0. Introdução

Este relatório de modelagem é referente ao Trabalho A2 da matéria de Computação Escalável, lecionada no 5º período do curso de Graduação em Ciência de Dados e Inteligência Artificial da FGV - EMAp. O relatório é constituído pela nossa modelagem do problema proposto, passando pelo projeto de exemplo e pela escolha de ferramentas e construção do sistema que atende às necessidades do projeto.

Integrantes do grupo

Anderson Gabriel Falcão dos Santos - andersonfalcaosantos@gmail.com
Guilherme Moreira Castilho - guilherme222castilho@gmail.com
Luan Rodrigues de Carvalho - luan.r.carvalho47@gmail.com
Pedro Santos Tokar - pedrotokar2004@gmail.com
Tomás Paiva de Lira - tomaspaivadelira1@gmail.com
Vitor Matheus do Nascimento Moreira - vitor.mnw@gmail.com

1. Projeto exemplo

1.1 Objetivos

O Banco **VT.PGA** recebe um grande fluxo de transações financeiras por segundo. O banco **VT.PGA** fornece uma proteção robusta contra fraudes, e garante que a aprovação de transações financeiras seja rápida e confiável.

A forma que uma transação é verificada envolve duas operações principais.

- **Cálculo determinístico** — Verifica se o saldo e o limite do cliente são compatíveis com o valor da transação.
- **Cálculo de risco** — Calcula valores de risco para características que podem indicar se uma transação é fraudulenta. Se uma certa função dos valores de risco for maior que um certo limiar, a transação não é liberada.

As análises que foram feitas por cima desse cenário no trabalho de A1 foram em sua maioria remoldadas e melhoradas para atender os requisitos da A2. Abaixo há um resumo de cada uma delas:

1.2 Análises

1 – Aprovações.
Análise feita sobre o objetivo final da pipeline: mostrar o número de transações aprovadas e rejeitadas. (Loader L2)

2 & 3 – Frequência de Transações por Valor & por Horário.
Análise de histogramas mostrando o número de transações são feitas em diferentes faixas de valor e nas faixas de horários do dia.

4 – Distribuição do Score de Risco de Valor, Valor da Transação & Status da Aprovação.
Mostra a relação entre valor e aprovação com os valores do respectivo score de risco.

5 – Relação Risco Horário × Aprovação.
Análise de verificação das relações entre o score de risco de horário e o status da aprovação.

6 – Região × Taxa de Aprovações.
Análise que consiste em verificar as taxas de aprovações em cada estado.

7 – Insuficiência de Saldo/Limite.
Análise que consiste em verificar quantas transações são rejeitadas por conta de saldo ou limite insuficiente.

8 - Tipo de Transferência com mais Transações Rejeitadas?
Análise que consiste em fazer comparações entre a frequência/quantidade de transações rejeitadas por compras debitando diretamente do saldo ou no crédito.
(Loader L7)

9 - Frequência de Transações por Horário.
Análise que consiste em reportar quantas transações ocorrem em cada horário, por exemplo com um gráfico de linha. A partir dessa análise dá para se inferir horários de pico e horários incomuns de transferências.
(Loader L8)

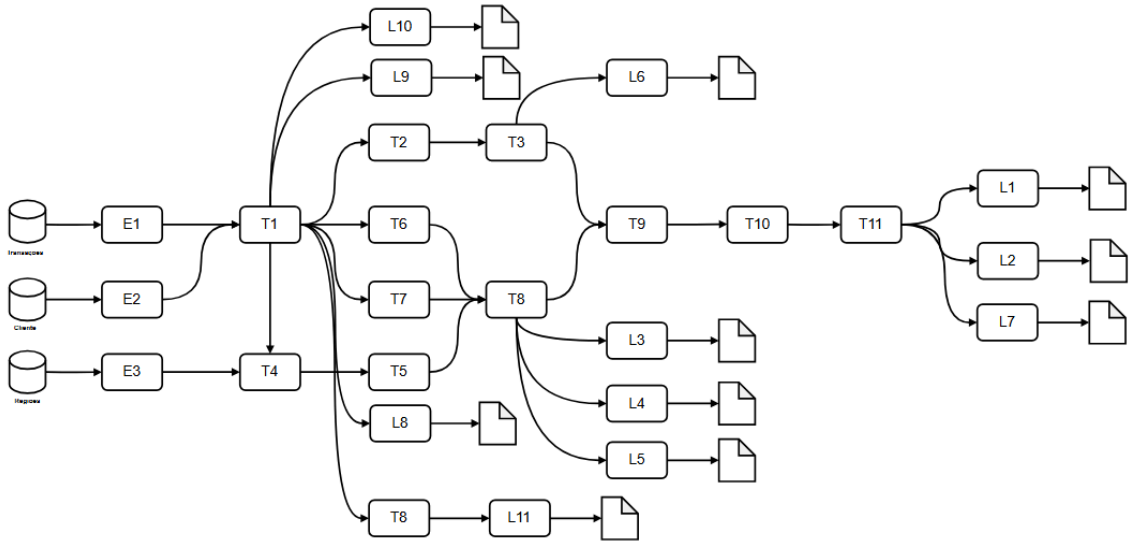
10 - Distância Geográfica Pagador-Recebedor X Aprovação.
Correlacionar a distância geográfica pagador-recebedor de um transação com a aprovação da mesma.
Agrupar transações em faixas de distância geográfica. Por exemplo: curta (<50km); média-curta (50-300km); média-longa (300-1000km); longa (>1000 km).
Para cada intervalo de tempo, calcular a proporção de transações de cada faixa de distância, aprovadas e rejeitadas. (Double stacked bar chart.)
(Loader L11)

