Resolução - Atividade 05 - CRUD vs Evento

Definições e Conceitos

Pré pesquisa sobre os conceitos de CRUD (Create, Read, Update, Delete) e Event Sourcing.

CRUD

CRUD é um acrônimo que representa as quatro operações básicas de persistência de dados em sistemas de banco de dados.

- O que é persistência de dados? De forma grosseira é armazenar dados no banco. Mas não só isso, é ler, atualizar e excluir esses dados também, é essa interação precisa ser PERSISTENTE, ou seja, durar no tempo como após a reinicialização do sistema, por exemplo.

Essas operações são:

- **Create (Criar)**: Adicionar novos registros ao banco de dados.
- **Read (Ler)**: Recuperar dados existentes do banco de dados.
- **Update (Atualizar)**: Modificar dados existentes no banco de dados.
- **Delete (Excluir)**: Remover dados do banco de dados.

Basicamente o CRUD é desenvolvido para manipular diretamente o estado atual dos dados armazenados em um banco de dados relacional ou não relacional. O desenvolvedor back-end geralmente implementa essas operações que conectam a aplicação ao banco de dados - o famoso dev crudzeiro.

Event Sourcing - Ou traduzindo, Armazenamento por Eventos, Origem por Eventos, Fonte de Eventos...

No CRUD a manipulação é direta no estado atual dos dados e o histórico das mudança não é armazenado - a menos que o desenvolvedor implemente isso manualmente - e apesar de alguns sistemas de banco de dados oferecem algo parecido com o histório de transações para replicação e rollback o intuito é do event sourcing é diferente, é para ter o histórico mesmo de todas as mudanças.

O Event source me lembra muito o modelo Fato x Dimensão que é muito usado em Data Warehouse e Data Lakes para análises de dados, onde o fato armazena os eventos de negócio (como vendas, transações) e as dimensões armazenam os atributos relacionados (como tempo, localização, produto). Então é uma tabela "de atributos" e uma tabela "de eventos" com as chaves relacionadas. Enfim... Acho que tem alguma relação conceitual. E também lembra o apache iceberg que armazena as mudanças de dados. Mas esses arquiteturas são mais focadas em análises de dados, enquanto o event sourcing é mais focado em sistemas transacionais... Voltando...

"Event-sourcing, centraliza a estrutura da aplicação em eventos, naturalmente facilitando a implementação do EDA (Event-driven Architecture)" - https://medium.com/@marcelomg21/event-sourcing-es-em-uma-arquitetura-de-microsserviços-852f6ce04595 - Marcelo M. Gonçalves

Cada mudança de estado gera um evento que é armazenado de forma imutável. O estado atual do sistema pode ser reconstruído "reaplicando" esses eventos em ordem cronológica. Isso oferece várias vantagens,

como auditabilidade completa, capacidade de reverter para estados anteriores e facilidade na implementação de funcionalidades como "time travel" (viagem no tempo) e "event replay" (reexecução de eventos) via checkpoints - realmente muito parecido com Apache Iceberg.

Event-Store: Esse é o banco de dados para armazenar os eventos. O acesso a esses eventos é feito geralmente via APIs ou bibliotecas específicas que dão uma abstração para trabalhar com eventos.

Quando um evento é persistido, um broadcast é feito para notificar outros componentes do sistema sobre essa mudança e os componentes "inscritos" podem reagir a esses eventos, atualizando seus próprios estados ou executando outras ações conforme necessário.

Diretamente do artigo:

> Naturalmente, sempre que um evento é armazenado no event-store, automaticamente também é publicado em um canal de transmissão, sendo difundido e chegando aos componentes inscritos no evento, sendo então estimulados por ele (event-handlers).
> O Event-store pode ser considerado o backbone (espinha dorsal) para aplicações baseadas em arquitetura de microsserviços utilizando EDA (Event-driven Architecture).
> - Marcelo M. Gonçalves

Segundo o Marcelo M. Gonçalves a vantagem do Event Sourcing em relação ao CRUD pode ser resumida em 3 pontos principais:

1. Auditabilidade completa: Cada mudança de estado é registrada como um evento imutável, permitindo rastrear todas as alterações feitas no sistema ao longo do tempo.
2. Reversibilidade: A capacidade de reverter para estados anteriores do sistema, facilitando a recuperação de dados e a correção de erros.
3. Facilidade na implementação de funcionalidades avançadas: Funcionalidades como "time travel" (viagem no tempo) e "event replay" (reexecução de eventos) podem ser implementadas de forma mais simples, permitindo análises históricas e testes de cenários.

