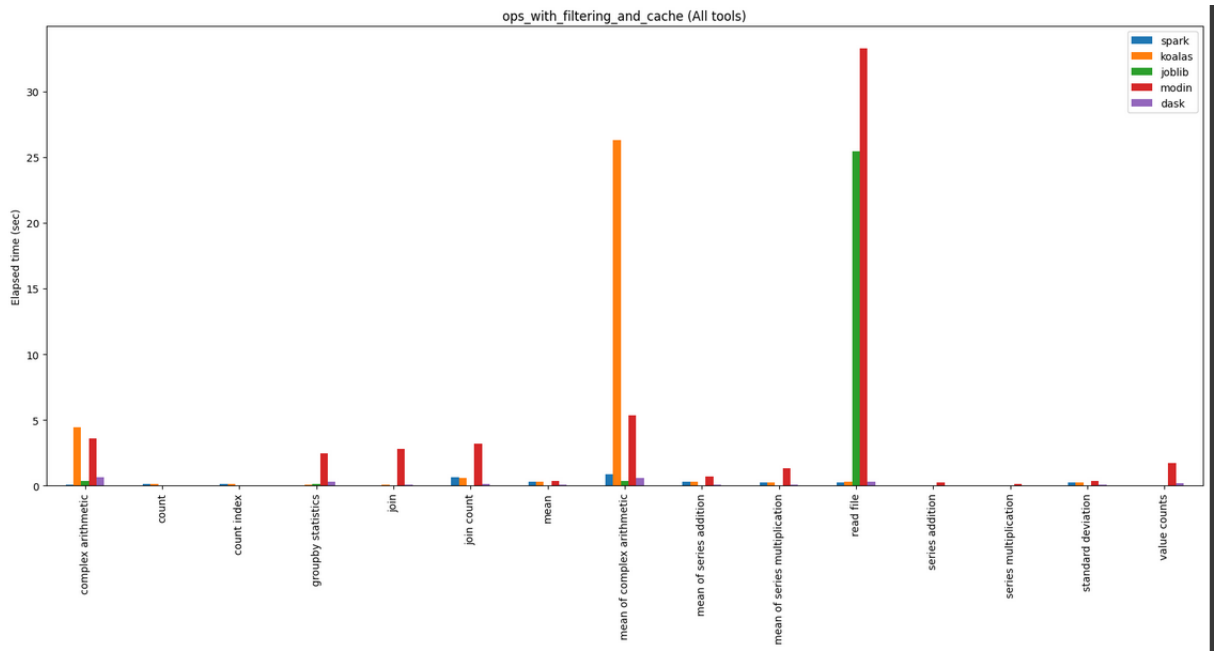


Figura 5: Standard Ops with Filtering and Cache - Single Node 1 worker

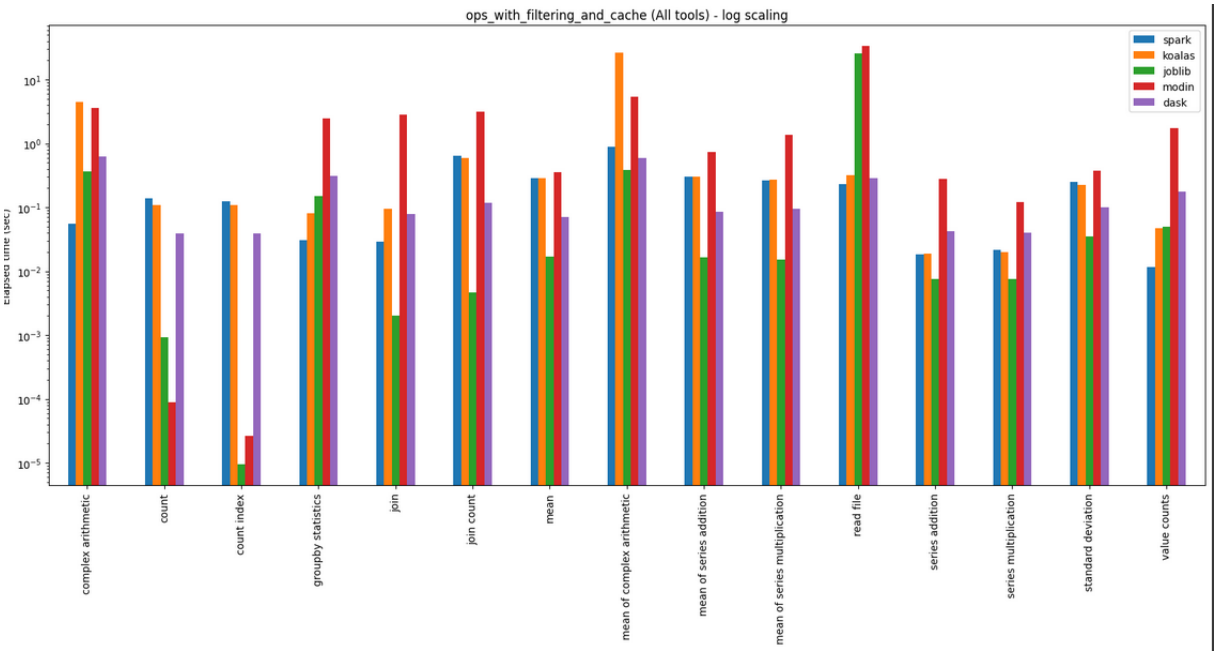


Figura 6: Standard Ops with Filtering and Cache Scaled- Single Node 1 worker

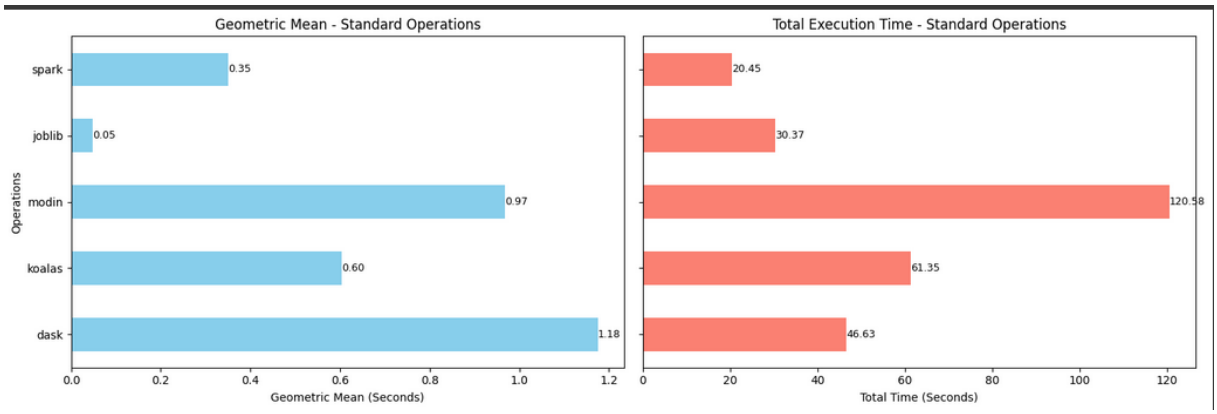


Figura 7: Aggregated metrics - Standard Operations

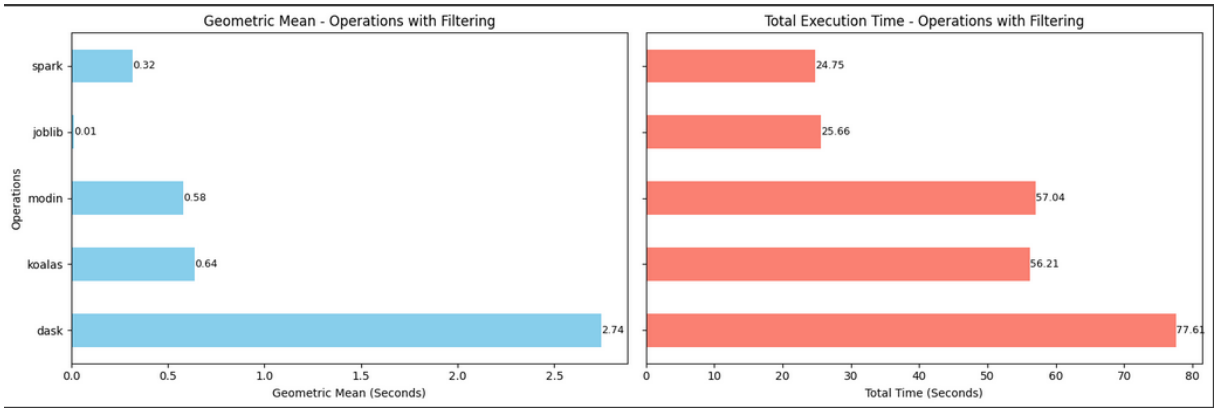


Figura 8: Aggregated metrics - Operations with Filtering

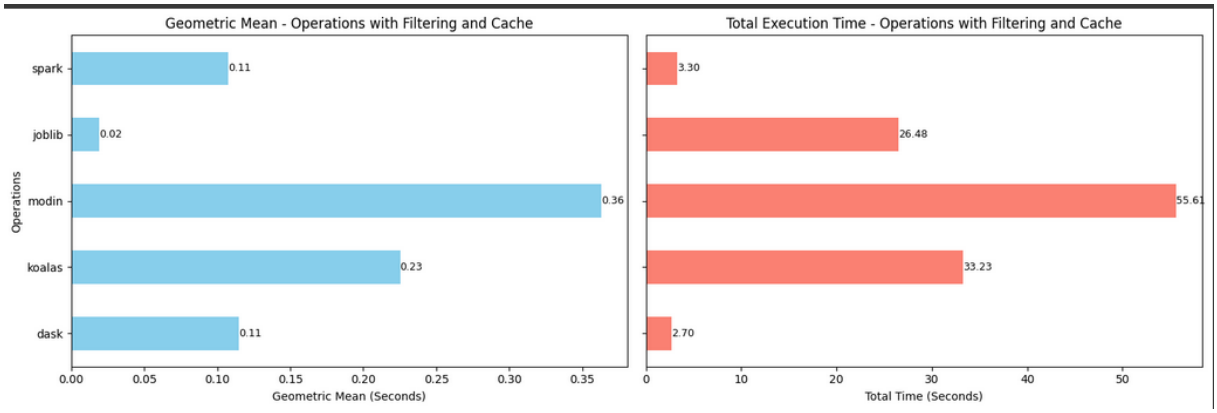


Figura 9: Aggregated metrics - Operations with Filtering and Cache

8 Single Node Benchmarking (8 workers):

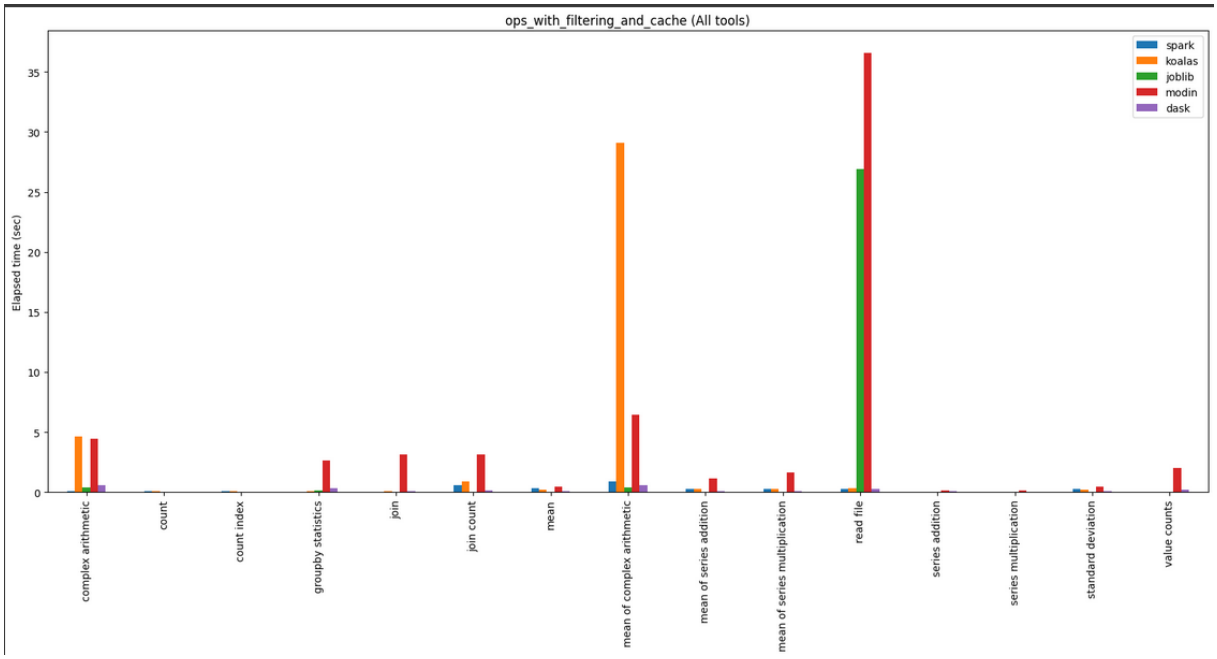


Figura 10: Standard Ops - Single Node 1 worker

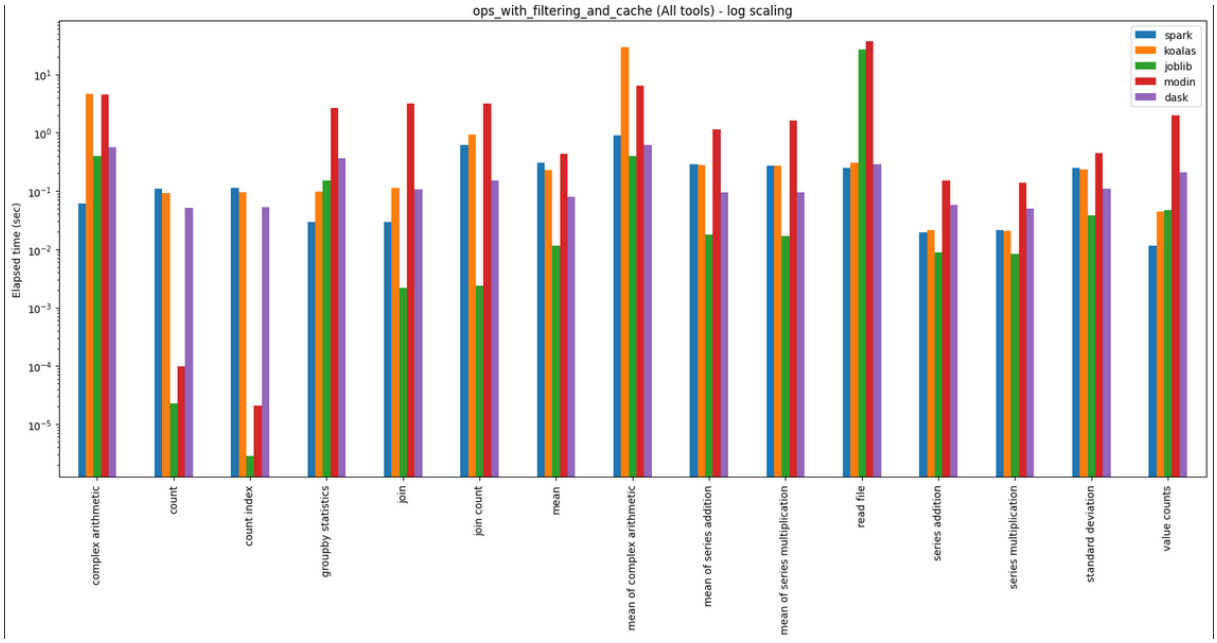


Figura 11: Standard Ops Scaled - Single Node 1 worker

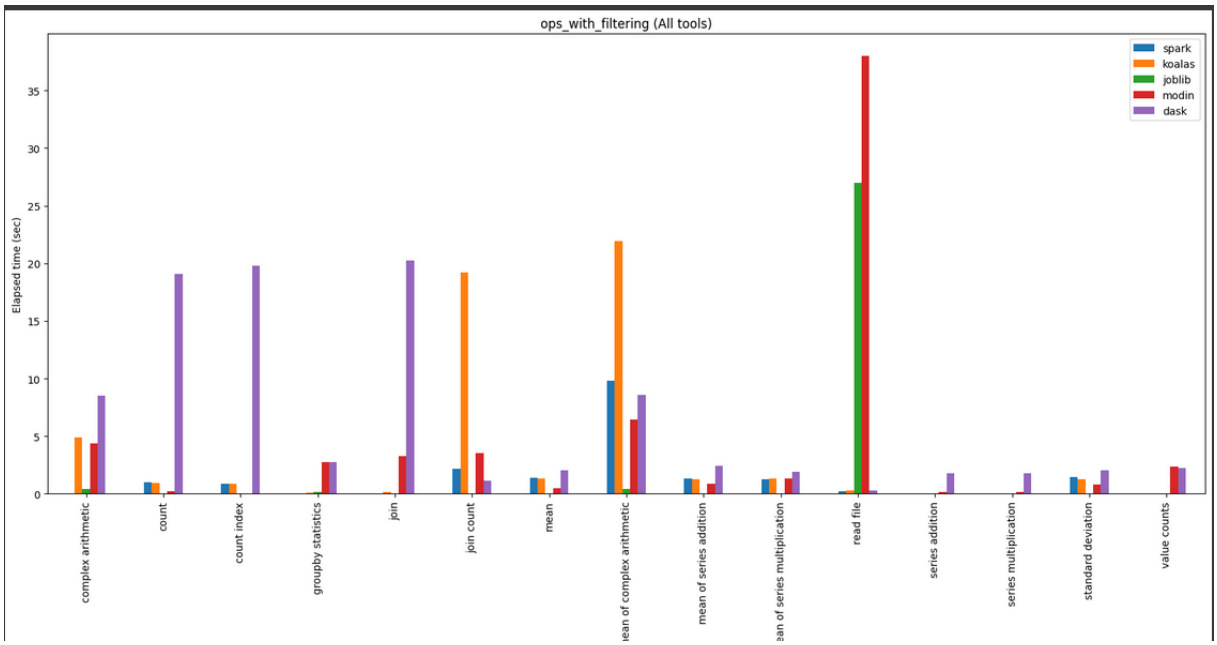


Figura 12: Standard Ops with Filtering- Single Node 1 worker

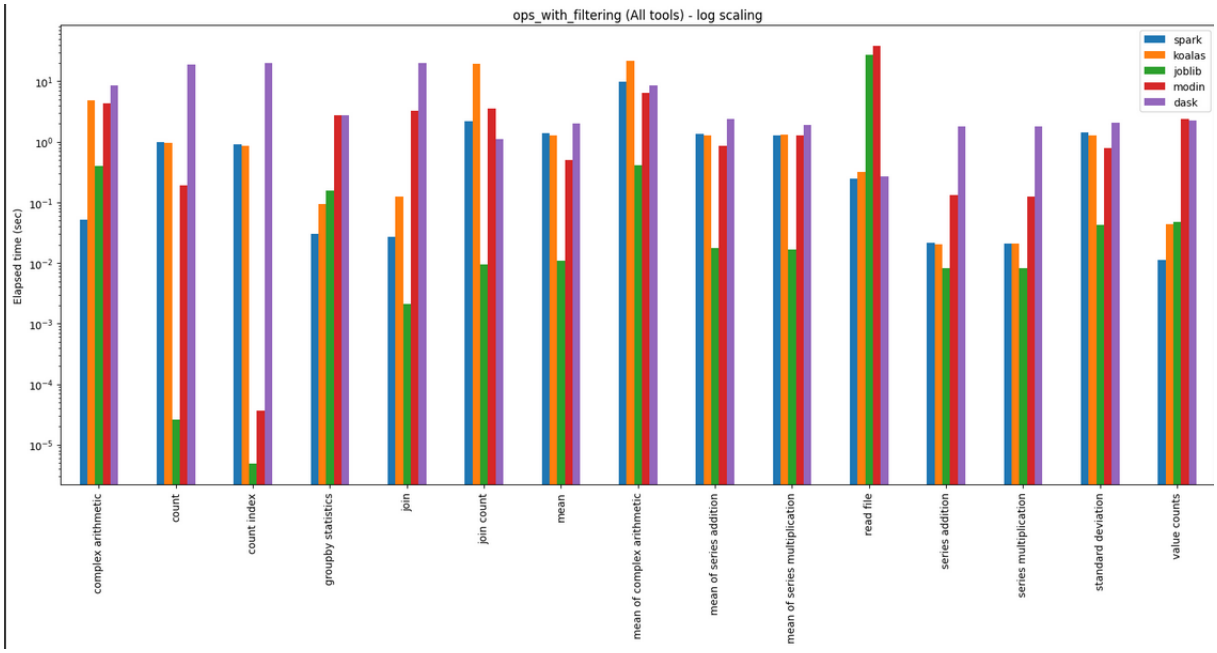


Figura 13: Standard Ops with Filtering Scaled- Single Node 1 worker

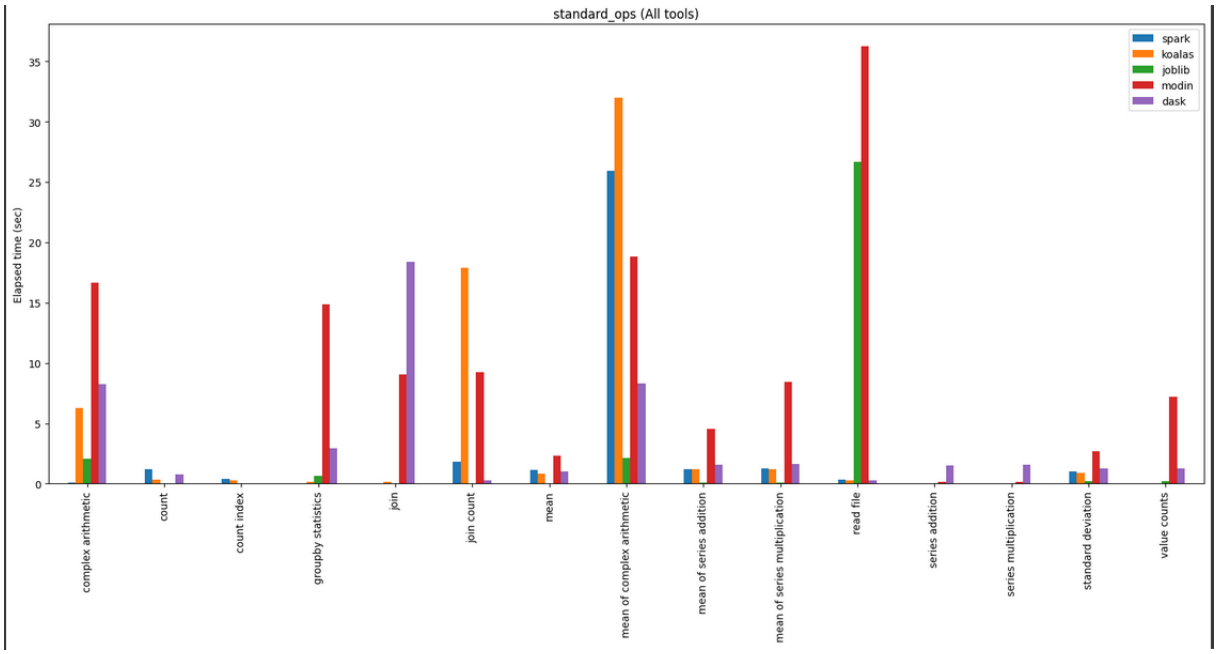