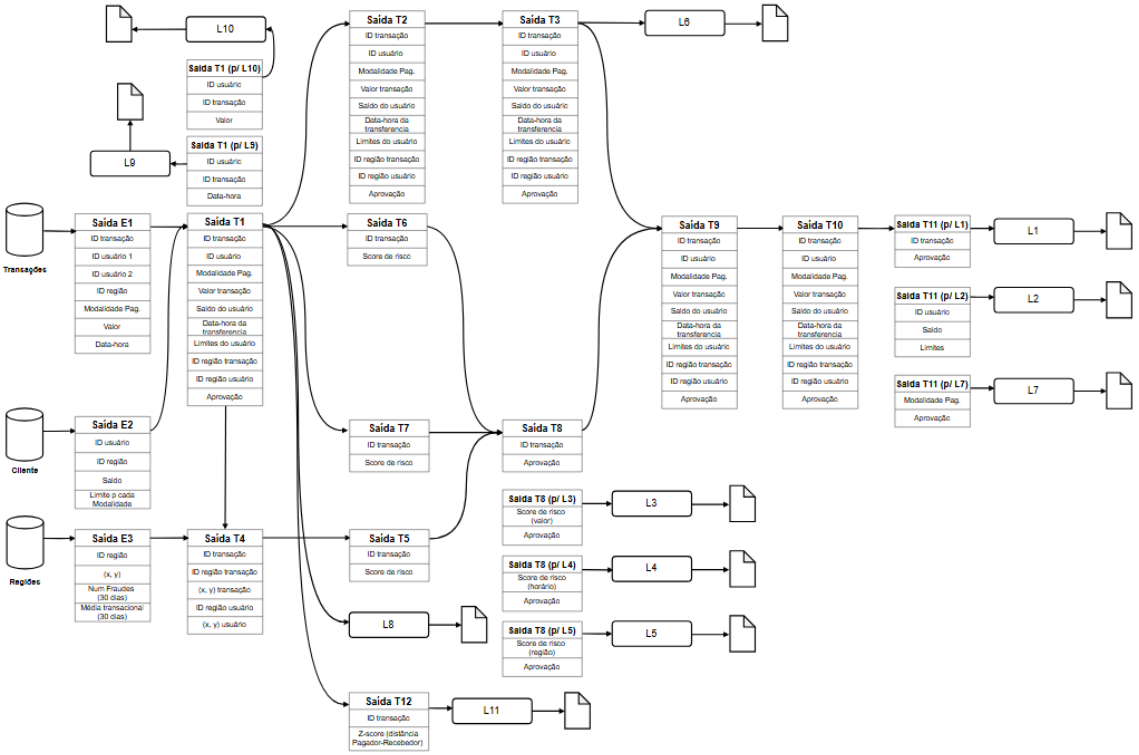
1.3 Pipeline

As operações realizadas no ETL seguiram o mesmo molde da pipeline desenvolvida na A1. As imagens ilustram como a pipeline se organizava, mas, como melhor explicado na sessão do Spark, o conceito do DAG das operações foi abstraído do desenvolvimento, e as operações foram declaradas sequencialmente.

As principais diferenças da pipeline para o segundo trabalho se resumem ao salvamento de dados que antes eram usados apenas em etapas intermediárias do cálculo, como a região do cadastro dos usuários. Esses dados agora são salvos para permitir a realização de análises mais completas.

Uma outra mudança mais expressiva foi na operação de cálculo de um dos scores de risco. O score que era calculado com a média das últimas transações de um usuário foi trocado por um score calculado comparando o valor da transação com um valor fixo. Esse novo score, quando tem valor 0, evita que a transação seja negada por outros scores, mas ainda permite a negação por limite ou saldo.





O formato que as transações seguem é deixado aqui para referência em outras sessões:

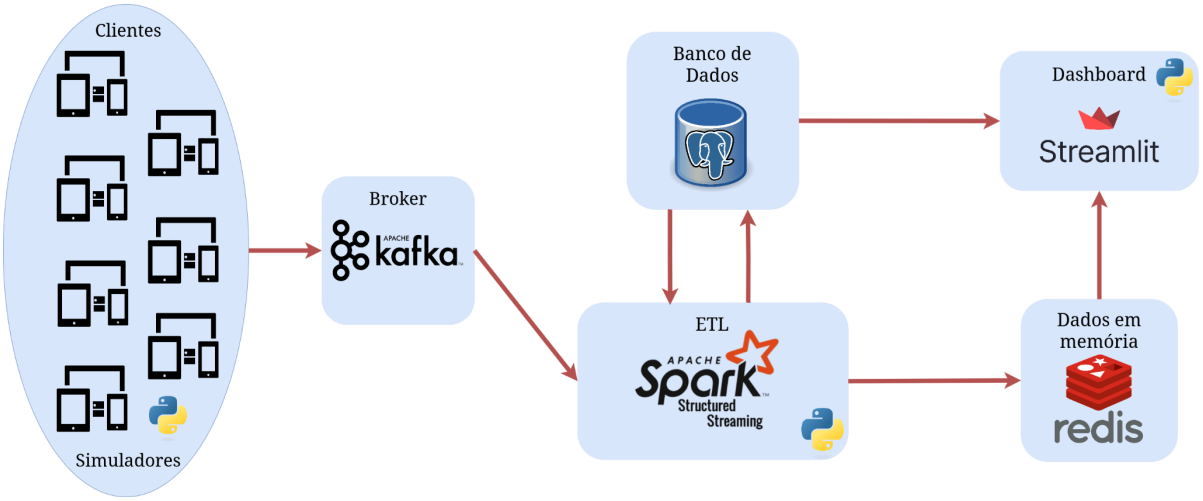
Transações	
Descrição coluna	**Tipo**
ID da transação	string
ID do usuário pagador	string
ID do usuário recebedor	string
ID região	string
Modalidade do pagamento	string
Data e Horário	datetime
Valor da Transação	float

2. Modelagem do Sistema

2.1. Sobre as decisões arquiteturais

Todas as ferramentas e paradigmas de comunicação usados na implementação do sistema do projeto foram pensados de forma que ele apresentasse as características que se espera de um sistema bancário de verificação de fraudes em transações. Uma destas características, que se destaca das demais, é o **baixo tempo de resposta.** Um sistema bancário precisa confirmar rapidamente para um usuário o status (aprovado ou rejeitado) de sua transação, e por isso escolhemos seguir o **requisito alternativo: "Processamento em tempo real".**

Montamos, então, a seguinte arquitetura para o projeto:



A alimentação do sistema com dados de transações bancárias é feita por meio do padrão de Publish-Subscribe. Este padrão foi escolhido por se adequar ao formato em que muitos usuários/clientes fazem requisições ao sistema, e por tornar possível o enfileiramento e a persistência temporária dos pedidos. Assim, o sistema garante que as requisições são recebidas e armazenadas por tempo suficiente até serem atendidas, garantias que outros padrões, como o RPC, não forneceriam por padrão. Para a implementação, a ferramenta utilizada foi o **Apache Kafka**.

Para o processamento de dados com alta paralelização e escalabilidade, a ferramenta escolhida foi o **Apache Spark** em modo de **Structured Streaming**. Esse software foi escolhido por ser uma opção sólida, robusta e que tira do desenvolvimento da pipeline diversas preocupações relacionadas ao desempenho, como por exemplo a distribuição da carga por clusters e o gerenciamento de threads ou comunicação inter-processo. Além disso, ele já se integra com o **Apache Kafka**, o que proporciona baixa latência e agilidade no desenvolvimento.

No que se refere ao armazenamento dos dados processados, o **PostgreSQL** foi a ferramenta usada como banco de dados operacional do sistema, por ser uma opção estável e segura para armazenar dados, principalmente de usuários e históricos. O **Redis** foi escolhido para certas informações que precisam ser exibidas no dashboard com baixa latência: por armazenar os dados em memória, ele fornece acesso rápido, servindo como um cache para que os resultados de tempo real que saem da pipeline possam ser acessados rapidamente pelo dashboard.

Já o dashboard com os dados processados e as análises foi feito utilizando **Streamlit.** Essa biblioteca de **Python** foi escolhida por oferecer uma API de desenvolvimento de dashboards bem fácil de usar e ao mesmo tempo robusta, com diversas opções de componentes e integração com bibliotecas de visualização familiares aos membros do grupo.

Para tornar possível o uso de todas as tecnologias durante o desenvolvimento local, foi utilizada a ferramenta **Docker Compose**, que permite executar cada ferramenta/serviço localmente em contêineres separados, mas com comunicação entre eles facilitada. Em conjunto, foi utilizado o **Apache Zookeeper** para coordenar a comunicação entre broker Kafka e Spark.

Em um ambiente de produção, as transações seriam enviadas por usuários do banco que estão fazendo uso do aplicativo do banco. Para finalidade de testar a pipeline e sua tolerância a cargas, implementamos geradores de transações que leem o banco de dados de usuários, geram transações com dados válidos e enviam ao broker.