Os benefícios ao utilizar um estilo de persistência baseado em uma fonte de eventos incluem a possibilidade de logs de auditoria com maior precisão se comparado a modelos de persistência tradicionais, incluindo a facilidade na composição de queries temporais, por conta do histórico completo mantido.

Recomenda-se que o estado atual da aplicação seja mantido em um data-store separado, seja desnormalizado ou relacional, mantendo o event-store como base de auditoria para execução de processamentos especiais e isolados nos dados.

- Mais uma vez uma similaridade com o modelo Fato x Dimensão, onde o fato armazena os eventos e a dimensão armazena o estado atual.

Os snapshots (checkpoints)... Por que existem?

- Aqui entra uma estratégia bem interessante para otimizar a reconstrução do estado atual do sistema quando há muitos eventos.
    - Exemplo: Muitas alteações foram feitas num mesmo registros, então cada alteração gera um evento que será armazenado no meu event-store... E ai a coisa pode crescer demais. Então para otimizar uma foto é "tirada" do estado atual do sistema em um ponto específico no tempo e esse ponto seria um checkpoint. Assim, ao reconstruir o estado atual, o sistema pode começar a partir do checkpoint e aplicar apenas os eventos que ocorreram após esse ponto, reduzindo o número de eventos que precisam ser processados. É quase uma partição de dados a nível event-store.

```
O event-store funciona com append-only, ou seja, os eventos são apenas
adicionados e nunca modificados ou excluídos. Isso garante a integridade
do histórico de eventos. Se algum evento precisar ser "removido" por
questões legais ou de privacidade, uma abordagem comum é adicionar um novo
evento que indica a remoção ou anonimização dos dados relacionados, em vez
de excluir o evento original. - MAIS UMA SIMILARIDADE COM O APACHE
ICEBERG, BIZARRO.
```

Enfim, lendo o artigo do Marcelo M. Gonçalves outras threads sao abertas como EDA e CQRS, mas isso foge do escopo dessa atividade...

Resumo

Event Sourcing é um padrão arquitetural onde as mudanças de estado de um sistema são armazenadas como uma sequência de eventos imutáveis. Em vez de armazenar apenas o estado atual dos dados. Uma arquitetura parecida na camada analítica é a Fato x Dimensão que tem um conceito parecido.

ENUNCIADO DA ATIVIDADE

Cenário

Você é arquiteto de dados responsável por escolher a forma de persistência em dois cenários distintos. Analise cada cenário e justifique se usaria CRUD ou Event Sourcing, destacando vantagens e limitações.

Sua organização é uma agência reguladora precisa armazenar informações sobre reclamações de consumidores contra outras empresas sobre algum produto. O ciclo de vida de uma reclamação é Registrada → Em Análise → Em Judice → Concluída.

PARTE 1 - CRUD

> 1. Modele a relação reclamação;
>
> 2. Use operações INSERT, UPDATE e DELETE para gerenciar as reclamações com dados artificiais. Mantenha 5 reclamações (no mínimo) em diferentes momentos do ciclo de vida;
>
> 3. Armazene apenas o estado atual de cada reclamação.
>
> - Perguntas orientadoras:

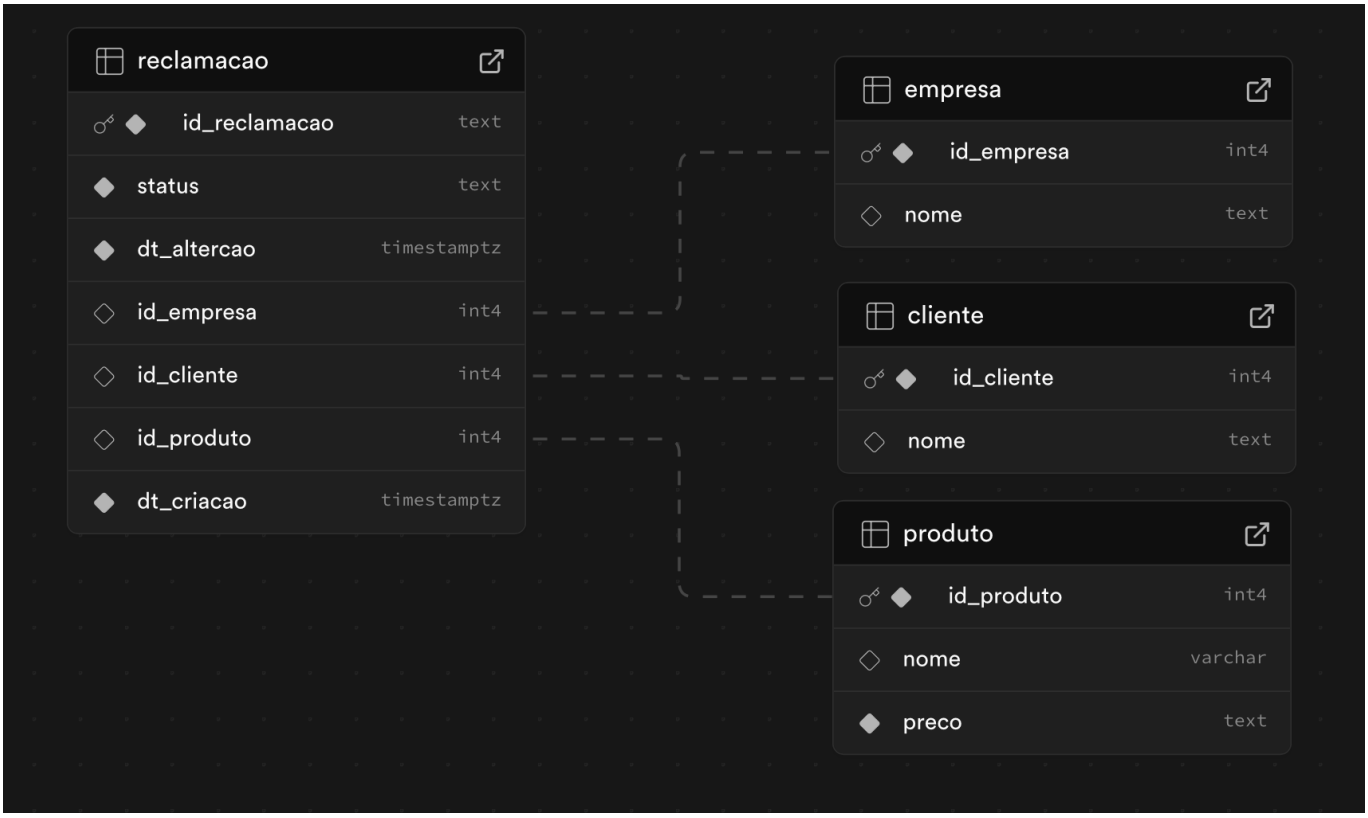
> Como ficaria a consulta para saber o status atual de uma reclamação?

> É possível responder "quanto tempo uma reclamação levou em cada etapa"?