Figura 14: Standard Ops with Filtering and Cache - Single Node 1 worker

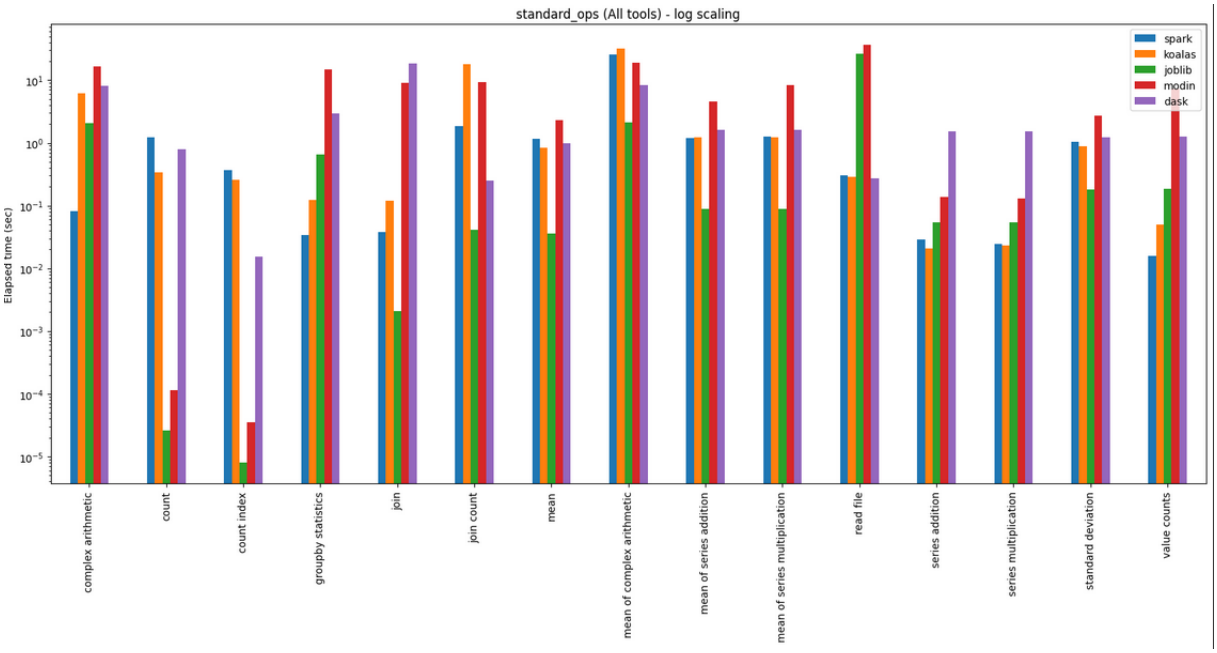


Figura 15: Standard Ops with Filtering and Cache Scaled- Single Node 1 worker

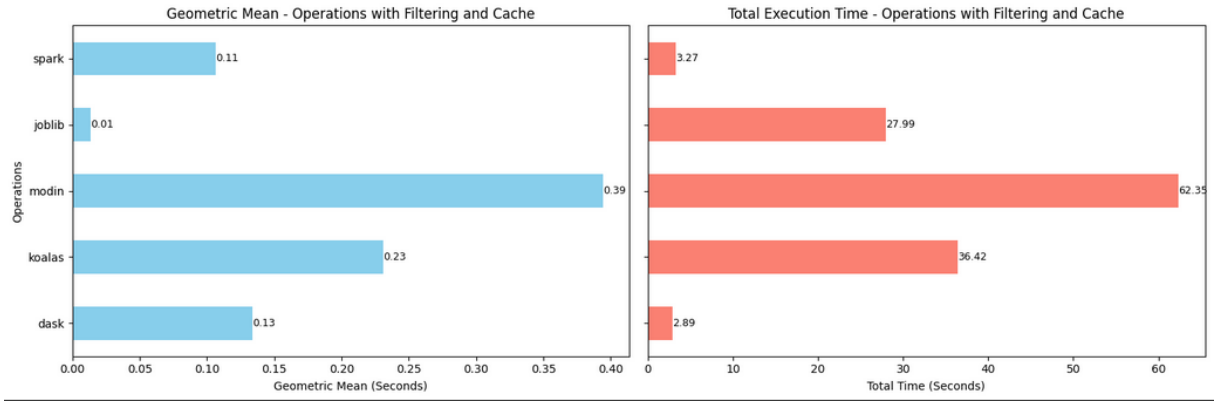


Figura 16: Aggregated metrics - Standard Operations

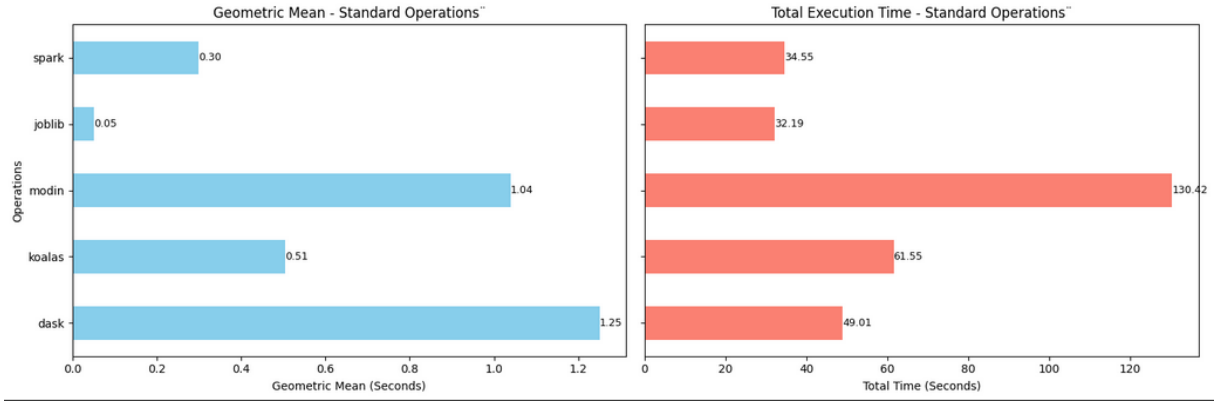


Figura 17: Aggregated metrics - Operations with Filtering

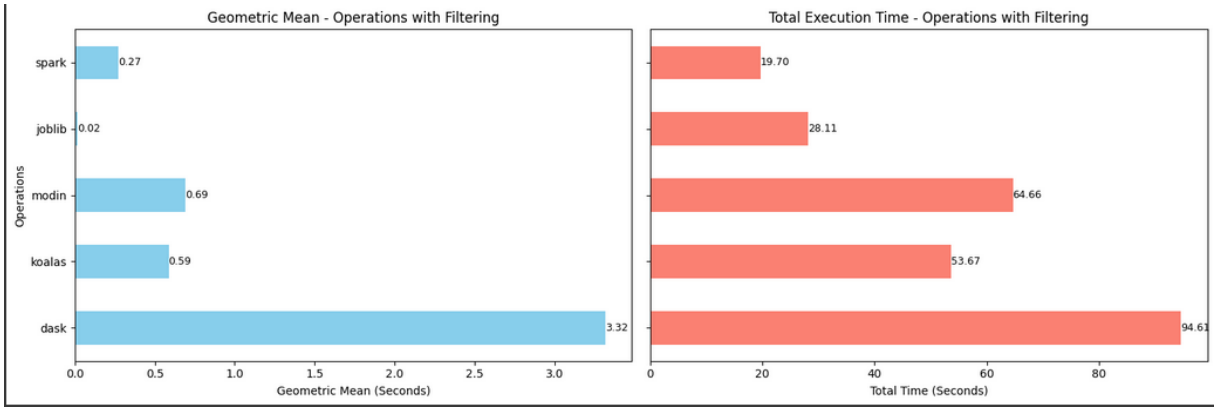


Figura 18: Aggregated metrics - Operations with Filtering and Cache

9 Single Node Benchmarking (1 worker x 8 workers):

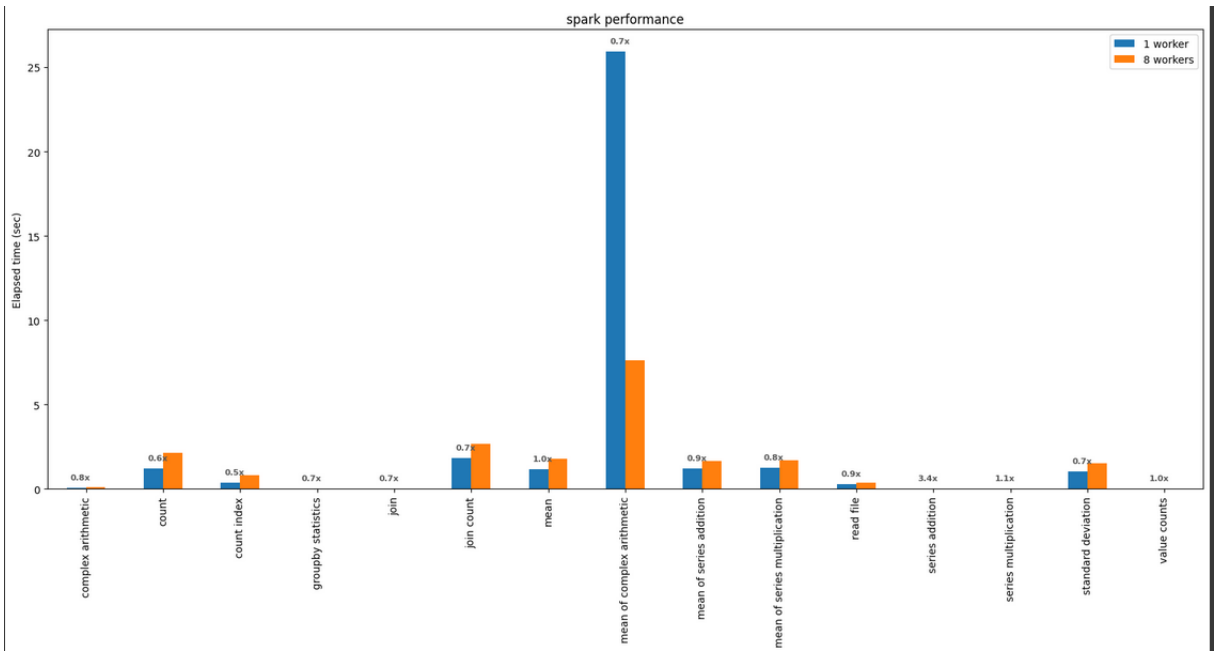


Figura 19: Spark comparison 1 worker x 8 workers

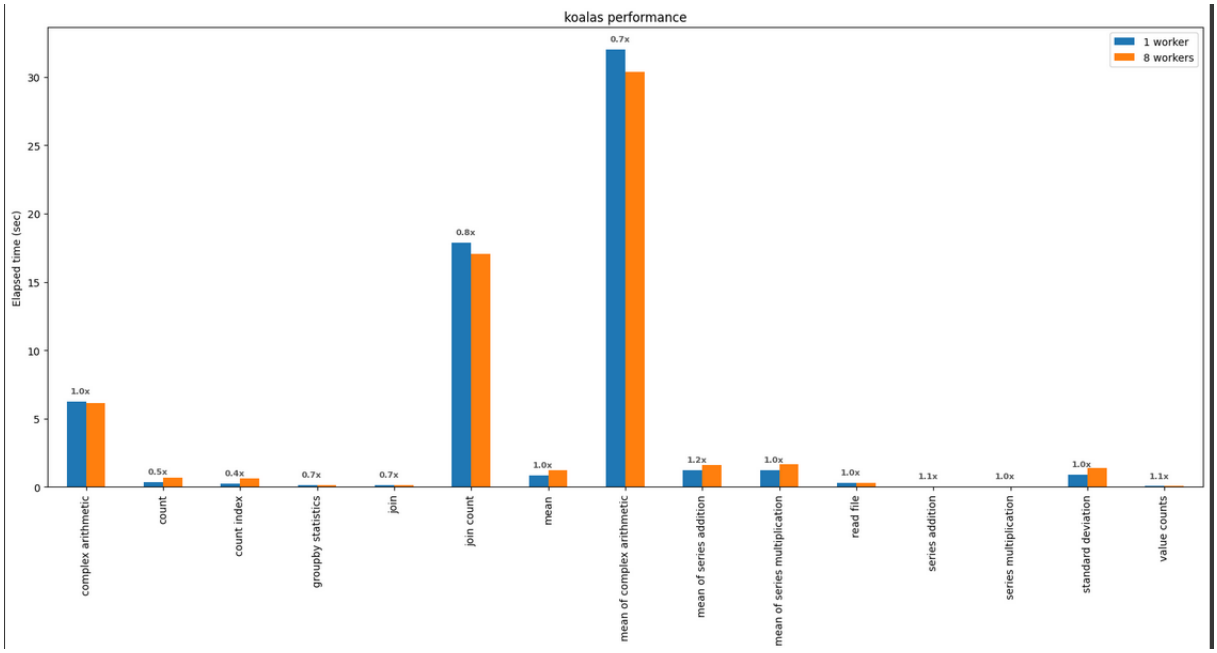


Figura 20: Koalas comparison 1 worker x 8 workers

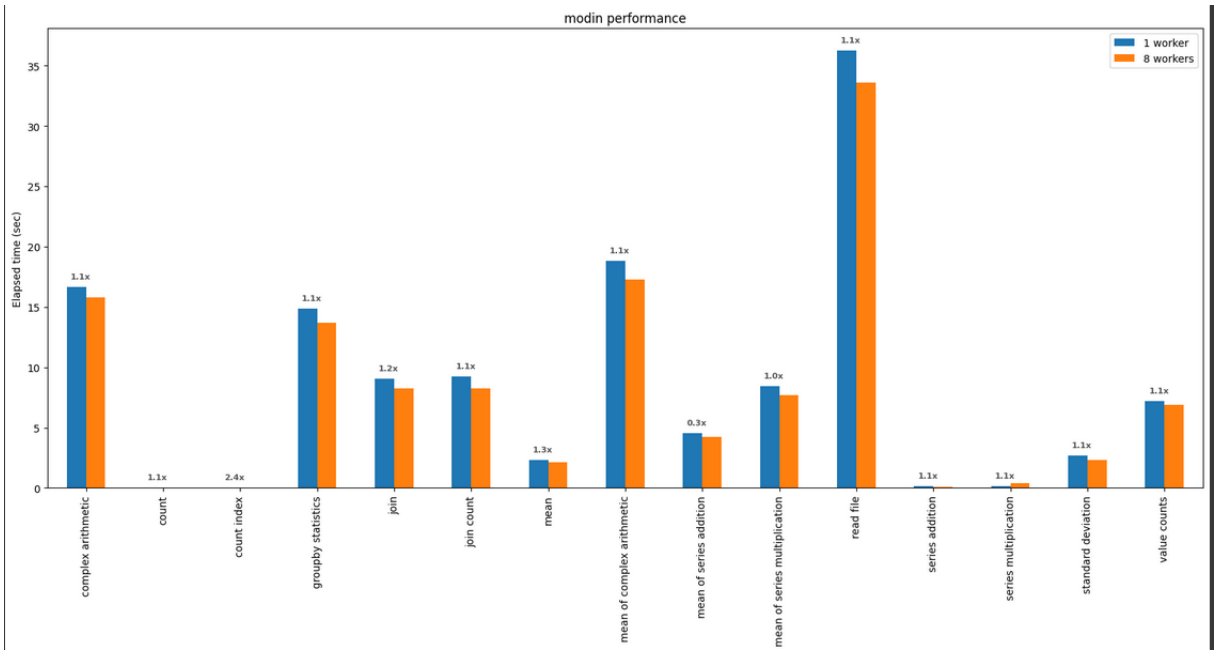


Figura 21: Modin comparison 1 worker x 8 workers

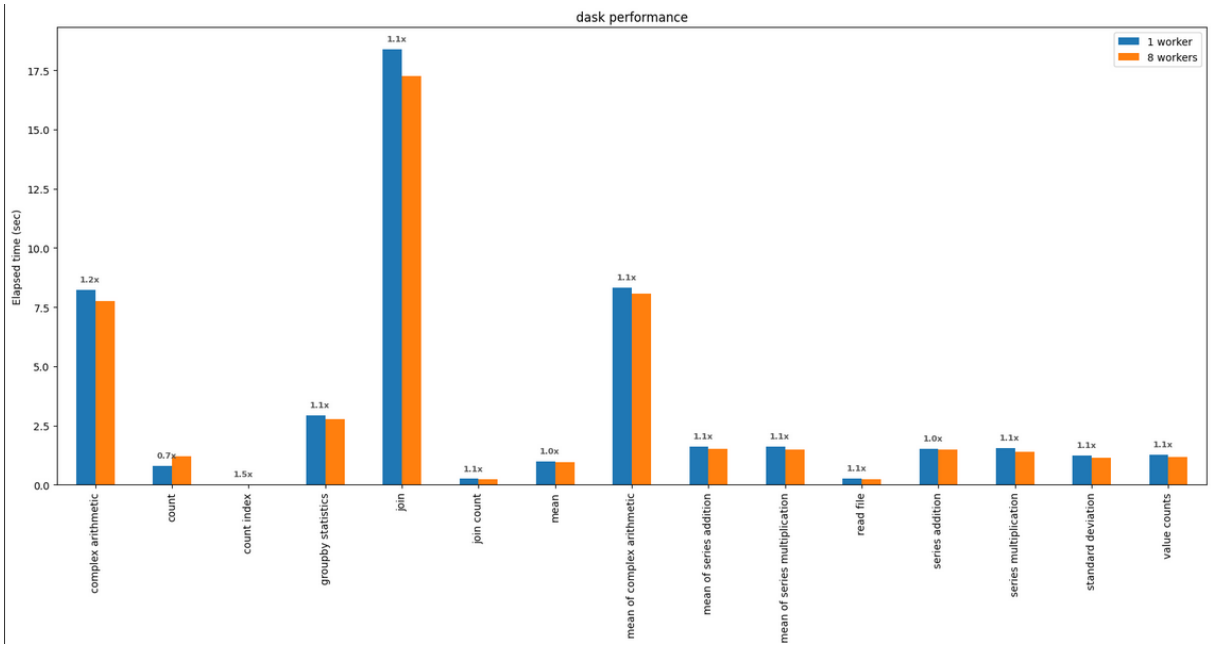


Figura 22: Dask comparison 1 worker x 8 workers