2.2. Pub-sub: Kafka

O **Apache Kafka** foi o software utilizado como broker para a implementação do padrão Publish-Subscribe. Ele atende aos requisitos: **escalabilidade, processamento em tempo real e tolerância a falhas**. Nesse contexto, os simuladores produzem **eventos** (transações que precisam ser aprovadas ou rejeitadas) que são publicados em um **tópico.**

A eficiência do Kafka está em sua arquitetura:

1. **Paralelismo:** Cada tópico é dividido em partições, permitindo que múltiplos consumidores processem o mesmo fluxo de dados simultaneamente.
2. **Tolerância a Falhas:** As partições são replicadas através de um cluster de brokers (servidores), garantindo a durabilidade dos dados e a continuidade do serviço mesmo que um servidor falhe. Isso é essencial para garantir que as transações não processadas não se percam caso algum dos componentes do sistema falhe.
3. **Desacoplamento:** Produtores (simuladores) e consumidores (no nosso caso, a pipeline Spark) são independentes. Um pode escalar ou falhar sem impactar o outro.

A comunicação dos clientes que publicam transações (**eventos**) no broker Kafka é feita por meio de um tópico `transacoes`. Esse tópico recebe em formato JSON as informações de uma transação: cada mensagem é um JSON com as chaves correspondendo aos campos presentes na tabela original de transações (presente na seção 1) e os valores correspondendo às informações da transação. Nos simuladores utilizados, a geração das strings JSON é feita em Python e o envio é feito utilizando a biblioteca `confluent-kafka-python`. As transações enviadas são validadas e feitas com os usuários presentes na base de dados Postgres, evitando assim inconsistências na simulação.

Já a pipeline Spark atua como inscrita no tópico, consumindo as transações assim que elas chegam no broker, como será melhor detalhado na próxima seção. O paralelismo do Spark para a operação de leitura do tópico é diretamente atrelado ao número de partições do Kafka, ou seja, o número de consumidores lendo deve ser menor ou igual ao número de partições do tópico. Ainda assim, o spark pode remanjar o número de workers para processar os dados de forma independente do Kafka.

Ademais, o Kafka garante que, dentro de uma mesma partição, as mensagens são sempre lidas na ordem em que foram escritas.

2.2.1 Motivações por trás da escolha

O Kafka fornece uma solução **durável** (as mensagens persistem) e tolerante a falhas, sendo uma solução altamente **escalável** para a transmissão de mensagens, o contexto do nosso ETL necessita destas características.

Ele também é muitas vezes usado para aplicações consideradas de tempo real, embora não tenha uma latência tão baixa quanto soluções síncronas como RPC, por exemplo. Porém, usando ele podemos liderar com picos de mensagens de forma mais confiável. Além disso, o Spark Structured Streaming, que é detalhado abaixo, tem suporte nativo ao Kafka.

2.3. Processamento de dados: Spark

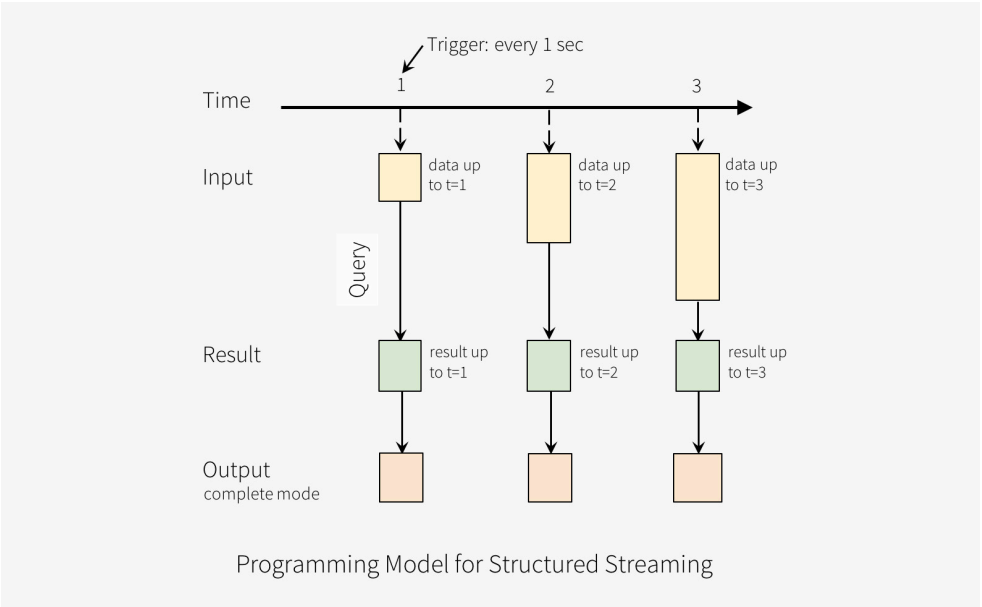
Para a implementação das operações de transformação do ETL, utilizamos o software Spark. Essa decisão aconteceu principalmente pelo **nível de abstração que o framework oferece**, gerenciando automaticamente e de forma eficiente os recursos disponíveis tanto em ambientes locais quanto em ambientes de nuvem. Ao utilizar a biblioteca, os esforços do desenvolvimento podem se voltar para outras questões importantes.

2.3.1. Spark Structured Streaming e características

Como o projeto está relacionado a um pipeline bancário de aprovação de transações, que depende de confiabilidade e de disponibilidade, e como optamos por fazer um sistema que possa processar as requisições (transações) no menor tempo possível (**tempo real**), utilizamos o [Spark Structured Streaming]. Essa é uma forma de usar o Spark pensada para o paradigma de **processar dados chegando em stream**, no lugar do modelo padrão de processar batches de dados após intervalos grandes de tempo.

O funcionamento do Structured Streaming se dá por meio da separação dos dados recebidos em micro-batches, que são processados em intervalos específicos de dados, que podem ser configurados pelo usuário.

Para garantir a baixa latência de resposta da ETL, esses **intervalos deveriam ser calibrados para serem os menores possíveis** sem prejudicar o processamento do ETL (na casa dos **milissegundos**). Essa calibração seria feita por meio de testes de carga, para entender como o comportamento do Spark muda para diferentes intervalos. Esse teste, porém, não pôde ser executado na implementação desse trabalho, e a pipeline está com um trigger padrão de 500 milissegundos entre as execuções.



Programming Model for Structured Streaming

Para que o sistema funcione corretamente com o Spark Structured Streaming, **algumas regras de negócio deveriam ser definidas**:

- O cadastro dos usuários deveria conter a timestamp de sua última transação aprovada. Se uma outra transação gerada antes chegasse depois que essa última foi aprovada, ela não seria processada.
- Se a diferença entre a geração e o recebimento de uma transação for alta (por exemplo, 1s), ela não deveria ser processada.

Essas regras são importantes para garantir a idempotência da pipeline, já que o Spark Structured Streaming pode requisitar o processamento de uma mesma linha mais de uma vez, como mecanismo que visa não perder o processamento de nenhuma linha de dado em caso de falhas. Nesse caso não é desejado que a pipeline processe novamente uma transação e possivelmente faça novas alterações no saldo do usuário.