Resolução - Parte 1 - CRUD

1. Modelo da relação reclamação:

Usei o Supabase para modelar a relação de reclamação com as tabelas relacionadas de cliente, empresa e produto. Ele gera o script SQL a partir do desenho, só clicar em "Database" e depois em "Copy as SQL".




```sql
-- WARNING: This schema is for context only and is not meant to be run.
-- Table order and constraints may not be valid for execution.

CREATE TABLE public.cliente (
  id_cliente integer NOT NULL,
  nome text,
  CONSTRAINT cliente_pkey PRIMARY KEY (id_cliente)
);
CREATE TABLE public.empresa (
  id_empresa integer NOT NULL,
  nome text,
  CONSTRAINT empresa_pkey PRIMARY KEY (id_empresa)
);
CREATE TABLE public.produto (
  id_produto integer NOT NULL,
  nome character varying,
  preco text NOT NULL DEFAULT '0'::text,
  CONSTRAINT produto_pkey PRIMARY KEY (id_produto)
);
```

```
CREATE TABLE public.reclamacao (
  id_reclamacao text NOT NULL,
  status text NOT NULL,
  dt_ult_alteracao timestamp with time zone NOT NULL,
  id_empresa integer,
  id_cliente integer,
  id_produto integer,
  dt_criacao timestamp with time zone NOT NULL,
  CONSTRAINT reclamacao_pkey PRIMARY KEY (id_reclamacao),
  CONSTRAINT reclamacao_id_empresa_fkey FOREIGN KEY (id_empresa)
REFERENCES public.empresa(id_empresa),
  CONSTRAINT reclamacao_id_cliente_fkey FOREIGN KEY (id_cliente)
REFERENCES public.cliente(id_cliente),
  CONSTRAINT reclamacao_id_produto_fkey FOREIGN KEY (id_produto)
REFERENCES public.produto(id_produto)
);
```

```
-- Populando tabelas relacionadas
INSERT INTO public.cliente (id_cliente, nome) VALUES
(1, 'Cliente 1'),
(2, 'Cliente 2'),
(3, 'Cliente 3'),
(4, 'Cliente 4'),
(5, 'Cliente 5');

INSERT INTO public.empresa (id_empresa, nome) VALUES
(1, 'Empresa 1'),
(2, 'Empresa 2'),
(3, 'Empresa 3'),
(4, 'Empresa 4'),
(5, 'Empresa 5');

INSERT INTO public.produto (id_produto, nome, preco) VALUES
(1, 'Produto 1', '10.00'),
(2, 'Produto 2', '20.00'),
(3, 'Produto 3', '30.00'),
(4, 'Produto 4', '40.00'),
(5, 'Produto 5', '50.00');
```

2. Operações INSERT, UPDATE e DELETE para gerenciar as reclamações:

Para gerenciar as reclamações achei conveniente criar procedures para cada operação de CRUD.

- Procedure para atualizar reclamação:

```
CREATE OR REPLACE PROCEDURE atualizar_status_reclamacao(id_da_reclamacao
text, novo_status text)
AS $$
```

```
BEGIN
  UPDATE reclamacao r
  SET status = novo_status,
      dt_ult_alteracao = NOW()
  WHERE r.id_reclamacao = id_da_reclamacao;
END;
$$ LANGUAGE plpgsql;
```

- Procedure para deletar reclamação:

```
CREATE OR replace procedure deletar_reclamacao(id_da_reclamacao text)
as $$
begin
    delete from RECLAMACAO
    where ID_RECLAMACAO = id_da_reclamacao;
end;
$$ language plpgsql;
```

- Procedure para inserir reclamação:

```
CREATE OR REPLACE PROCEDURE inserir_reclamacao(id_da_reclamacao text,
status_inicial text, id_da_empresa integer, id_do_cliente integer,
id_produto integer)
AS $$
BEGIN
  INSERT INTO RECLAMACAO (ID_RECLAMACAO, STATUS, DT_ULT_ALTERACAO,
ID_EMPRESA, ID_CLIENTE, ID_PRODUTO, DT_CRIACAO)
  VALUES (id_da_reclamacao, status_inicial, NOW(), id_da_empresa,
id_do_cliente, id_produto, NOW());
END;
$$ LANGUAGE plpgsql;
```

Criando 5 reclamações em diferentes estados do ciclo de vida:

- R1

```
call inserir_reclamacao('R1', 'Registrada', 1, 1, 1);

call atualizar_status_reclamacao('R1', 'Em Análise');
```

- R2

```
call inserir_reclamacao('R2', 'Registrada', 2, 2, 2);
```

- R3

```
call inserir_reclamacao('R3', 'Registrada', 3, 3, 3);
call atualizar_status_reclamacao('R3', 'Em Análise');
call atualizar_status_reclamacao('R3', 'Em Judice');
```