Para nosso caso de uso, o processamento com 8 workers resultou em tempos menores na maioria das tarefas. Entretanto, as tarefas complexas e demoradas apresentaram os maiores benefícios, enquanto operações simples como join_count e count_index tiveram tempos de execução maiores com 8 workers. Esse comportamento ocorre devido ao overhead de sincronização e criação dos workers, que se torna significativo quando comparado ao tempo de execução de tarefas rápidas.

10 Distributed Benchmarking (1 worker):

Não conseguimos rodar os benchmarkings no cluster devido à erros de conexão entre API do kubectl e as sessions spark/dask.

Optamos por deixar uma comparação apenas entre Single Node com 1 worker e com 8 workers como alternativa. Pedimos desculpas pelo inconveniente.

11 Análise de resultados:

11.1 Spark

Pontos Positivos:

- Excelente para operações complexas
- Boa performance em joins e agregações
- Escalabilidade robusta para datasets grandes

Pontos Negativos:

- Alto overhead em operações simples (count, mean)
- Tempo de inicialização elevado
- Complexidade de configuração para dados pequenos

11.2 Koalas

Pontos Positivos:

- Boas performances overall

Pontos Negativos:

- Performance inconsistentes em operações simples
- Overhead do Spark sem benefícios completos

11.3 Joblib

Pontos Positivos:

- Melhores tempos de execução
- Baixo overhead para operações simples
- Fácil paralelização de funções Python

Pontos Negativos:

- Leitura de arquivos ineficiente (Explorado no profiler Modin)
- Performance inconsistente em operações complexas

11.4 Modin

Pontos Positivos:

- Boa performance em tarefas simples e paralelizaveis

Pontos Negativos:

- Pior performance em read file e altos tempos de execução.
- Alto overhead em operações simples
- Leitura de arquivos ineficiente (Explorado no profiler)

11.5 Dask

Pontos Positivos:

- Performance razoável em todas operações
- Baixo overhead em tarefas simples

Pontos Negativos:

12 Profilers

O profiling foi feito utilizando a lib cProfile e apenas nas operações standard com 1 worker. O profile foi feito em ambiente local de testes (computador pessoal dos integrantes do grupo) com as seguintes especificações: 3

Todos os arquivos de profiling estão no repositório github:

`https://github.com/RobertGleison/cdle-assignment`

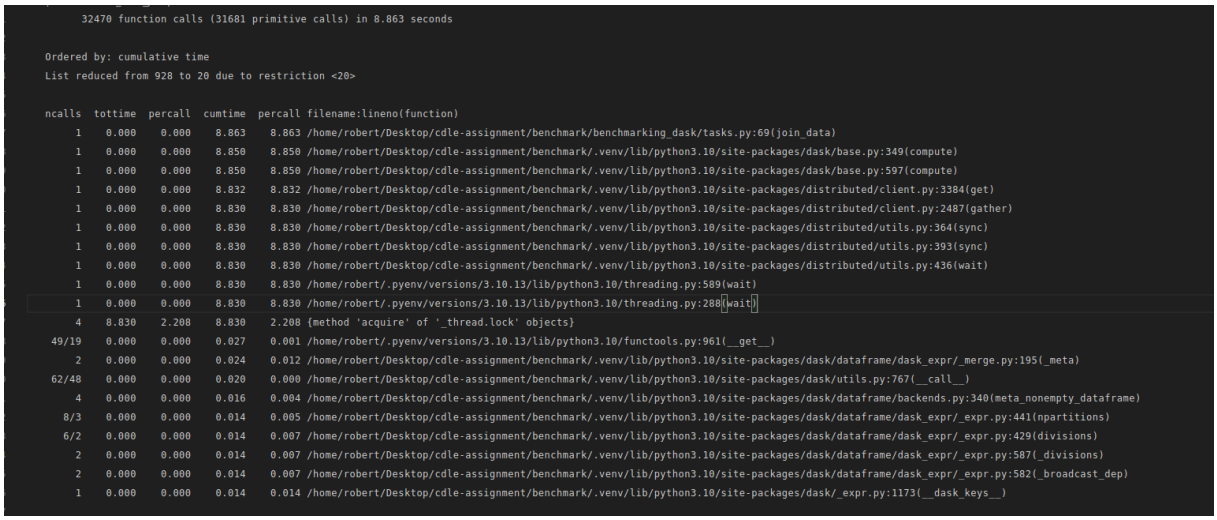
Não iremos falar de todos os profiling de todas as tasks, de todas as ferramentas, de todos modos (com ou sem cache ou filtro) pois isso levaria muito tempo e esforço. Apenas iremos abordar os pontos que nos chamaram a atenção e que podem ser otimizados.

Importante mencionar que o profiling foi feito apenas das tasks, não contando o tempo gasto fora delas. Para processamento em Spark e em Koalas, o tempo que se gastar para iniciar as task são muito superiores à joblib ou dask por exemplo.

- **Joblib:** O joblib possui suporte apenas para single node. O joblib apresenta tempos de execução baixos em comparação com as outras ferramentas devido ao seu baixo overhead e processamento direto dos dados sem chamadas adicionais. Para nosso caso de uso single node com um dataset de 500mb, pode ser adequado. Como estamos usando 1 worker e executando dataframes que retornam seus valores nas tasks, na prática o profiler está mostrando os resultados do pandas (não utilizamos cache e ao aumentar o número de tarefas paralelas, a diferença no tempo de execução não foi significativa).

- **Dask:** Podemos ver que o dask.distributed adiciona overhead em algumas computações. Para um dataset de 500MB, a tarefa 'join data', responsável por dar join entre 2 datasets, demora 8.863 segundos. Mas quase 8.83 segundos foram utilizados por: {`method 'acquire' of '_thread.lock' objects`} Isto indica que a

  thread principal está bloqueada à espera que os workers distribuídos completem a computação. O resultado é similar tanto usando 1 worker como 8 workers.



```
      32470 function calls (31681 primitive calls) in 8.863 seconds

Ordered by: cumulative time
List reduced from 928 to 20 due to restriction <20>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    8.863    8.863 /home/robert/Desktop/cdle-assignment/benchmark/benchmarking_dask/tasks.py:69(join_data)
     1    0.000    0.000    8.850    8.850 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/base.py:349(compute)
     1    0.000    0.000    8.850    8.850 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/base.py:597(compute)
     1    0.000    0.000    8.832    8.832 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/client.py:3384(get)
     1    0.000    0.000    8.830    8.830 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/client.py:2487(gather)
     1    0.000    0.000    8.830    8.830 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/utils.py:364(sync)
     1    0.000    0.000    8.830    8.830 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/utils.py:393(sync)
     1    0.000    0.000    8.830    8.830 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/utils.py:436(wait)
     1    0.000    0.000    8.830    8.830 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/threading.py:589(wait)
     1    0.000    0.000    8.830    8.830 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/threading.py:288(wait)
     4    8.830    2.208    8.830    2.208 {method 'acquire' of '_thread.lock' objects}
 49/19    0.000    0.000    0.027    0.001 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/functools.py:961(__get__)
     2    0.000    0.000    0.024    0.012 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/dataframe/dask_expr/_merge.py:195(_meta)
 62/48    0.000    0.000    0.020    0.000 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/utils.py:767(__call__)
     4    0.000    0.000    0.016    0.004 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/dataframe/backends.py:340(meta_nonempty_dataframe)
   8/3    0.000    0.000    0.014    0.005 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/dataframe/dask_expr/_expr.py:441(npartitions)
   6/2    0.000    0.000    0.014    0.007 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/dataframe/dask_expr/_expr.py:429(divisions)
     2    0.000    0.000    0.014    0.007 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/dataframe/dask_expr/_expr.py:587(_divisions)
     2    0.000    0.000    0.014    0.007 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/dataframe/dask_expr/_expr.py:582(_broadcast_dep)
     1    0.000    0.000    0.014    0.014 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/dask/_expr.py:1173(__dask_keys__)
```

Figura 23: Dask join operation profiling

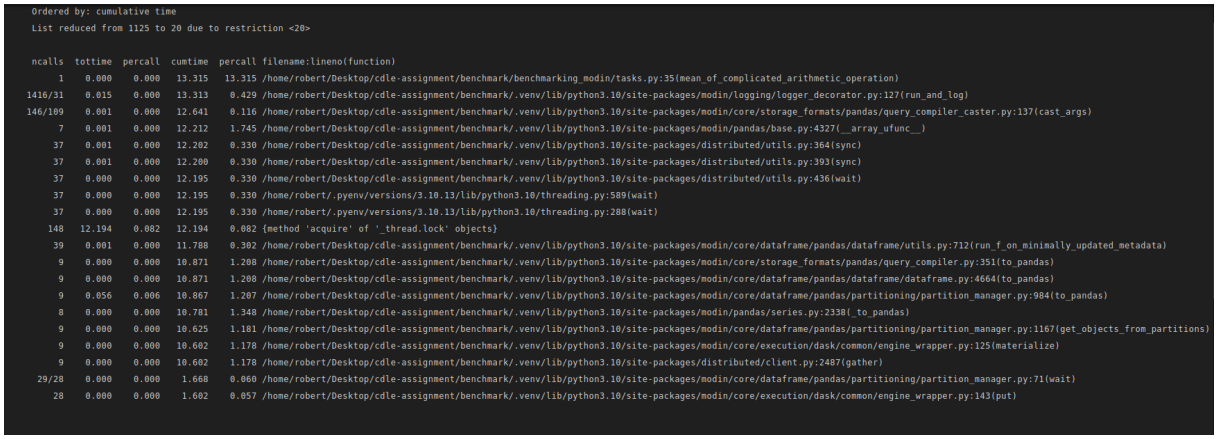
Para todas as outras tasks utilizadas pelo Dask, o block por workers parece ser a maior fonte de espera. Portanto, podemos presumir que para datasets pequenos e médios, talvez o dask possa ser substituído por alguma ferramenta que não acrescente overhead. Entretanto, para task simples como contar número de linhas, ler arquivos, adicionar colunas, o dask se saiu bem e pouco overhead foi adicionado. Lembrando que dask possui lazy evaluation, portanto tarefas simples como count de linhas ou tasks que não precisem carregar um dataframe em memória são bem eficientes, pois o mesmo utiliza os metadados do parquet sempre que possível.

- **Modin:** Para o modin, vemos que também possui overhead pois é uma camada de abstração a mais (estamos usando modin + dask). Vemos que para a task de processamento aritmético complexo, o maior overhead vem do client da engine Dask. Aém disso, múltiplas chamadas são feitas para `distributed.client.gather`, frequentes materializações e pelo menos 8 chamadas para `to_pandas()`, responsáveis por conversões de pandas para dask.

  Como este método de cálculo aritmético possui multiplas linhas de execução, se é esperado que múltiplas conversôes sejam feitas. talvez exista uma forma mais otimizada de organizar o algorítmo.

  Olhando agora para a função de `read_file`:

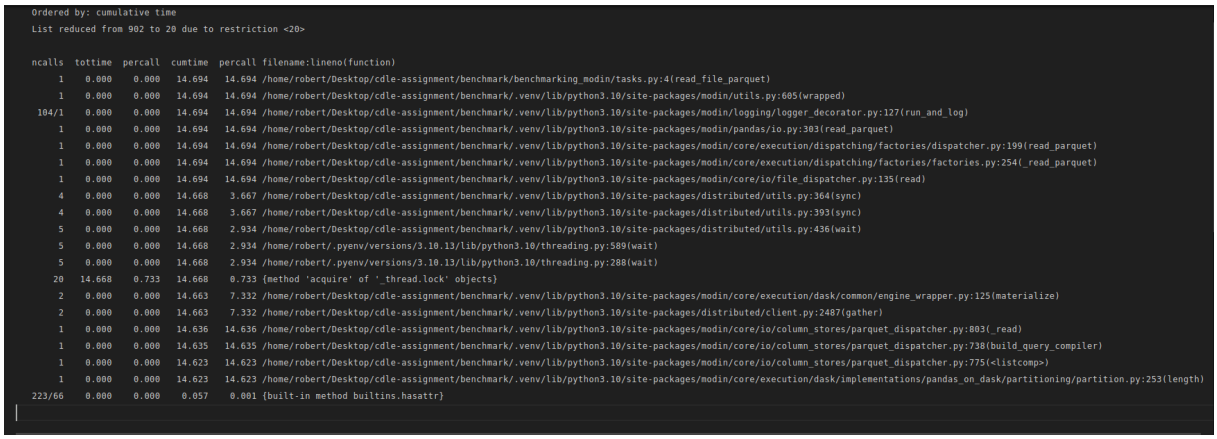
  Podemos ver que o Modin leva 14.694 segundos para ler o parquet, sendo que 14.623 segundos são usados pela: `partitioning/partition.py:253(length)`.



```
Ordered by: cumulative time
List reduced from 1125 to 20 due to restriction <20>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   13.315   13.315 /home/robert/Desktop/cdle-assignment/benchmark/benchmarking_modin/tasks.py:35(mean_of_complicated_arithmetic_operation)
1416/31    0.015    0.000   13.313    0.429 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/logging/logger_decorator.py:127(run_and_log)
146/109    0.001    0.000   12.641    0.116 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/storage_formats/pandas/query_compiler_caster.py:137(cast_args)
     7    0.001    0.000   12.212    1.745 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/pandas/base.py:4327(__array_ufunc__)
    37    0.001    0.000   12.202    0.330 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/utils.py:364(sync)
    37    0.001    0.000   12.200    0.330 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/utils.py:393(sync)
    37    0.000    0.000   12.195    0.330 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/utils.py:436(wait)
    37    0.000    0.000   12.195    0.330 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/threading.py:589(wait)
    37    0.000    0.000   12.195    0.330 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/threading.py:288(wait)
   148   12.194    0.082   12.194    0.082 {method 'acquire' of '_thread.lock' objects}
    39    0.001    0.000   11.788    0.302 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/dataframe/pandas/dataframe/utils.py:712(run_f_on_minimally_updated_metadata)
     9    0.000    0.000   10.871    1.208 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/storage_formats/pandas/query_compiler.py:351(to_pandas)
     9    0.000    0.000   10.871    1.208 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/dataframe/pandas/dataframe/dataframe.py:4664(to_pandas)
     9    0.056    0.006   10.867    1.207 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/dataframe/pandas/partitioning/partition_manager.py:984(to_pandas)
     8    0.000    0.000   10.781    1.348 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/pandas/series.py:2338(_to_pandas)
     9    0.000    0.000   10.625    1.181 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/dataframe/pandas/partitioning/partition_manager.py:1167(get_objects_from_partitions)
     9    0.000    0.000   10.602    1.178 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/execution/dask/common/engine_wrapper.py:125(materialize)
     9    0.000    0.000   10.602    1.178 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/client.py:2487(gather)
 29/28    0.000    0.000    1.668    0.060 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/dataframe/pandas/partitioning/partition_manager.py:71(wait)
    28    0.000    0.000    1.602    0.057 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/execution/dask/common/engine_wrapper.py:143(put)
```