Na implementação final, porém, essas regras não foram asseguradas. Durante os testes realizados no desenvolvimento, tentamos assegurar a primeira regra por meio de uma etapa de filtragem, usando o método `applyInPandasWithState`, que permite que operações com estado sejam realizadas no dataframe streaming. Esse método permitia com que a informação da última transação do usuário fosse salva e usada entre batches.

O que observamos após a implementação foi que adicionar a etapa de filtragem pelo horário da última transação levava a execução dos microbatches do streaming com um intervalo superior a um minuto. Após algumas pesquisas, a conclusão a que chegamos foi que o uso da API de operações com estado introduziu um overhead muito grande para o Spark, que não foi mais capaz de executar todas as operações em Java. Como esse intervalo de tempo compromete a velocidade de processamento almejada na pipeline, ele não foi colocado na versão final (sua implementação se encontra na branch `feat/etl-improvements`).

Como os dados são gerados por simulador, a ausência das regras não é prejudicial para a implementação atual. Para um ambiente de produção, porém, seria necessário estudar outras opções e formas de assegurá-las que não comprometesse o desempenho da pipeline.

2.3.2. Entradas e saídas

Como já mencionado na visão geral das ferramentas, a forma com que a comunicação com clientes é feita usa o padrão de comunicação pub-sub com o serviço **Kafka**. O Structured Streaming oferece, por padrão, integração com este serviço por meio de métodos nativos, tornando o sistema mais robusto e melhor integrado.

A principal saída da pipeline é uma tabela com as transações processadas, com uma coluna indicando se foi aprovada ou recusada. Existe uma diferenciação entre transações que explicitamente são recusadas, seja por score de risco ou saldo insuficiente, e transações que não são processadas, como transações que chegam atrasadas. Se as regras descritas acima fossem asseguradas, o segundo tipo de transação não entraria na tabela de resultados. Além de serem escritas na base Postgres, as transações também são armazenadas temporariamente na base Redis, para serem acessadas com velocidade pelo dashboard.

Além disso, a pipeline idealmente também deveria atualizar as informações de cadastro de usuários com o saldo novo e a timestamp da última transação processada. Devido aos resultados com o teste de contabilizar o momento da última transação, essa ferramenta não foi implementada, já que também dependeria de uma lógica stateful e isso tornaria a pipeline muito mais lenta. Uma direção futura pode ser pensar em maneiras de executar essas operações sem depender do operador stateful arbitrário `applyInPandasWithState`.

2.3.3. Implementação

O Spark oferece formas de utilizar o framework em diversas linguagens de programação. Devido à familiaridade dos integrantes do grupo com a linguagem **Python** e devido ao seu uso em outros componentes do sistema, a forma de uso do Spark escolhida foi por meio da biblioteca `pyspark`.

Durante o desenvolvimento do primeiro trabalho, uma grande preocupação foi com a estrutura das operações da pipeline. Havíamos desenhado um DAG manualmente, selecionando quais tarefas podem ser feitas em paralelo, quais não podiam e quais colunas eram usadas em qual. Com o uso do Spark, não foi mais necessário definir explicitamente o DAG.

No lugar de defini-lo, a pipeline é feito com as operações oferecidas pela API do Spark são usadas, e o Spark internamente constrói um DAG e otimiza as operações e divisões entre as unidades de processamento e máquinas disponíveis. Essa abordagem é mais robusta, já que a biblioteca tem maneiras maduras e consolidadas de dividir o processamento e também leva em conta a natureza de streaming para fazer decisões coerentes.

Em detalhes, o que é feito na pipeline implementado.

- Definimos os esquemas das transações, seguindo o JSON do tópico Kafka (usando os objetos `StructType` e `StructField`).
- Informamos ao Spark como ler as transações da stream Kafka, fornecendo informações do tópico e configurações (usando o método `readStream.load()` do objeto `SparkSession` que representa a sessão do Spark).
- Instruímos o Spark para ler os dados de usuários e regiões do banco de dados Postgres, também fornecendo informações de conexão e tabelas a serem lidas (usando o método `read.jdbc()` do objeto da sessão spark).
- Com os dados carregados, realizamos um LEFT JOIN das três tabelas utilizadas, adicionando para cada transação colunas com informações do usuário pagador, da região do usuário e da região da transação (usando o método `join()` do objeto `DataFrame` do Spark). Apesar do script declarar essas operações antes das transformações e cálculos, o Spark otimiza internamente qual será o momento em que o JOIN realmente será feito.
- Utilizamos as operações de dataframes do spark (como `withColumn()`, `F.when()` e `select()`) para efetivamente calcular os índices e fazer as verificações da pipeline. Essas declarações são feitas de forma sequencial.

- Com o dataframe das transações feito (com colunas de aprovação e saldo nos scores), a escrita dos resultados no redis e no Postgres é feita usando a sink `foreachBatch()`, já que o Spark não tem sinks de Postgres e Redis nativas no Structured Streaming. A função que é associada ao `foreachBatch()` faz seleções das colunas e usa o formato `jdbc` para escrever no Postgres e o formato `org.apache.spark.sql.redis` para escrever no Redis.

2.4. Banco Relacional: PostgreSQL

O **PostgreSQL** foi adotado como banco de dados relacional principal do sistema, atendendo aos requisitos de consistência, integridade transacional e suporte a consultas complexas - características essenciais para aplicações financeiras. Sua robustez garante que cada transação bancária registrada ocorra de forma segura e confiável, mesmo em cenários de concorrência intensa ou falhas parciais.

O modelo relacional permite uma modelagem estruturada das entidades principais do sistema: usuários, contas, transações, limites e auditoria. Com o uso de chaves primárias, estrangeiras e constraints, conseguimos garantir que os dados armazenados estejam sempre em conformidade com as regras de negócio, o que é crucial em sistemas financeiros.

Além disso, o **PostgreSQL** oferece recursos como:

- **Índices eficientes** (incluindo índices compostos e parciais), que aceleram consultas frequentes por id de usuário, data da transação, entre outros campos estratégicos.
- **Views materializadas e triggers**, que podem ser usadas para manter pré-processamentos ou lógicas automatizadas, sem prejudicar a performance da ingestão de dados.

O esquema final conta com as tabelas de **usuários** e de **regiões**, para armazenar informações como localização das regiões e saldo dos usuários. Essas tabelas são consultadas pelo sistema no início da sua execução. Também há uma tabela de **transações,** que contém as colunas apresentadas inicialmente, uma coluna booleana indicando a aprovação ou não da transação e algumas colunas para análise de métricas relacionadas à latência do sistema. Uma última tabela, de **scores de transações**, contém apenas os scores de risco de cada transação, para facilitar a geração de análises.

Em adição ao servidor Postgres, que é iniciado pelo Docker e tem uma imagem própria, também há um container `db-seed`, cuja finalidade é gerar dados históricos falsos para o banco. Esse container executa um script em Python que gera o esquema das tabelas na base e popula elas com dados sintéticos de usuários e regiões, caso não existam. Isso garante que qualquer execução local do sistema tenha essas tabelas disponíveis.

Já as tabelas relacionadas a transações são alimentadas diretamente com os dados já processados pelo Spark. Após o ETL, os dados são validados, agregados (quando necessário), e persistidos no **PostgreSQL** para garantir durabilidade, integridade referencial e suporte a análises futuras.

