Q1

—

1. Alta Disponibilidade e Durabilidade ("Zero Downtime"):

Precisamos garantir que o banco de dados permaneça operacional continuamente e que os dados sejam robustos e resistentes a possíveis falhas.

Solução Proposta: Replicação Assíncrona

Arquitetura de Replicação: Utiliza-se o modelo *publish/subscribe* de replicação do PostgreSQL. Uma instância principal (Publisher), responsável por todas as operações de escrita (INSERT, UPDATE, DELETE), será configurada. Para otimizar a leitura, réplicas (Subscribers) de leitura (*read-only*) são posicionadas geograficamente próximas aos usuários (São paulo, Nova York e Londres). A replicação é baseada no *Write-Ahead Log* (WAL), que registra todas as modificações no banco. O servidor principal transmite esses registros de forma contínua para as réplicas, que os aplicam para se manterem sincronizadas. Além disso, é possível provisionar mais de uma instância para as réplicas, caso haja falha.

Failover e Recuperação: Em caso de falha da instância principal, uma das réplicas pode ser promovida para Publisher, garantindo a continuidade do serviço. Comandos como o `pg_promote` ou funcionalidades do provedor Cloud (como AWS RDS) facilitam esse processo. Para recuperação de desastres, poderíamos também implementar o Point-in-Time Recovery (PITR).

Trade-offs e considerações:

Disponibilidade vs. Consistência (CAP Theorem): A replicação de forma assíncrona favorece a alta disponibilidade em detrimento da consistência. Poderia haver replication lag entre a escrita no master e a atualização nas réplicas.

Escalabilidade: A arquitetura com um único master é limitada ao vertical scaling. O PostgreSQL, nativamente, não suporta replicação multi-master *(*escalabilidade horizontal).

Complexidade: A configuração de múltiplas réplicas aumenta a complexidade da gestão, mas permite alta disponibilidade para reads e writes.

Configs importantes para esse caso:

- `wal_level = logical`: Habilita a replicação lógica.
- `max_wal_senders`: Número de conexões de replicação simultâneas.

- `max_replication_slots`: Garante que o master não apague os registros WAL antes que sejam consumidos pelas réplicas.
- `archive_command`: Configura o arquivamento contínuo do WAL para recuperação
- `max_wal_size`: Tamanho do arquivo WAL para flush no disco (checkpoints automáticos).

2. Alta performance/ baixa Latência ("Alta Frequência de Leitura e Gravação"):

Precisamos suportar a alta demanda de writes/reads, para isso precisamos de uma arquitetura de baixa latência e alto throughput.

Solução Proposta: Distribuição Geográfica e Otimizações

Latência de Leitura: A presença de réplicas locais em São Paulo, Nova York e Londres garante que as consultas de leitura sejam executadas com baixa latência para os usuários em cada região.

Indexação e particionamento: Para as subscribers fazer a indexação de colunas mais utilizadas (PKs, FKs e colunas usadas em WHERE e JOINS). Isso aumentaria a velocidade de reads. Para as publishers, considerar o particionamento, pois poderia auxiliar com o throughput. Comandos como `pg_partman` ajudariam nesse caso.

Trade-off: Índices aceleram a leitura, mas consomem espaço em disco e adicionam um pequeno custo de performance às operações de escrita. É preciso escolhê-los cuidadosamente assim com partições. Se uma partição não for escolhida adequadamente poderemos ter problemas de partition imbalance. Além disso, ambos criam complexidades a mais.

3. Eficiência Operacional ("Time Pequeno de Dados"):

Com uma equipe reduzida, a automação e o uso de serviços gerenciados são cruciais para o funcionamento do banco.

Escolher um provedor cloud: Utilizar um serviço como Amazon RDS e Google Cloud SQL abstrai grande parte da complexidade de infraestrutura, pois o provedor gerencia tarefas de provisionamento de hardware, backups e segurança.

Trade-off: Reduz o controle granular e aumenta os custos, mas economiza tempo da equipe.

Monitoramento e Alertas: Implementar um sistema de monitoramento para acompanhar métricas vitais como uso de CPU, memória, I/O de disco, latência de replicação e

performance de consultas. Ferramentas do próprio provedor de nuvem (ex: Amazon CloudWatch) combinadas com as visões internas do PostgreSQL (`pg_stat_activity`, `pg_stat_replication`) são essenciais.