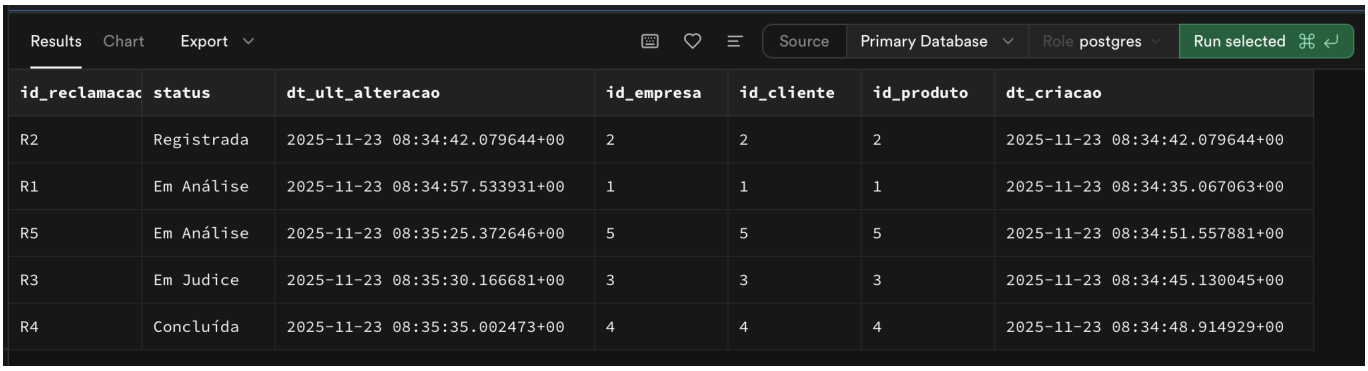
- R4

```
call inserir_reclamacao('R4', 'Registrada', 4, 4, 4);
call atualizar_status_reclamacao('R4', 'Em Análise');
call atualizar_status_reclamacao('R4', 'Em Judice');
call atualizar_status_reclamacao('R4', 'Concluída');
```

- R5

```
call inserir_reclamacao('R5', 'Registrada', 5, 5, 5);
call atualizar_status_reclamacao('R5', 'Em Análise');
```

Rodando em intervalos de segundos para diferenciar os timestamps:



id_reclamacao	status	dt_ult_alteracao	id_empresa	id_cliente	id_produto	dt_criacao
R2	Registrada	2025-11-23 08:34:42.079644+00	2	2	2	2025-11-23 08:34:42.079644+00
R1	Em Análise	2025-11-23 08:34:57.533931+00	1	1	1	2025-11-23 08:34:35.067063+00
R5	Em Análise	2025-11-23 08:35:25.372646+00	5	5	5	2025-11-23 08:34:51.557881+00
R3	Em Judice	2025-11-23 08:35:30.166681+00	3	3	3	2025-11-23 08:34:45.130045+00
R4	Concluída	2025-11-23 08:35:35.002473+00	4	4	4	2025-11-23 08:34:48.914929+00

3. Armazenar apenas o estado atual de cada reclamação.

Feito no passo anterior, onde cada reclamação tem apenas o estado atual armazenado na tabela RECLAMACAO.

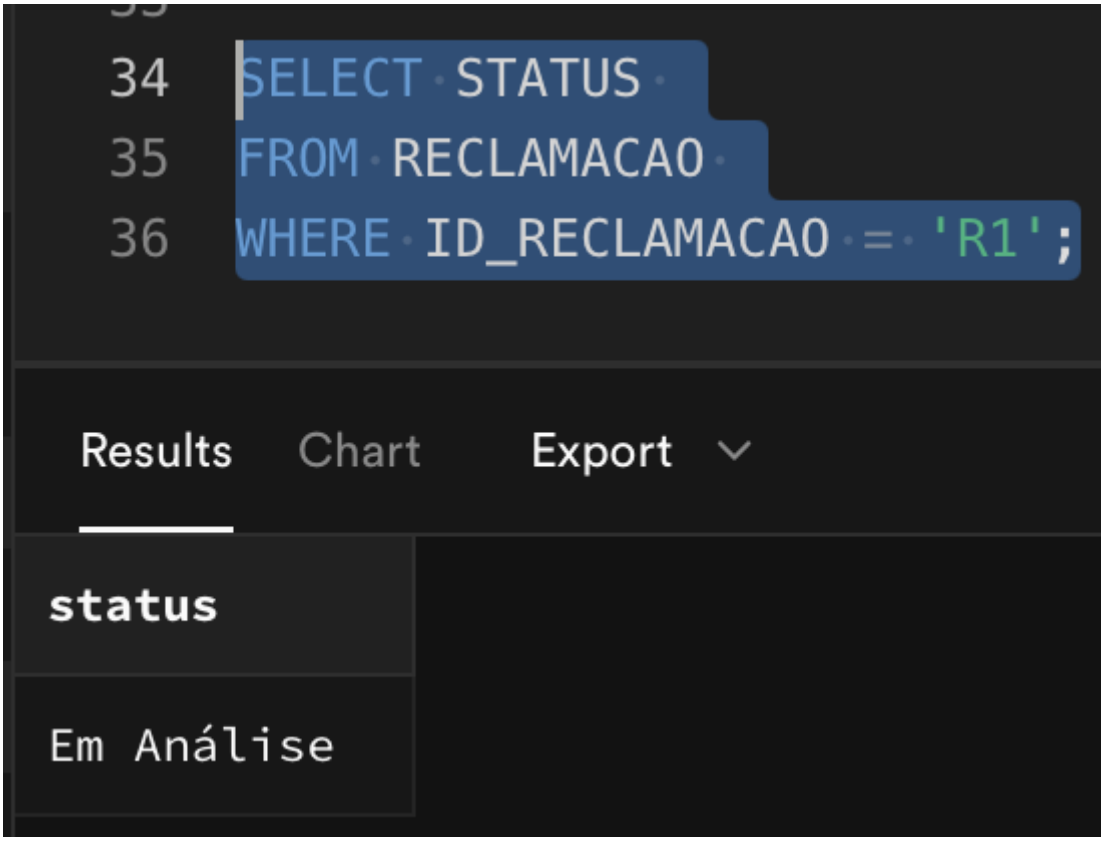
Respondendo as perguntas orientadoras:

> Como ficaria a consulta para saber o status atual de uma reclamação?

Consulta para saber o status atual de uma reclamação: Basta fazer a consulta simples buscando o ID da reclamação desejada.

```
SELECT STATUS
FROM RECLAMACAO
```

```
WHERE ID_RECLAMACAO = 'R1';
```



> É possível responder "quanto tempo uma reclamação levou em cada etapa"?

Não, com o modelo CRUD tradicional não é possível rastrear o tempo que uma reclamação levou em cada etapa, pois apenas o estado atual é armazenado, toda vez que o status precisa ser atualizado o regristro é sobrescrito.

Mas possivel é possível saber delta do tempo da criação até o estado atual - que poderia ser calculado com a diferença entre DT_CRIACAO e DT_ULT_ALTERACAO, mas não o tempo em cada etapa individualmente.

PARTE 2 - EVENT SOURCING

> 1. Modele uma relação de eventos;

> 2. Crie eventos que representam mudança no ciclo de vida das reclamações. Faça isso para 5 reclamações no mínimo, mantendo-as em diferentes estados do ciclo de vida;

> 3. Reconstrua o estado de todas as reclamações aplicando a sequencia de eventos em ordem cronológica.

> - Perguntas orientadoras:

> Como ficaria a consulta para saber o status atual de uma reclamação?

> É possível responder "quanto tempo uma reclamação levou em cada etapa"?

para modelar a relação de eventos, criei uma tabela chamada EVENTO_RECLAMACAO que armazena os eventos de mudança de status das reclamações. Cada evento inclui o ID da reclamação, o novo status, e o

timestamp da mudança.

A modelegagem foi basead nos slides da aula sobre introdução ao Event Sourcing, basicamente adaptei o exemplo dado de pedidos para reclamações.

```
CREATE TABLE eventos_reclamacao (
  id_evento    uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  aggregate_id text NOT NULL,           -- id_da_reclamacao (ex: 'R1')
  aggregate_type text NOT NULL DEFAULT 'Reclamacao',
  versao       int NOT NULL DEFAULT 1,
  ts           timestamptz NOT NULL DEFAULT now(),
  tipo_evento  text NOT NULL,           -- ex: 'ReclamacaoRegistrada',
'StatusAlterado'
  payload      jsonb NOT NULL
);
```

Aqui ao invés de 4 relações (tabelas) como no CRUD, temos apenas 1 tabela que armazena todos os eventos relacionados às reclamações, então cada mudança de status gera um novo registro nessa tabela.