2.4.1 Motivações por trás da escolha

A escolha pelo **PostgreSQL** se justifica pela maturidade da tecnologia, sua forte aderência ao modelo relacional e a capacidade de lidar com cargas médias e pesadas sem comprometer a consistência dos dados.

Além disso, o **PostgreSQL** é software livre, bem documentado e amplamente utilizado em produção, oferecendo um equilíbrio entre desempenho, confiabilidade e custo. Sua integração com outras ferramentas do ecossistema (como **Redis** e **Streamlit**) é direta, o que favorece a arquitetura modular do sistema.

2.5. Armazenamento de Baixa Latência: Redis

O Redis foi utilizado como armazenamento auxiliar para os dados que compõem o dashboard em tempo real. Por se tratar de um banco de dados chave-valor *in-memory*, o **Redis** oferece uma latência extremamente baixa, ideal para casos em que a atualização e a leitura de dados precisam ser quase instantâneas — como no monitoramento de transações bancárias em tempo real.

Diferente do **PostgreSQL**, que prioriza integridade e persistência, o Redis é utilizado aqui como um cache inteligente para informações resumidas ou temporárias, tais como:

- Número de transações por minuto
- Saldo atualizado por usuário
- Alertas de atividade suspeita
- Agregações por região, agência ou faixa horária

Esses dados são atualizados continuamente pelo **Spark** após o processamento dos eventos **Kafka** e ficam disponíveis para leitura direta pelo **Streamlit**, sem necessidade de consultas custosas ao banco relacional.

Para o nosso projeto, utilizamos um folder chamado *transacoes,* que armazena hashes com os dados da saída do nossa pipeline. No entanto, a princípio, existe um problema: não conhecemos diretamente a ordenação das chaves destes hashes (uuids das transações), e nosso interesse é ter informações das transações mais recentes. Para resolver isso, estamos utilizando um **Sorted Set** do redis, que nos dá uma forma eficiente de obter os hashes de forma ordenada (aqui estamos usando a timestamp para a ordenação).

2.5.1 Motivações por trás da escolha

A escolha pelo Redis foi motivada pela necessidade de alta performance na leitura e escrita de dados que mudam com frequência e precisam ser acessados em tempo real.

Sua arquitetura em memória o torna uma solução ideal para dashboards reativos e responsivos, além de complementar perfeitamente o **PostgreSQL**: enquanto este é voltado à durabilidade e consistência, o **Redis** atua como camada de acesso rápido para dados voláteis ou de alta frequência de consulta.

Por fim, o **Redis** é leve, fácil de integrar com **Python** (via `redis-py`) e pode ser escalado horizontalmente com mecanismos como Redis Cluster, caso necessário.

2.6. Apresentação das análises: Dashboard com Streamlit

Para a apresentação dos dados em tempo real, utilizamos o **Streamlit** como framework de visualização. A escolha se deu pela simplicidade de desenvolvimento, e integração com Python.

O dashboard exibe informações críticas do sistema, como volume de transações, resultados da aprovação, alertas e métricas agregadas. Estas informações são consultadas diretamente do **Redis** e do **Postgres**, e transferidas para dataframes de pandas, seguindo algumas estratégias para manter a solução eficiente.

- Em relação ao **Redis:**
    - Utilizamos um cache do streamlit para os dados recebidos do Redis que dura *x* segundos, após o tempo de vida desse cache os dados podem ser recarregados. Os dados são automaticamente recarregados após *y* segundos (escolhemos 2 para esse parâmetro), este carregamento é independente dos outros componentes, apenas essas parte do dashboard é recarregada.
    -
- Em relação ao **Postgres:**
    - Parte dos dados lidos do Postgres se tratam dos dados históricos (também lemos dados constantes como o da database de usuários e a de regiões).
    - Os dados são armazenados em cache do Streamlit (pickle) de duas formas:
        - Os dados constantes são armazenados em cache sem limite de duração. Por isso os dados só são obtidos uma vez na primeira run do dashboard. Isso aumenta consideravelmente a eficiência pois não precisamos buscar dados que não mudam toda vez que o dashboard é atualizado.
        - Os dados relacionados com a transação são armazenados em um cache do Streamlit com duração de *x* segundos. Essa escolha tem como base o princípio de que, pela natureza dos dados históricos, não precisamos de análises necessariamente em tempo real aqui, por isso podemos reduzir a quantidade de vezes que estes dados são inicializados seguindo essa estratégia de cache.
        - Em seguida os dados são processados dentro da aplicação do dashboard para exibir os gráficos e métricas.

Para todas as análises, o dashboard lê dinamicamente os resultados dos dados processados na pipeline. A fim de agilizar esta etapa final, alguns cálculos posteriores de métricas e manipulações de dataframes fazem uso de métodos *built-in* de `pandas`, tais como operações de *merge*, agrupamento e renomeação de colunas. Fizemos esta escolha pois a modelagem principal da pipeline já está definida e estas operações finais visam apenas preparar os resultados para a página do dashboard.

Uma aplicação em produção idealmente se aproveitaria das próprias capacidades do spark e da pipeline existente para também calcular os dados históricos, possivelmente utilizando a API de windowing do Spark Structured Streaming.

2.6.1. Motivações para a escolha

O **Streamlit** permite o desenvolvimento ágil de interfaces interativas sem a necessidade de conhecimentos avançados em frontend, o que acelerou a prototipação do dashboard. Além disso, sua integração com **Redis** e **PostgreSQL** via **Python** favorece uma arquitetura simples e eficiente para aplicações analíticas em tempo real.

3. Implantação em nuvem

A implantação em nuvem de todos os sistemas foi feita com base no serviço **Amazon Web Services.** A base da infraestrutura foi composta por **quatro instâncias EC2 (Elastic Compute Cloud)**.

- A primeira hospedou o Kafka, Zookeeper e Kafka-ui. Essa instância basicamente foi responsável por conter o broker, e ela foi separada das demais para evitar que a memória gasta não interferisse nem na produção dos dados e nem no desempenho da pipeline.
- A segunda hospedou o gerador de dados. Ao separar o gerador dos outros componentes, fomos capazes de mudar a quantidade de geradores em execução para fazer os testes de carga sem a preocupação de usar recursos que seriam usados pela pipeline.
- A terceira foi dedicada ao processamento com Apache Spark e armazenamento do cache com Redis. Assim como no caso dos produtores, isolar esses componentes permitiu que eles usassem todos os recursos da EC2 sem interferência.
- A quarta foi dedicada à execução do dashboard. Ao executar o dashboard em uma máquina virtual, fomos capazes de testar o restante dos serviços Amazon com mais facilidade e com o uso de um link público.

Para o armazenamento de dados, o serviço que utilizamos foi o **RDS** configurado com o PostgreSQL. A escolha por esse serviço se deu principalmente pela sua robustez: ele é especializado em bancos de dados e já conta com diversas otimizações internas das operações comuns do SQL. Ele também oferece uma forma fácil de configurar e levantar o serviço, uma característica muito importante e que permitiu que os esforços de desenvolvimento fossem empregados nos demais componentes da nuvem.

Apesar de existirem outros serviços da AWS especializados para os softwares que utilizamos, como o Amazon ElastiCache (que poderia ter sido usado no lugar de executar o Redis no mesmo EC2 que o Spark), eles não foram empregados por dois motivos principais: a falta de disponibilidade de alguns no laboratório para estudantes e pela complexidade envolvida com seu uso. Utilizando instâncias EC2, fomos capazes de reduzir a complexidade associada a criar novas instâncias de novos serviços e acessá-las, fator essencial para o deploy na nuvem.