Automação com CI/CD: Criar pipelines de integração e entrega contínua (CI/CD) para agilidade das operações.

4. Segurança (Qualquer sistema):

Princípio do Menor Privilégio: Dar ao usuário apenas as permissões necessárias. Utilizar roles para agrupar permissões (ex: analista, desenvolvedor) e os comandos GRANT e REVOKE para gerenciar acessos a tabelas e operações específicas.

Controle de Acesso à Rede: O acesso ao banco de dados deve ser restrito a uma lista de IPs confiáveis. No cloud provider faríamos isso pelos Security Groups ou regras de firewall. No caso do postgresql, usamos o arquivo `pg_hba.conf` para controlar quais hosts podem se conectar.

Q2

—

Modelo de ER aqui e no arquivo **Q2 - ER Model**.

ER Model:

Client (clientID PK, clientName, address, email, phone, registrationDate)
Armazena os dados cadastrais de cada cliente.

Portfolio (portfolioID PK, clientID FK) : Representa um portfólio de investimentos vinculado a um cliente.

Transactions (transactionID PK, portfolioID FK, securityID FK, transactionType, transactionQuantity, transactionPrice, date, transactionFee, time): Registra cada operação de compra ou venda realizada em um portfólio.

PortfolioHoldings (portfolioID PK, FK, securityID PK, FK, date PK, portfolioQuantity): Mantém o histórico da quantidade de cada ativo em um portfólio em uma data específica.

Exchange (exchangeID PK, exchangeName, country, currency): Representa uma bolsa de valores, com suas informações principais.

Security (securityID PK, assetID FK, exchangeID FK, ticker): Representa a listagem de um ativo em uma bolsa específica, identificada por ticker.

Assets (assetID PK, isin, assetType): Define um instrumento financeiro globalmente, identificado pelo ISIN e tipo.

Stocks (assetID PK, FK, shareClass): Subtipo de ativo para ações, diferenciando classes de ações.

Funds (assetID PK, FK, fundType, managementFee, administrator, benchmark): Subtipo de ativo para fundos de investimento, incluindo dados de administração e benchmark.

Bonds (assetID PK, FK, faceValue, maturityDate, couponRate, rating, couponFrequency): Subtipo de ativo para títulos de dívida, com informações de valor de face, vencimento e cupons.

SecurityPrices (securityID PK, FK, date PK, priceClose): Histórico de preços por Security em cada data.

AssetEvents (eventID PK, assetID FK, eventType, date, amount): Registra eventos financeiros associados a um ativo, como dividendos, cupons ou splits.

Observações importantes:

Quantidade vs. Peso: Optamos por armazenar a **quantidade em carteira** em vez do peso percentual. O peso é um dado derivado (qty / Σ(qty·preço)) e não deve ser adicionado para evitar inconsistências e update anomaly. Assim, garantimos o princípio de **raw facts e single source of truth**.

Fonte da quantidade: Assumimos que a informação de **quantidade em carteira** (**PortfolioQuantity**) vem do administrador, por isso criamos a junction table **PortfolioHoldings**. Caso contrário, poderíamos calculá-la a partir das transações. Essa redundância é aceitável porque tratamos o valor informado pelo administrador como raw fact e podemos ainda usá-lo para fazer o data quality dos dados contra as transações.

Vários mercados: Como um mesmo ativo pode estar listado em diferentes bolsas, criamos as entidades **Exchange** (junction table) e **Securities** para modelar a relação M:M entre ativos e bolsas. Isso também permite também separar o **ticker** (dependente da bolsa) do **ISIN** (identificador global).

Supertipo e subtipos: Modelamos ativos como um **supertipo** (**Assets**) com **subtipos** (Stocks, Bonds, Funds), de modo a representar características específicas de cada classe de ativo sem redundância.

Tabela AssetEvents: criada supondo que não temos o preço ajustado das securities, nesse caso, usamos os eventos corporativos para cálculo juntamente com o preço de fechamento.