Figura 24: Modin complex arithmetic



```
Ordered by: cumulative time
List reduced from 902 to 20 due to restriction <20>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   14.694   14.694 /home/robert/Desktop/cdle-assignment/benchmark/benchmarking_modin/tasks.py:4(read_file_parquet)
     1    0.000    0.000   14.694   14.694 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/utils.py:605(wrapped)
 104/1    0.000    0.000   14.694   14.694 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/logging/logger_decorator.py:127(run_and_log)
     1    0.000    0.000   14.694   14.694 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/pandas/io.py:303(read_parquet)
     1    0.000    0.000   14.694   14.694 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/execution/dispatching/factories/dispatcher.py:199(read_parquet)
     1    0.000    0.000   14.694   14.694 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/execution/dispatching/factories/factories.py:254(_read_parquet)
     1    0.000    0.000   14.694   14.694 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/io/file_dispatcher.py:135(read)
     4    0.000    0.000   14.668    3.667 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/utils.py:364(sync)
     4    0.000    0.000   14.668    3.667 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/utils.py:393(sync)
     5    0.000    0.000   14.668    2.934 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/utils.py:436(wait)
     5    0.000    0.000   14.668    2.934 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/threading.py:589(wait)
     5    0.000    0.000   14.668    2.934 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/threading.py:288(wait)
    20   14.668    0.733   14.668    0.733 {method 'acquire' of '_thread.lock' objects}
     2    0.000    0.000   14.663    7.332 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/execution/dask/common/engine_wrapper.py:125(materialize)
     2    0.000    0.000   14.663    7.332 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/distributed/client.py:2487(gather)
     1    0.000    0.000   14.636   14.636 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/io/column_stores/parquet_dispatcher.py:803(_read)
     1    0.000    0.000   14.635   14.635 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/io/column_stores/parquet_dispatcher.py:738(build_query_compiler)
     1    0.000    0.000   14.623   14.623 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/io/column_stores/parquet_dispatcher.py:775(<listcomp>)
     1    0.000    0.000   14.623   14.623 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/modin/core/execution/dask/implementations/pandas_on_dask/partitioning/partition.py:253(length)
223/66    0.000    0.000    0.057    0.001 {built-in method builtins.hasattr}
```

Figura 25: Modin read parquet

Ao olhar para os metadados do parquet `yellow_tripdata_2010-01.parquet` temos:

```
created_by: parquet-cpp-arrow version 8.0.0
num_columns: 18
num_rows: 14863778
num_row_groups: 1
format_version: 1.0
serialized_size: 10228
```

Basicamente o parquet tem apenas 1 row_group, e é um grande e único chunk de dados, portanto modin + dask não consegue fazer uma partição eficiente dos dados. Dask puro não tem este problema, pois seu processamento é lazy e consegue pegar este valor dos metadados, comforme imagem abaixo usando as chamadas _dataset_info e checksum

```
dask/dataframe/dask_expr/io/parquet.py:792(checksum)
dask/dataframe/dask_expr/io/parquet.py:1312(_dataset_info)
dask/dataframe/io/parquet/arrow.py:903(_collect_dataset_info)
dask/dataframe/io/parquet/arrow.py:1116(_create_dd_meta)
```

Como modin se comporta como pandas, tem processamento eager e realiza a materialização do dataset, como visto na chamada abaixo:

`dask/common/engine_wrapper.py:125(materialize)`

Portanto, podemos presumir que modin tem limitações pelo comportamento eager e não performaria bem em grandes datasets, mas que pode ajudar aplicações que usam pandas a utilizar recursos do dask, ainda que com limitações. Além disso, modin suporta partições de linha, coluna e células, que pode ser útil para use cases que não precisamos usar todas as colunas (Dask é apenas partição por linhas). A maioria das outras task simples são rápidas o suficiente para não se tornarem um gargalo.

Uma opção de otimização seria configurar partições corretamente e ler apenas colunas que vão ser utilizadas.

- **Koalas:**

  Koalas embora opere por cima do Apache Spark, em nossos testes single node com 1 worker, se comportou bem e sem grandes overheads. Pudemos ver que grande parte do tempo utilizado foi realmente utilizado para processamento dos dados, sendo estes que foram surpreendentemente rápidos.

  Um exemplo foi a task de calculo da média de cálculos complexos:

  Vemos que a maior parte do tempo foi gaasto pelo spark para pegar o schema, computar os dados e dar collect dos dados. Logo após temos uma conversâo para um dataframe pandas (método toPandas()) e um reduce para pegar a média dos

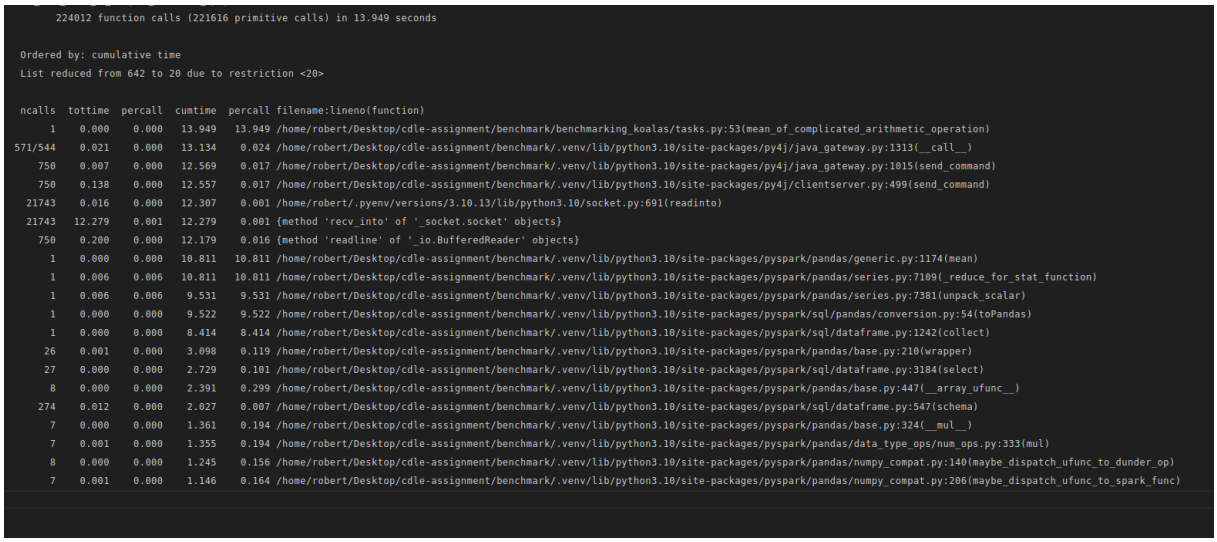


Figura 26: Koalas mean complex arithmetic

cálculos. Mas o processamento em spark tomou o maior tempo (8 segundos de 13). Temos algum overhead, como por exemplo as várias calls para o java gateway da JVM, provavelmente fruto da conversão do código python para uma código compilável pela JVM, mas este overhead nâo foi significativo.

Portanto, tendo em conta a comparação com outras ferramentas analisadas, o pyspark.pandas (koalas) é uma boa solução para nosso use case de datasets médios.

Percebemos que foram feitas 274 chamadas para validação de schemas do dataset. Utilizando cache ou especificando o schema como um metadado do parquet, esse processamento pode cair.

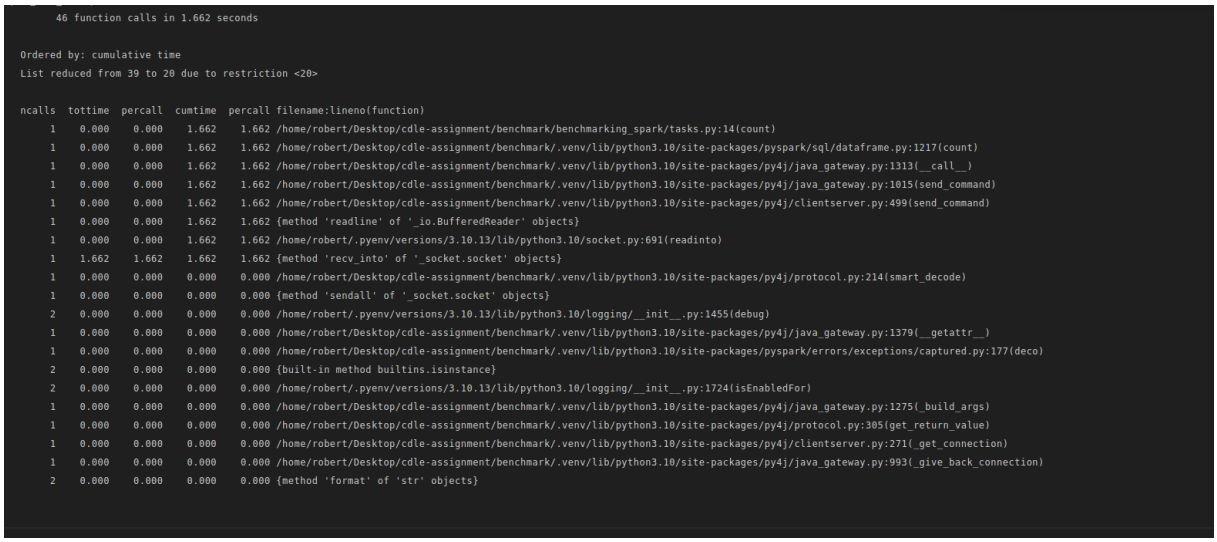
- **Spark:** Analisando os profilings, percebemos que o spark é bom em processamentos complexos como joins e cálculos aritméticos complexos, mas é lento para processamentos simples como simples contagem de linhas, adição de colunas e value_counts.