Como é mostrado no vídeo de implantação da nuvem entregue junto ao relatório, as instâncias foram implementadas em diferentes contas da AWS. Em cada conta, todas as instâncias foram feitas por cima de um VPC padrão para a conta. Nesses VPCs foram configurados **Security Groups,** que funcionam como firewalls virtuais e liberam apenas as portas necessárias para comunicação dos componentes. Esses grupos também têm a capacidade de limitar os IPs que podem acessar a pipeline, mas para fins de teste, as instâncias foram abertas para acesso geral. Em um ambiente de produção, os grupos seriam configurados para aceitar conexões apenas de IPs confiáveis.

O acesso remoto às instâncias EC2 pelos integrantes do grupo foi realizado por meio de **pares de chaves SSH**, que permitiram a configuração e o monitoramento dos serviços de forma segura. O controle de permissões, em um ambiente de produção, seria feito pelo **IAM**, mas o gerenciamento de permissões não está disponível para o laboratório de estudantes. Por isso, instâncias do ECS não foram usadas e a implantação foi feita utilizando EC2 diretamente, sem orquestração de containers.

Durante a configuração, enfrentamos alguns desafios. A escolha de instâncias **t2.micro**, com apenas 1GB de RAM, exigiu ajustes e otimizações finas nas configurações do Docker Compose e dos próprios serviços. Outro problema surgiu na **configuração de rede**, pois os containers precisavam se comunicar entre instâncias EC2. Isso foi resolvido expondo os IPs externos de cada serviço. Além disso, o código original usava `localhost` como padrão de conexão, o que nos levou a configurar **variáveis de ambiente específicas para a AWS**, garantindo conectividade correta.

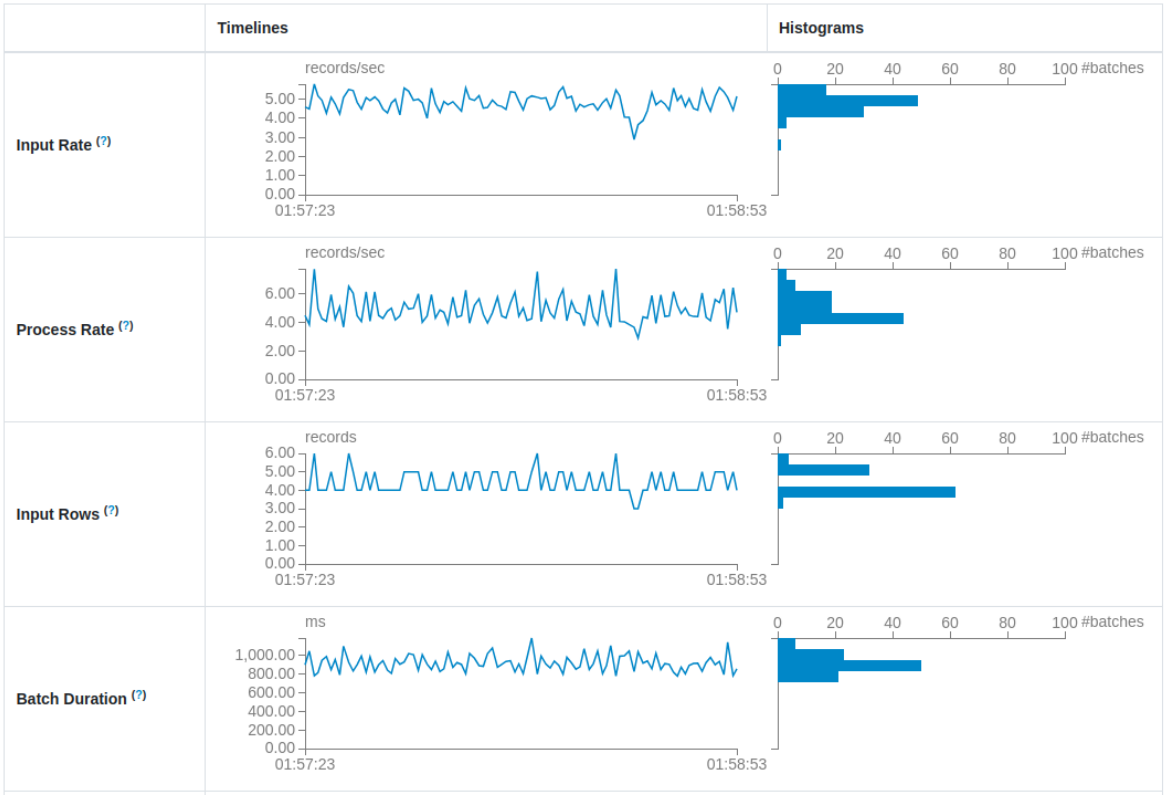
Os resultados da implantação foram mostrados no vídeo. O banco de dados foi o primeiro componente levantado, e foi populado usando o script de geração do container `db-seed`, mas executado localmente com os dados de conexão do RDS (como esse preenchimento do banco só precisa ser feito uma vez, não havia motivo para uma instância de nuvem para isso). Após isso, foi feito um teste com todos os outros serviços sendo executados em uma mesma instância do EC2, para garantir o funcionamento de todos na máquina virtual. A última etapa foi a separação em diferentes instâncias do EC2, pelos motivos citados acima.

A escolha da AWS se justifica por vários fatores. A **escalabilidade** permitiu a alocação independente de recursos por componente. O **gerenciamento facilitado** do banco via RDS eliminou a complexidade de manutenção. O **isolamento de rede** garantido pela VPC aumentou a segurança. A **flexibilidade** das instâncias EC2 nos permitiu ajustar recursos conforme a demanda.

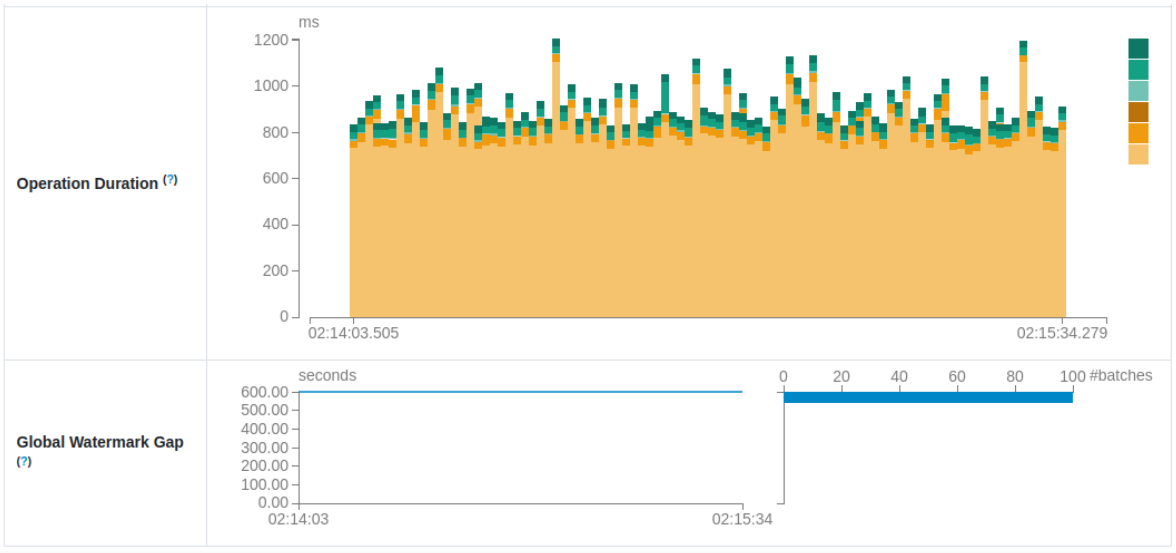
4. Resultados práticos

Para fazer os testes de carga e enumerar os resultados, utilizamos a UI do Spark, que sumariza informações de carga e processamento.