Q3

—

Schema aqui e no arquivo **Q3 Star Schema**

Schema e descrição:

DimProduto: skproduto, idproduto, nomeproduto, categoria, start date, end_date

Descrição: Tabela com informações do produto (contexto do que foi vendido).
Granularidade: uma linha por **idproduto por período de validade** (SCD2: start_date, end_date).
Keys: PK: skproduto **AK:** (idproduto, start_date)

DimClient: skcliente, idcliente, nome, email, telefone, start date, end_date

Descrição: Informações cadastrais/históricas do cliente.
Granularidade: uma linha por **idcliente por start_date** (SCD2).
Keys: PK: skcliente **AK:** (idcliente, start_date)

DimFuncionario: skfuncionario, idfuncionario, nome, cargo, start date, end_date, skloja

Descrição: Informações históricas do funcionário.
Granularidade: uma linha por **idfuncionario por start_date** (SCD2).
Keys: PK: skfuncionario **AK:** (idfuncionario, start_date)

Dimloja: skloja, idloja, nome, localização, gerente, start date, end_date

Descrição: Informações históricas da loja.
Granularidade: uma linha por **idloja por start_date** (SCD2).
Keys: PK: skloja **AK:** (idloja, start_date)

Dimdata: skdata, diasemana, mês, trimestre, ano, feriado

Descrição: Calendário corporativo com diferentes granularidades para as datas.
Granularidade: uma linha por dia.
Keys: PK: skdata

FactSales: skproduto (fk), skclient (fk), skfuncionario (fk), skloja (fk), skdata (fk), quantidade, valor total, idvenda

Descrição: Tabela para a venda dos produtos.
Granularidade: uma linha por (idvenda x produto)

Keys: FKs: skproduto, skcliente, skfuncionario, skloja, skdata **DD:** idvenda (degenerate, sem FK)

FactEstoque: skproduto, skloja, skdata, estoque

Descrição: Snapshot de estoque.
Granularidade: uma linha por (produto × loja × data) no corte definido (ex.: fim do dia, fim do mês).
Keys: PK: (skproduto, skloja, skdata), **FKs:** skproduto, skloja, skdata

ER → Star Decisions:

Identificar fatos: Eventos/medidas/métricas quantitativas. Já que o processo central de análises é venda, a tabela de vendas do modelo ER virou a tabela de fatos

Identificar “dimensões”: as dimensões dão o contexto para os fatos, por exemplo, um produto vendido possui uma quantidade (métrica quantitativa), mas também possui um nome e uma categoria (contexto, medida qualitativa) e são denormalizadas para facilitar interpretação, análises, queries etc.

Para a coluna preço unitário, decidimos não adicioná-la, pois é um fato não aditivo. Isso significa que não podemos somar através das outras dimensões (Produto, Cliente, etc). Segundo as melhores práticas de normalização para Data Warehouses, devemos manter somente fatos semi/aditivos nas tabelas por questões de consistência (todos os fatos podem ser somados através de algumas ou todas as dimensões, isso gera maior compreensão e entendimento) e simplifica as colunas (menor número de colunas usadas). Fatos aditivos servem para fazer agregações (SUM, COUNT, etc) que são operações otimizadas em data warehouses. Além disso, medidas não aditivas tornam difíceis análises em hierarquias, quando se quer analisar uma coluna em diferentes níveis de granularidade. Por fim, preço pode ser obtido fazendo valor total / quantidade, ou seja, também é dado derivado.

Criação da tabela DimData que permite agregações e análises em diferentes hierarquias de tempo (semana, mês, ano etc). Bom para identificarmos alguma tendência ou sazonalidade.

A coluna Data da venda foi removida da tabela FactSales, assumindo que ela possui o mesmo valor que a coluna SKData da tabela DimData. Se a data da venda fosse um timestamp com hora:min:seg poderíamos mantê-la como uma nova coluna.

Adição de StartDate e EndDate na maioria das colunas de dimensão para modelagem SCD2 assim como SurrogateKeys. SurrogateKeys também são importantes para termos um histórico de mudança, além de desacoplar os dados do sistema legado de ER. Por exemplo, se na tabela DimProduto a categoria do produto muda, sem mudar o ID do produto, adicionamos uma nova linha com uma nova SurrogateKey, apesar do ID não ter mudado.

