**Projeto: Mineração de Padrões em dados de supermercados**

| **Data** | **21 de Outubro de 2025** |
| --- | --- |
| Versão do Código | instacart\_son\_dask.py (v1.0) |
| Desenvolvedor Responsável | Ageu Felipe Nunes Moraes, André Cecato Justus, Ciro Guilherme Nass |
| Foco do Documento | Decisões Algorítmicas, Escalabilidade e Pipeline de Dados |

**1. Decisão Técnica Chave: Escolha do Algoritmo e Framework**

**1.1. Contexto do Problema**

O objetivo é realizar a Análise de Cesta de Mercado (Market Basket Analysis) para encontrar regras de associação (ex: pão $\to$ leite) em um volume massivo de dados transacionais de supermercado (similar ao dataset Instacart, com milhões de transações).

**1.2. Desafio Enfrentado: Memória e Tempo de Processamento**

O algoritmo clássico Apriori, executado em um único nó, exige que o *itemset* de candidatos seja mantido na memória principal. Com o número de itens de supermercado (cerca de 50.000 no Instacart) e um baixo *suporte* mínimo (ex: 0.1%), o número de *itemsets* candidatos explode, levando a:

1. **Out-of-Memory (OOM) Errors:** Incapacidade de armazenar todos os candidatos em uma única máquina.
2. **Tempo de Execução Inviável:** O Apriori tem um alto custo computacional, especialmente na fase de contagem.

**1.3. Solução Adotada: Algoritmo SON + Framework Dask**

| **Decisão** | **Justificativa Técnica** |
| --- | --- |
| **Algoritmo** | **SON (Sampling, Optimality, and Near-Optimality)** |
| **Framework** | **Dask (Dask.DataFrame / Dask.Bag)** |

**2. Detalhes da Implementação e Soluções (Dask Pipeline)**

| **Etapa/Decisão Técnica** | **Desafio/Problema Resolvido** | **Solução Adotada (Implementação no Código)** |
| --- | --- | --- |
| **1. Pipeline de Dados** | O order\_products\_\_prior.csv precisa ser agrupado por order\_id para formar listas de transações. | Uso de dask.dataframe.map\_partitions com agregação local de Pandas (.apply(list)) para criar o *bag* de transações (trans\_bag) de forma eficiente e distribuída. |
| **2. Suporte Local** | O suporte mínimo absoluto deve ser ajustado para cada partição, garantindo que nenhum *itemset* verdadeiramente frequente seja perdido (propriedade do SON). | O global\_support\_count é calculado com base no suporte relativo (min\_support\_rel \* total\_tx), e o suporte local é ajustado de forma proporcional ao tamanho da partição (max(1, int(math.ceil(global\_support\_count \* (part\_size / total\_tx))))). |
| **3. Paralelismo da Fase 1** | Como distribuir a execução do Apriori para cada pedaço de dado? | O método trans\_bag.to\_delayed() é usado para converter as partições do Dask Bag em tarefas Dask. A função apriori\_partition\_wrapper é então chamada via dask.delayed em cada tarefa, permitindo a execução paralela do Apriori local. |
| **4. Contagem Global (Fase 2)** | Contar a frequência dos candidatos únicos em todo o *dataset* de forma distribuída. | Após coletar os candidatos, a função map\_partition\_count é mapeada em todas as partições do trans\_bag (usando .map\_partitions). Isso garante que cada *worker* conte os candidatos em seu respectivo bloco de dados, e a contagem é agregada de forma eficiente pelo Dask. |
| **5. Saída Amigável** | Os resultados da mineração utilizam product\_id (inteiros), que são ininteligíveis para usuários de negócios. | Após gerar as regras, o products.csv é lido, e um mapeamento (id\_to\_name) é criado. As colunas antecedent\_names e consequent\_names são adicionadas ao DataFrame final com nomes legíveis antes de salvar. |

**3. Desafios Adicionais e Mitigações**

| **Desafio** | **Mitigação / Solução Adotada** |
| --- | --- |
| **Sobrecarga de Comunicação** | O SON tem uma sobrecarga na fase de agregação de candidatos. Para mitigar, os parâmetros do *cluster* Dask (--nworkers, --nthreads, --memory-limit) são expostos como argumentos. Isso permite que o usuário ajuste a configuração ideal de paralelismo versus *overhead* de comunicação para o hardware disponível. |
| **Serialização de Itemsets** | O Dask precisa serializar os frozenset (itemsets) entre os *workers*. No código, a serialização e desserialização são tratadas internamente pelas funções de dask.delayed e dask.compute. Na contagem global, a agregação final usa Counter para evitar problemas de colisão durante a soma. |
| **Validação do Apriori Local** | Para garantir que o Apriori local (apriori\_local) é correto, ele foi implementado como uma função Python pura (usando Counter e itertools.combinations), separada da lógica Dask. Isso facilita testes unitários e garante a fidelidade ao algoritmo original. |