No primeiro teste, executamos uma instância do produtor de dados localmente com um delay de 10 milissegundos entre cada envio de transações. Após 5 minutos, observamos que a pipeline se manteve estável no processamento desses dados:

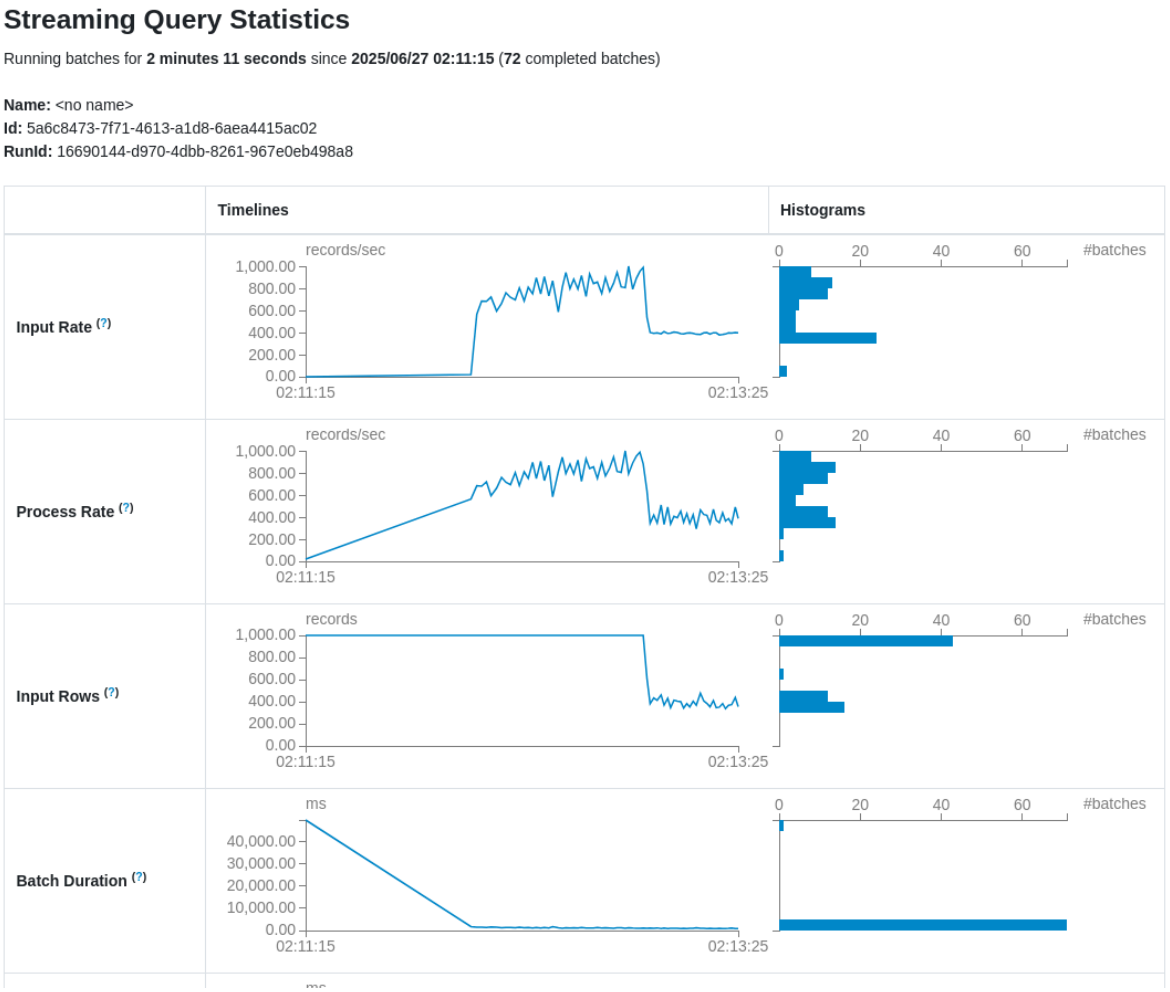


Nessas condições de envio, a duração de cada microbatch foi elencada como 1000 milissegundos pela UI:

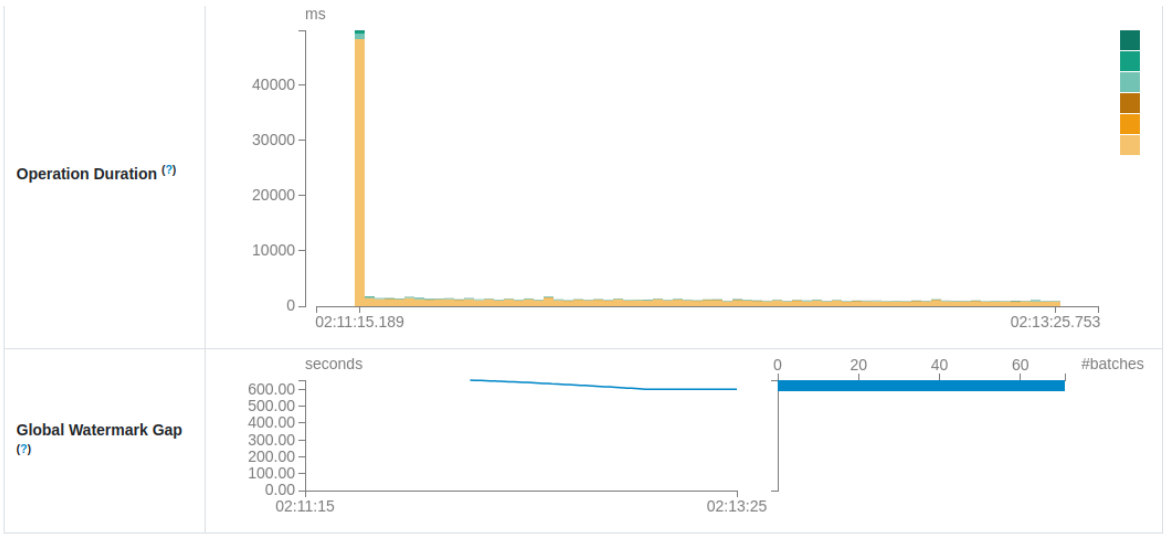


Como esse é um baixo volume de dados, os resultados eram esperados. O fato do produtor rodar localmente e enviar os dados via internet fez com que poucas transações fossem realmente enviadas por segundo.