  Podemos ver na figura acima que grande parte do tempo para completude da task de contagem de linhas é destinada a chamada de recv_into(), uma chamada de network para JVM. O que me deixou na dúvida se o cProfile consegue calcular o tempo gasto de processamento na JVM e não apenas em ambientes python. O padrão segue para algumas outras task simples, mas que possuem overhead de I/O.

  Já para processamentos mais complexos, o spark lida bem devido a ter um bom query optimizer. Para vermos bem isso na prática, podemos ver as optimizações feitas pelo optimizer Catalyst na prática.

  Para a task 'mean_complex_arithmetic_operations' temos que:

  - O Spark estimou que a tabela original tem 491.5 MiB quando todas as 18 colunas são lidas
  - Após a otimização de projeção de colunas, apenas 87.8 MiB precisam ser lidos (só as 4 colunas de coordenadas) Isso representa uma redução de 82% na quantidade de dados que precisam ser lidos do disco



```
         46 function calls in 1.662 seconds

   Ordered by: cumulative time
   List reduced from 39 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.662    1.662 /home/robert/Desktop/cdle-assignment/benchmark/benchmarking_spark/tasks.py:14(count)
        1    0.000    0.000    1.662    1.662 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/pyspark/sql/dataframe.py:1217(count)
        1    0.000    0.000    1.662    1.662 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/py4j/java_gateway.py:1313(__call__)
        1    0.000    0.000    1.662    1.662 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/py4j/java_gateway.py:1015(send_command)
        1    0.000    0.000    1.662    1.662 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/py4j/clientserver.py:499(send_command)
        1    0.000    0.000    1.662    1.662 {method 'readline' of '_io.BufferedReader' objects}
        1    0.000    0.000    1.662    1.662 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/socket.py:691(readinto)
        1    1.662    1.662    1.662    1.662 {method 'recv_into' of '_socket.socket' objects}
        1    0.000    0.000    0.000    0.000 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/py4j/protocol.py:214(smart_decode)
        1    0.000    0.000    0.000    0.000 {method 'sendall' of '_socket.socket' objects}
        2    0.000    0.000    0.000    0.000 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/logging/__init__.py:1455(debug)
        1    0.000    0.000    0.000    0.000 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/py4j/java_gateway.py:1379(__getattr__)
        1    0.000    0.000    0.000    0.000 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/pyspark/errors/exceptions/captured.py:177(deco)
        2    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        2    0.000    0.000    0.000    0.000 /home/robert/.pyenv/versions/3.10.13/lib/python3.10/logging/__init__.py:1724(isEnabledFor)
        1    0.000    0.000    0.000    0.000 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/py4j/java_gateway.py:1275(_build_args)
        1    0.000    0.000    0.000    0.000 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/py4j/protocol.py:305(get_return_value)
        1    0.000    0.000    0.000    0.000 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/py4j/clientserver.py:271(_get_connection)
        1    0.000    0.000    0.000    0.000 /home/robert/Desktop/cdle-assignment/benchmark/.venv/lib/python3.10/site-packages/py4j/java_gateway.py:993(_give_back_connection)
        2    0.000    0.000    0.000    0.000 {method 'format' of 'str' objects}
```

Figura 27: Spark count of lines

```
=== COST-BASED OPTIMIZATION ===

+- Project [pickup_longitude#5, pickup_latitude#6, dropoff_longitude#9, dropoff_latitude#10], Statistics(sizeInBytes=87.8 MiB)
   +- Relation [vendor_id#0,pickup_datetime#1,dropoff_datetime#2,passenger_count#3L,trip_distance#4,pickup_longitude#5,
   pickup_latitude#6,rate_code#7,store_and_fwd_flag#8,dropoff_longitude#9,dropoff_latitude#10,payment_type#11,fare_amount#12,
   surcharge#13,mta_tax#14,tip_amount#15,tolls_amount#16,total_amount#17] parquet, Statistics(sizeInBytes=491.5 MiB)
```

Figura 28: Catalyst cost optimmization

O Catalyst também consegue otimizar algorítmos. Para este mesmo exemplo, diversas otimização foram feitas, em especial, o processamento incremental da tabela enquanto se atualizava o valor numa hashtable (feito pela chamada HashAggregate 2-phase optimizaation).

13 Pipeline de Machine Learning

Adicionalmente ao Benchmarking, tivemos a tarefa de construir uma pipeline de machine learning que fosse escalável e eficiente com datasets pesados, como é o caso dos datasets do NYC Taxi Driver.

O objetivo foi prever o valor das tarifas das corridas de taxi (fare_amount), utilizando técnicas avançadas de Big data e Machine Learning.

13.1 Abordagem Inicial

Inicialmente, nossa abordagem envolvia o uso da bilbioteca Modin, com o backend Dask, para acelerar as operações em dataframes. Escolhemos o Modin por apresentar uma interface muito semelhante ao Pandas, combinada com engines distribuídas como o Dask, o que facilitaria o desenvolvimento inicial. Entretanto, não foi possível seguir com essa abordagem, devido ao aumento significativo no volume de dados. Em um primeiro momento, conseguimos rodar localmente com um dataset utilizando o Modin, porém ao tentar escalar para mais datasets na VM encontramos muitos obstáculos. Apesar de termos testado a

execução em uma VM com 52Gb de memória RAM no GCP, rapidamente essa abordagem se mostrou inviável por suas falhas frequentes, mesmo ao tentar processar poucos datasets. Isso se deu pelo fato do Modin, por padrão, manter todos os dados inteiramente em memória (eager), o que limita a sua escalabilidade em workloads mais pesados. Durante as etapas de carregamento, especialmente ao tentar concatenar dataframes, o consumo de RAM crescia rapidamente, fazendo a VM *crashar* por esgotamento de recursos.

Por este motivo, decidimos por adotar uma outra abordagem, que permitisse o processamento escalável de grandes volumes de dados sem depender exclusivamente da memória RAM. Para isso, utilizamos a biblioteca Dask, que oferece suporte nativo a operações distribuídas e paralelismo em disco. Assim, foi possível rodar a pipeline em datasets de maior volume e em tempos razoáveis.

O primeiro passo no desenvolvimento foi a configuração do Client do Dask. Durante a fase de testes, experimentamos diferentes configurações de paralelismo:

- 1 worker com 1 thread,
- 1 worker com 4 threads,
- 4 workers com 4 threads cada.

Todas essas configurações funcionaram corretamente, porém o melhor desempenho foi alcançado com 4 workers e 4 threads por worker, evidenciando o ganho obtido ao explorar paralelismo tanto entre workers quanto entre threads. Essa configuração aproveitou melhor os recursos da máquina, reduzindo o tempo total de execução da pipeline.

Assim, o Client foi configurado com 4 workers, 4 threads por worker e um limite de memória ajustado para 40 GB, mesmo em uma máquina virtual com 52 GB disponíveis. Essa configuração garantiu um bom equilíbrio entre desempenho, utilização de recursos e estabilidade durante o processamento e treinamento do modelo.

13.2 Leitura e Pré-processamento dos Dados

Na fase de leitura dos datasets, utilizamos a biblioteca dask.dataframe, que replica a API do Pandas, mas com suporte para escalabilidade e paralelismo, uma vez que os dados são divididos em *chunks* e processados em paralelo.

Os datasets foram armazenados em um bucket no GCP e então, como sãao em formato Parquet, foram lidos com a função dask.dataframe.read_parquet. A leitura é feita de maneira *lazy*, ou seja, os dados não são carregados automaticamente na memória, evitando sobrecarga na inicialização.

Quanto ao pre-processamento dos dados, realizamos diversas operações a fim de obter o dataset mais útil possivel para o treinamento do modelo.

Começamos por remover colunas irrelevantes. Estas colunas foram consideradas irrelevantes por conter apenas valores nulos ou zeros, conter o mesmo valor em todas as linhas, ou conter valores nulos na maioria das suas linhas, o que exigiria definir valores para inputar nestas linhas e passa a não fazer sentido quando a maioria de suas linhas teria valores fictícios.

À seguir, filtramos valores inválidos a fim de reduzir o ruído no modelo. Consideramos que as colunas passenger_count e trip_distance não poderiam ter valores zero nelas, pois

não há sentido ter um registro de uma corrida sem passageiros ou sem distância, portanto estes valores tem de estar incorretos.

Por fim, realizamos feature engeneering nas colunas tpep_pickup_datetime e

tpep_dropoff_datetime, que apresentam um horário e data no formato datetime, que não seria útil para a predição do modelo, portanto transformamos estas colunas em novas colunas que representam a hora, dia da semana e mês em que as corridas foram iniciadas e finalizadas.

Após o pre-processamento de cada dataframe lido, este é concatenado em um dataframe final com a função dask.dataframe.concat e então apagado da memória.

Ao usar a computação *lazy* do Dask, nenhuma operação é realizada enquanto uma ação, como compute(), não for chamada.

13.3 Treinamento e Validação

Para o treinamento, escolhemos o algoritmo XGBoost para realizar regressão. Essa escolha se deu pelo fato do XGBoost permitir processamento distribuído e oferecer um ótimo desempenho em tarefas de predição com dados tabulares, além de ser bastante utilizado em casos como este.

Inicialmente, utilizamos a biblioteca padrão do XGBoost, mas após a migração para o Dask, passamos a usar o DaskXGBRegressor, uma versão da API do XGBoost adaptada para trabalhar diretamente com estruturas de dados distribuídas do Dask, o que possibilita o processamento em paralelo sobre grandes volumes de dados.

O pipeline de treinamento seguiu os seguintes passos:

- **Divisão dos dados:** Utilizamos a função dask_ml.model_selection.train_test_split para dividir o dataset em conjuntos de treino e teste, mantendo o particionamento distribuído e evitando o carregamento completo dos dados na memória.

- **Conversão para Dask Array:** Os dados foram convertidos para o formato Dask Array, que é o esperado pelo DaskXGBRegressor, garantindo compatibilidade e eficiência no treinamento.

- **Grid Search:** Em princípio, iríamos utilizar o Grid Search da biblioteca scikit-learn, porém depois da migração para o Dask, se evidenciou o fato do Dask não possuir suporte nativo para uma implementação completa e integrada de Grid Search. Portanto, tivemos de fazer esse passo de forma manual, testando iterativamente as combinações de parâmetros.

  Utilizamos a seguinte grade de parâmetros:

  - n_estimators: [50, 100, 200]
  - learning_rate: [0.01, 0.1, 0.2]
  - max_depth: [3, 5, 7]

- **Validação e avaiação** Após o treinamento, realizamos as predições utilizando o método compute() do Dask, que executa as operações pendentes e traz os resultados para a memória. Em seguida, avaliamos a performance do modelo com métricas clássicas de regressão, como RMSE, MAE e R^2.

13.4 Teste e Avaliação Final

Durante o próprio processo de Grid Search, já realizamos a avaliação do modelo com métricas clássicas de regressão (RMSE, MAE e R^2) em cada combinação testada. Assim, ao final do grid search, obtivemos:

- Métricas quantitativas de desempenho para cada configuração de parâmetros.
- Histogramas dos resíduos do melhor modelo, que ajudaram a visualizar os padrões de erro do modelo.
- Gráfico de importância das features do melhor modelo, útil para entender o peso relativo de cada variável na predição da tarifa.