Estoque como uma “periodic snapshot fact table”, que nada mais é do que um snapshot periódico, atualizado em toda SKData, pois é um fato semi aditivo (só pode ser somado

através das dimensões loja e produto). Separamos da FactSales, apesar de estar contido na sua granularidade, pois queremos guardar o estoque independentemente se houve uma venda ou não.

Adição de SKLoja na tabela de funcionários, pois podemos querer saber se um funcionario mudou de loja ou não (mais porque tinha no ER original), mas poderiamos excluir esse campo se pensarmos que já temos essa informação na FactSales, só teríamos que esperar uma venda ocorrer. Assumimos que um funcionário só pode trabalhar em uma loja.

Adição de IDVenda como uma degenerate dimension na FactSales. É importante mantê-la na tabela de fatos, pois agrupa fatos relevantes que ocorreram juntos. Por exemplo: uma venda com vários produtos.

Q4

—

Desenho do pipeline no arquivo **Q4 arquitetura**

Trigger e Orchestration

A automação do pipeline é iniciada por eventos, garantindo o processamento assim que os dados chegam no S3. Poderíamos também usar um trigger por hora, mas escolhemos por evento para ser conservador (talvez não o mais prático de se fazer).

S3 Event-Based Trigger → AWS EventBridge → AWS Step Functions

Fluxo: Um novo arquivo depositado em um bucket S3 gera uma notificação. O EventBridge captura essa notificação, filtra por critérios específicos (ex: nome do bucket, sufixo `.csv`) e aciona o AWS Step Functions.

Step Functions: escolhido pela simplicidade e integração nativa para orquestrar um pipeline único e linear, sendo mais adequado que o **Airflow** para este escopo específico que não exige backfills complexos. A própria Step Function pode também realizar algumas validações de metadados iniciais (ex: tamanho do arquivo, formato do nome).

Processamento e Transformação de Dados (ETL)

O processamento pesado dos dados é feito pelo AWS Glue que usa Spark.

AWS Glue (com PySpark): Sendo um serviço serverless e nativo da AWS, elimina a necessidade de gerenciar clusters. É projetado para processamento pesado e paralelo de grandes volumes de dados, com integração nativa ao orquestrador (Step Functions) e ao catálogo de dados (Glue Data Catalog). O downside é que ele gera dependência do ecossistema AWS (vendor lock-in). Se o padrão da equipe fosse Databricks, seria

necessário criar uma camada de integração adicional (ex: uma função Lambda) para acionar o job.

Jobs:

- **AWS Glue Job 1 (Bronze → Silver):** Focado em data quality, data cleaning e normalizações. Aplica o schema correto (nomes e tipos de dados), realiza a limpeza de dados (nulos, espaços em branco, formatação) e executa validações de qualidade com **Great Expectations (GX)** para bloquear dados ruins (ex: IDs duplicados, valores fora do intervalo esperado).
- **AWS Glue Job 2 (Silver → Gold):** Focado em regras de negócio, agregações e modelagem. Transforma os dados limpos em um modelo otimizado para análise (ex: Star Schema), calculando indicadores e preparando os dados para consumo por ferramentas de BI. O resultado é particionado por data e salvo em formato Parquet para otimizar a performance de futuras consultas.

Data loading / Serving (Data Warehouse)

Lambda para Loading no warehouse: Uma função Lambda simples é usada para fazer o load dos dados do S3 (camada Gold) para uma tabela de staging no Snowflake, realizando em seguida um *UPSERT* na tabela final (poderíamos fazer um *APPEND* também, mas corremos o risco de deixar dados reprocessados na tabela) .

Snowflake: arquitetura de Virtual Warehouses permite que diferentes cargas de trabalho (ex: pipeline de ELT e consultas de BI) operem com recursos computacionais isolados, evitando conflitos de performance. É multi-cloud, o que reduz o vendor lock-in da camada de dados. Possui configuração mínima, gerenciamento de infraestrutura abstraído e suspensão automática para controle de custos. Se quisermos mudar, posteriormente, como o processamento dos dados é feito não teríamos grandes problemas. O downside é que a integração com serviços AWS não é tão nativa quanto a do Amazon Redshift, por exemplo. Existe um risco de custos elevados se os Virtual Warehouses não forem monitorados e gerenciados ativamente.