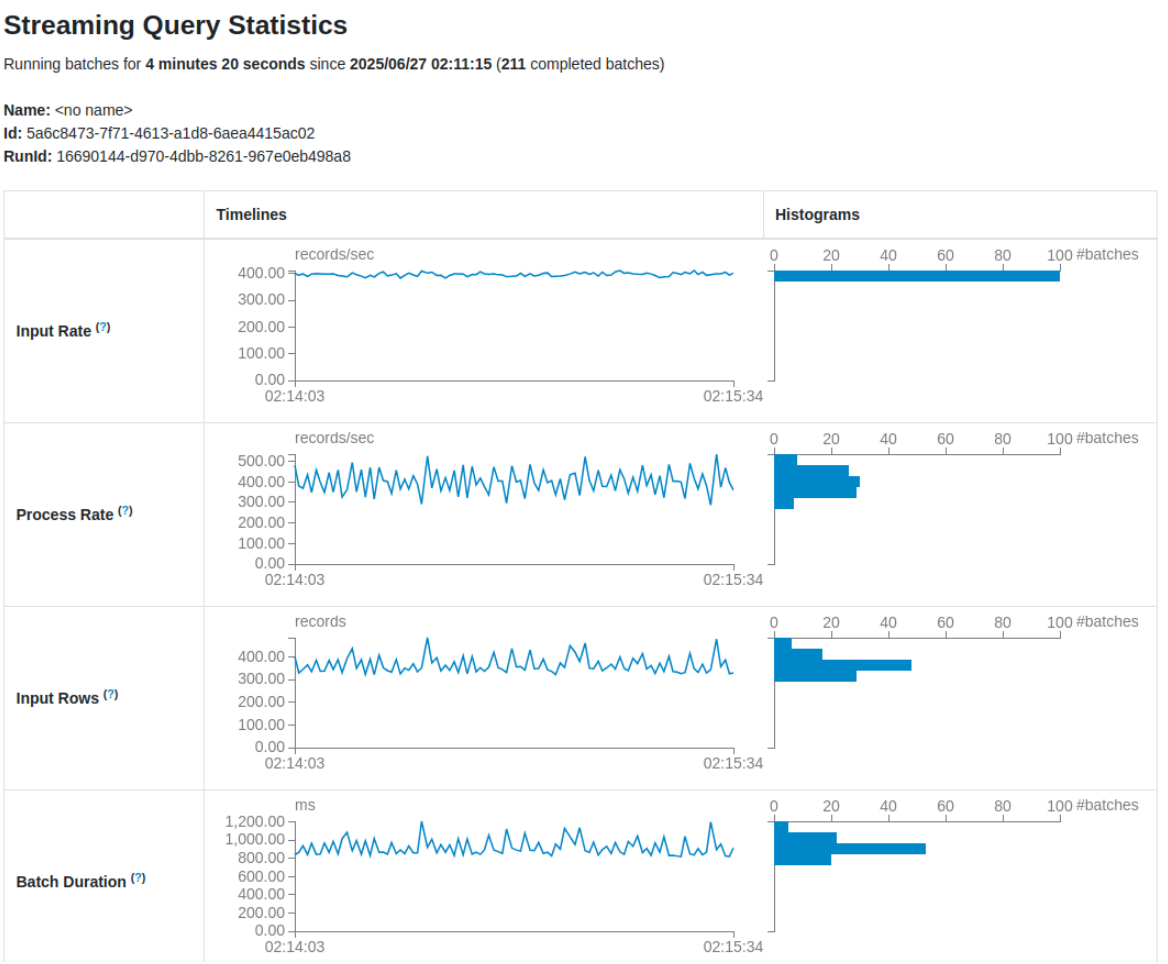
Para o segundo teste, executamos um produtor de dados na AWS sem nenhum delay entre os envios. Aqui, temos dois momentos de análise: após 2 minutos de execução e após 4. Ao executarmos a pipeline pela primeira vez, ela passou por um timeout e se reiniciou antes do produtor de dados começar o envio de dados. Como consequência, ela iniciou a segunda execução com dados já acumulados no tópico do Kafka, e os gráficos mostram como ela se comportou diante disso:



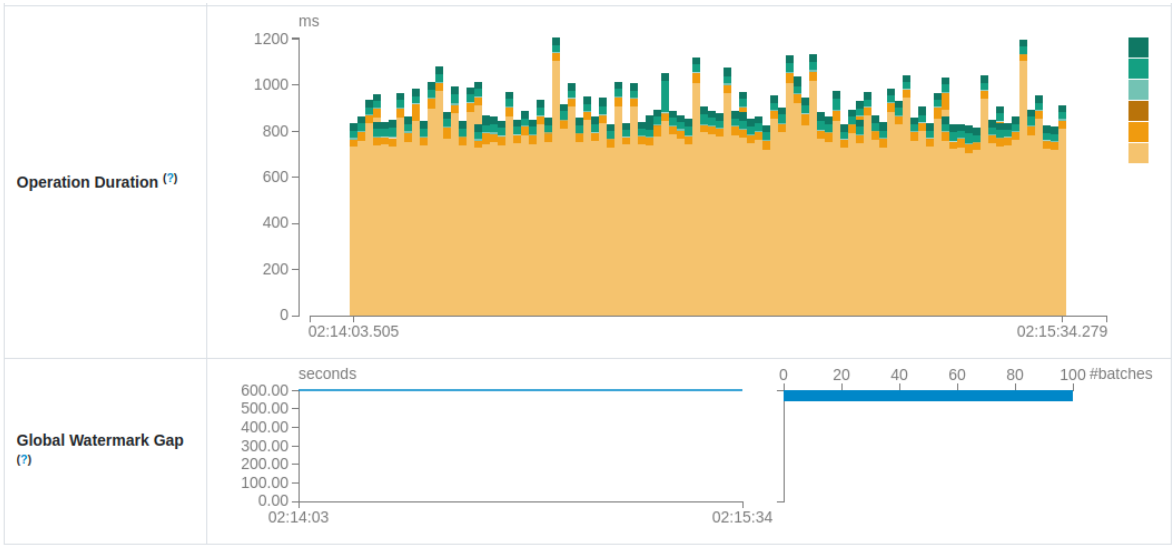
A pipeline Spark chega a processar até mil transações por segundo para “tirar o atraso” das transações e voltar a processar. Observamos que há um pico bem grande na duração das operações, correspondendo a diferença de tempo entre o começo da produção e da pipeline.



Após mais alguns minutos, a leitura é estabilizada e a pipeline começa a, de fato, processar os dados em tempo real. A taxa de processamento se mantém sempre flutuando um pouco em torno da taxa de produção, mas não ficando para trás de forma que prejudique a latência.



Apesar do volume de dados ser bem maior, a duração das operações segue a mesma. Isso demonstra o poder da engine Spark Structured Streaming, que lida muito bem com o aumento de transações e consegue empregar os recursos disponíveis de maneira eficiente.



Futuras direções de experimentos podem incluir testes com aumento constante de volume, para analisar como o Spark lida com taxas variáveis de entrada, e testes com quantidades massivas de volumes, para analisar qual a taxa máxima em que a engine pode operar. Esses segundos testes seriam ainda mais enriquecidos se diferentes instâncias de EC2 fossem testadas e comparadas entre si e entre serviços mais especializados, como o Amazon EMR.

Ainda assim, os testes realizados demonstram que a pipeline implementada foi capaz de gerenciar corretamente os recursos disponíveis para lidar com o volume de dados. Esta é uma característica muito importante para quase todos os sistemas em operação atualmente.