Essas análises nos permitiram não só identificar a melhor configuração de hiperparâmetros, mas também interpretar o comportamento geral do modelo.

13.5 Resultados

O modelo final, treinado com os melhores parâmetros encontrados (`n_estimators=200`, `learning_rate=0.1`, `max_depth=7`), apresentou métricas com resultados bastante satisfatórios.

- **RMSE (Root Mean Squared Error)**: 0.88
- **MAE (Mean Absolute Error)**: 0.11
- **R^2 (Coeficiente de Determinação)**: 0.9873

Estes resultados podem ser interpretados como bons resultados pois a MAE de 0.11 indica que o erro absoluto médio é de apenas 11 centavos, ou seja, o modelo erra em média apenas 11 centavos por corrida. Considerando que os valores da coluna **fare_amount** variam tipicamente entre 5 e 50 dólares, este resultado é excelente. Além disso, a RMSE de 0.88 significa que, em média, o erro quadrático da previsão é inferior a 1 dólar, ou seja, erros maiores pesam mais na média. Por fim, o R^2 de 0.98 significa que 98% da variância do é explicada pelo modelo e, em tarefas de regressão, quanto maior o R^2, melhor a qualidade do modelo.

Ao avaliar as importancias das features, estes resultados se consolidam como boas previsões. Nas figuras 29 e 30, é possivel perceber que as features que mais influenciaram nos resultados do modelo são **total_amount, RatecodeID, trip_distance, tip_amount e extra**, respectivamente. Todas estas colunas, com exceção da **trip_distance**, representam valores monetários, que contribuem diretamente para a predição do valor de **fare_amount**. Além disso, a coluna **trip_distance** também tem contribuição direta para a determinação do valor de **fare_amount**, pois esta é calculada a partir da distancia da corrida e do tempo despendido.

feature_importance

Feature	Importance
total_amount	0.8777228
RatecodeID	0.04926356
trip_distance	0.035090398
tip_amount	0.020710742
extra	0.0055164094
DOLocationID	0.0020086465
PULocationID	0.0017574942
payment_type	0.0012589012
hour_dropoff	0.0011795135
hour_pickup	0.0011025325
day_of_the_week_pickup	0.000942888
VendorID	0.0009162327
day_of_the_week_dropoff	0.0007454265
passenger_count	0.00072225125
month_pickup	0.0005644993
month_dropoff	0.00049774046

Figura 29: Feature Importance

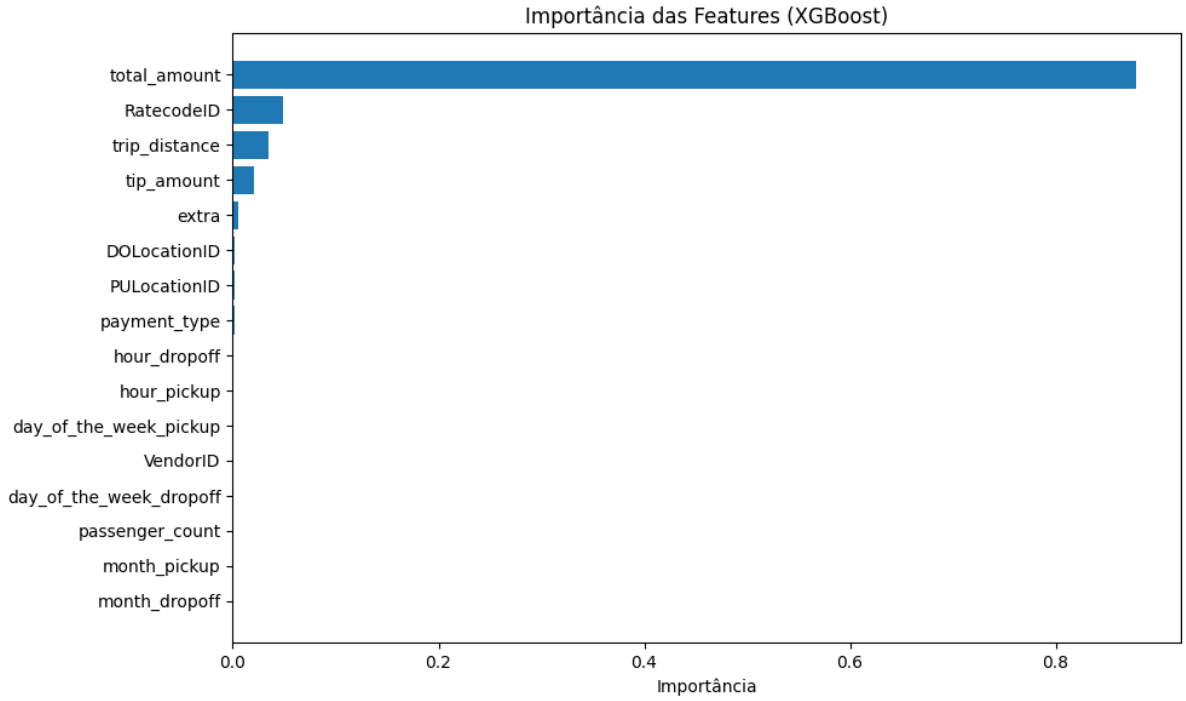


Figura 30: Feature Importance Plot

13.6 Testes e tempos de execução

Para avaliar a eficiência do pipeline e a influência da configuração do cluster Dask, realizamos testes com diferentes combinações de workers e threads, medindo o tempo total de execução para processar diferentes volumes de dados apenas com o melhor conjunto de parametros.

- 1 worker com 1 thread, processando 6 datasets
- 1 worker com 4 threads, processando 10 datasets
- 4 workers com 4 threads cada, processando 12 datasets

Os resultados foram os seguintes:

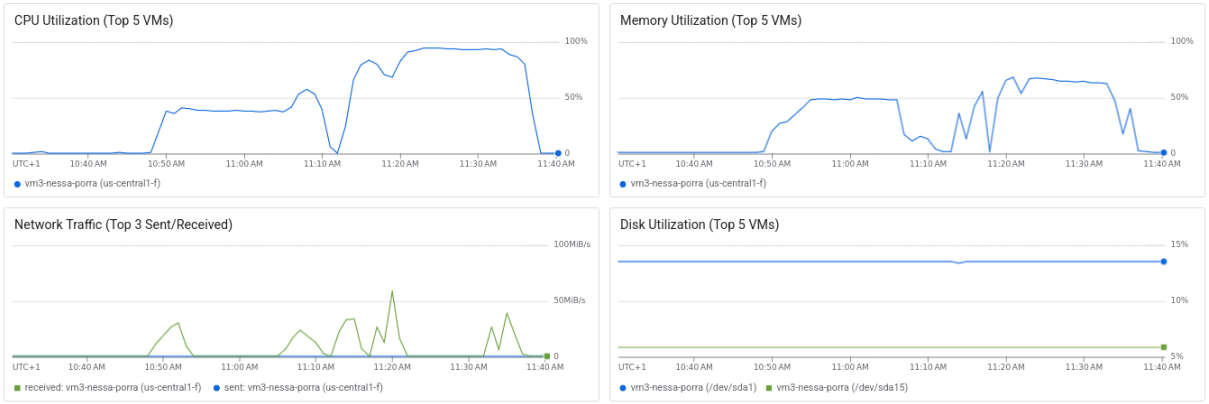
- 6 datasets, 1 worker e 1 thread: tempo total de execução de 2283,12 segundos - aproximadamente 38 minutos;
- 10 datasets, 1 worker e 4 threads: tempo total de execução de 1284,94 segundos - aproximadamente 22 minutos;
- 12 datasets, 4 workers e 4 threads: tempo total de execução de 1109,50 segundos - aproximadamente 19 minutos.

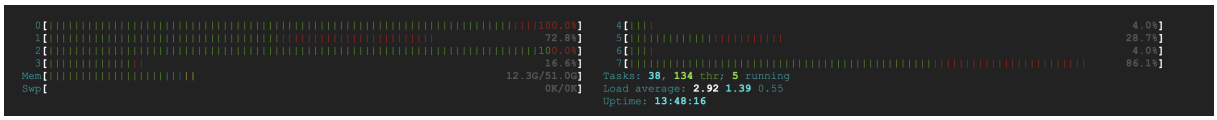
Apesar do aumento no volume de dados processados, a configuração com 4 workers e 4 threads apresentou o melhor desempenho, reduzindo o tempo total em relação às configurações com menos paralelismo. Já a configuração com 1 worker e 4 threads também trouxe ganhos expressivos, reduzindo o tempo pela metade em comparação com a execução totalmente serial (1 worker, 1 thread).

Esses resultados evidenciam que a utilização de múltiplos workers e threads permite explorar melhor os recursos do cluster Dask, acelerando tanto as operações de leitura e pré-processamento quanto o treinamento do modelo. Contudo, é importante balancear o paralelismo para evitar overheads e garantir a estabilidade do pipeline.

13.7 Consumo de Recursos

Podemos ver que, durante a execução com 1 worker e 4 threads, o consumo de vCPU's atingiu quase os 50%, mas utilizando 4 workers e 4 threads, o consumo de vCPU's atingiu quase 100%, portanto, extraíndo o máximo de eficiência possível de nossa máquina virtual, o que se refletiu nos baixos tempos de execução do treino do modelo.





14 Discussão e Conclusões

Este trabalho ajudou muito os integrantes do grupo a entenderem como funciona o processamento de dados, muito além do ferramental. Tivemos que explorar metadados de parquets, usar profilers para analisar eficiência de código, entender como funciona processamento distribuído e como fazê-lo em nuvem. Alem de aumentar nossos conhecimentos sobre deploy, workers, threads, processos e monitoramento de recursos em máquinas virtuais e clusters.

No desenvolvimento do pipeline de machine learning, fomos desafiados a aplicar esses conceitos para lidar com datasets pesados e complexos, escolhendo ferramentas e estratégias que equilibrassem escalabilidade, desempenho e estabilidade. Aprendemos a integrar o Dask para paralelismo e processamento distribuído com o XGBoost, a ajustar hiperparâmetros manualmente para otimizar resultados, e a interpretar métricas de performance em cenários reais. Além disso, entendemos na prática a importância da gestão cuidadosa de recursos, configurando workers e threads para evitar problemas como erros de sequência e inconsistências nos dados.

Foi um grande aprendizado, mesmo que com todas as dificuldades, e que particularmente estes que vos falam acharam muito divertido. Independentemente da nota do trabalho, os conhecimentos adquiridos serão utilizados durante a carreira.

14.1 Bibliografia

https://medium.com/%40liam.lim/nyc-yellow-taxi-trip-record-analysis-7eb389a0470c

https://joblib.readthedocs.io/en/stable/

https://koalas.readthedocs.io/en/latest/

https://docs.dask.org/en/stable/dataframe.html

https://spark.apache.org/docs/latest/

https://www.ibm.com/think/topics/xgboost

https://modin.readthedocs.io/en/stable/getting_started/why_modin/modin_vs_dask_vs_koalas.html

https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.SparkSession.html

https://www.turing.com/kb/python-code-with-cprofile

https://www.geeksforgeeks.org/difference-between-process-and-thread/