Ferramentas de Suporte:

Amazon CloudWatch: Solução nativa para monitoramento de todo o pipeline (Step Functions, Glue, Lambda), permitindo a criação de alarmes para falhas, tempos de execução anômalos ou consumo excessivo de recursos.

Great Expectations (GX): Escolhido pela sua abordagem declarativa e por ser open source, evitando lock-in. Permite definir expectativas claras sobre os dados e gera documentação automática (Data Docs), melhorando a governança e a confiabilidade.

Q5

—

Assumindo que precisamos disponibilizar os dados da maneira mais rápida possível, escolheríamos o caminho “hot path” / “cold path” (baseado em hot data e cold data). (Arquitetura no arquivo **Q5 arquitetura - hot & cold**)

Hot Path: Kafka topic → *Apache Flink (processamento inicial simples) → Insert no ElasticSearch → Kibana (mostrar dados)*

Cold path: Kafka topic → *Apache flink (ingest no datalake bronze) --> Apache Structured Streaming (AWS Glue Streaming) para processamento da bronze --> silver → gold --> Delta tables -- > AWS Athena (queries)→ QuickSights (dashboard)*

Hot Path: Disponibilização dos dados o mais rápido possível

- ElasticSearch: baixa latência para consultas, escalabilidade horizontal e SQL queries.
- Kibana: Integração com ElasticSearch, visualização de dados em tempo real, permite visualizações sem conhecimento de SQL, alertas automáticos por email (ex: quando preço do ativo atinge um certo target)

Cold Path: Focado em processamento mais complexo, com maior validação dos dados, cálculo de indicadores adicionais e modelagem de dados. Bom para análises históricas e backtesting.

- Apache Flink: baixa latência, tolerância a falhas semântica de entrega sem duplicados, que gera consistência nos dados, high throughput, unificar hot e cold path num framework único.
- Apache Structure Streaming (com AWS Glue jobs) permite o processamento de grande quantidade de dados em tempo real. Processaria os dados desde a camada bronze até a gold.
- Delta tables: Integração com AWS Athena e propriedades ACID para tabelas.
- AWS Athena: Consultar diretamente as delta tables, integração com ferramentas de BI.
- QuickSight: visualização sem ter que administrar nenhum serviço novo, podemos fazer o embedded de visualizações no portal, aplicações e outros websites.

Análise do Pipeline:
Apesar de servirmos rapidamente os dados em um caminho, há complexidade operacional em gerenciar diversas tecnologias e um custo associado a isso. Além disso, poderiam haver inconsistências nos dados salvos na base snowflake os servidos nos ElasticSearch e

Kibana, pois estaríamos fazendo dois processamentos distintos (um no job do Apache flink e outro no job do Apache Structure Streaming). Usamos a combinação de delta tables + AWS Athena para tentar compensar os custos do hot path. Os dados viveriam dentro do S3 e não de um data warehouse.

Assumindo que os dados precisam ser disponibilizados de maneira rápida, mas não a mais rápida possível poderíamos ter um “warm path”:

Warm Path: *Kafka topic --> AWS Glue Streaming → Ingestação Bronze → Processing bronze --> silver --> gold → Snowflake*

- AWS Glue Streaming*:* podemos usar o serviço de AWS Glue Streaming para consumir os dados direto da fonte de streaming e para todo o processamento bronze → silver → gold. Possui integração nativa com o Kafka. Como só usamos o AWS Glue para processamento, podemos fazer a orquestração utilizando esse próprio serviço (AWS Glue Studio).
- Snowflake: permite consultas mais complexas, scaling, time travel, facilidade de integração com ferramentas de BI, otimização de queries e cache.

Análise do Pipeline:
A idéia aqui é ter um approach intermediário. Servimos os dados de maneira mais rápida, mas não a mais rápida possível (porém mais do que no “cold path”). O Serviço de Streaming da AWS Glue é rápido, mas de maior latência quando comparado com o Apache Flink. Diminuímos a complexidade operacional, pois teríamos um pipeline centralizado e com menos tecnologias. O Snowflake permitiria consultas mais rápidas, apesar de haver o overhead de subir os dados do data lake para o data warehouse.

obs: em ambos os pipelines também usaríamos ferramentas para monitoramento de data quality, assim como foram utilizados no pipeline de Q4. Omitimos aqui por simplicidade.