2. Criando eventos que representam mudança no ciclo de vida das reclamações:

```
INSERT INTO eventos_reclamacao (aggregate_id, versao, ts, tipo_evento,
payload) VALUES
('R1', 1, now() - interval '10 minutes', 'ReclamacaoRegistrada',
'{"novo_status":"Registrada","id_empresa":1,"id_cliente":1,"id_produto":1}
'),
('R1', 2, now() - interval '7 minutes',  'StatusAlterado',
'{"novo_status":"Em Análise"}'),
('R1', 3, now() - interval '3 minutes',  'StatusAlterado',
'{"novo_status":"Em Judice"}'),
('R1', 4, now() - interval '1 minute',   'StatusAlterado',
'{"novo_status":"Concluída"}');

INSERT INTO eventos_reclamacao (aggregate_id, versao, ts, tipo_evento,
payload) VALUES
('R2',1, now() - interval '8 minutes', 'ReclamacaoRegistrada',
'{"novo_status":"Registrada","id_empresa":2}'),
('R3',1, now() - interval '6 minutes', 'ReclamacaoRegistrada',
'{"novo_status":"Registrada","id_empresa":3}'),
('R4',1, now() - interval '5 minutes', 'ReclamacaoRegistrada',
'{"novo_status":"Registrada","id_empresa":4}'),
('R5',1, now() - interval '4 minutes', 'ReclamacaoRegistrada',
'{"novo_status":"Registrada","id_empresa":5}');

-- Atualizando status das reclamações para ter estados diferentes
INSERT INTO eventos_reclamacao (aggregate_id, versao, ts, tipo_evento,
payload) VALUES
('R3',2, now() - interval '1 minute', 'StatusAlterado',
'{"novo_status":"Em Análise"}'),
('R4',2, now() - interval '45 seconds', 'StatusAlterado',
```

```
  '{"novo_status":"Em Análise"}'),
  ('R4',3, now() - interval '20 seconds', 'StatusAlterado',
  '{"novo_status":"Em Judice"}'),
  ('R5',2, now() - interval '15 seconds', 'StatusAlterado',
  '{"novo_status":"Em Análise"}');
```

Como ficou a tabela de eventos após inserir os eventos acima:

id_evento	aggregate_id	aggregate_typ	versao	ts	tipo_evento	payload
eb1cf915-4807-43bd-a5f6-dbb840a736fe	R1	Reclamacao	1	2025-11-23 08:52:07.674286+00	ReclamacaoRegistrada	{"id_cliente":1,"id_empresa":1,"id_pr
30366e8b-d814-4fc1-bed8-7c38c70a77c2	R1	Reclamacao	2	2025-11-23 08:55:07.674286+00	StatusAlterado	{"novo_status":"Em Análise"}
44809c86-aaea-40bd-9a17-5a548ab077fa	R1	Reclamacao	3	2025-11-23 08:59:07.674286+00	StatusAlterado	{"novo_status":"Em Judice"}
a6e86f2c-d669-440a-8fc7-dd0e61016812	R1	Reclamacao	4	2025-11-23 09:01:07.674286+00	StatusAlterado	{"novo_status":"Concluída"}
988ebcb4-e24c-408a-8cc7-3f0adee57c09	R2	Reclamacao	1	2025-11-23 08:59:30.415229+00	ReclamacaoRegistrada	{"id_empresa":2,"novo_status":"Regist
61641958-7368-4999-9760-4bb63ef6504a	R3	Reclamacao	1	2025-11-23 09:01:30.415229+00	ReclamacaoRegistrada	{"id_empresa":3,"novo_status":"Regist
2033a2f9-01c8-465a-aade-94006c6d789e	R4	Reclamacao	1	2025-11-23 09:02:30.415229+00	ReclamacaoRegistrada	{"id_empresa":4,"novo_status":"Regist
28e0aa4b-843a-42c9-9e30-0e2bcc2c0992	R5	Reclamacao	1	2025-11-23 09:03:30.415229+00	ReclamacaoRegistrada	{"id_empresa":5,"novo_status":"Regist

3. Reconstruindo o estado de todas as reclamações aplicando a sequência de eventos em ordem cronológica:

@Dúvida: Aqui eu poderia criar uma view que mostra o estado atual de cada reclamação aplicando os eventos em ordem cronológica?

Aqui podemos usar a função de janela LEAD para calcular o tempo que cada reclamação passsou em cada etapa do ciclo de vida.

```
SELECT
  aggregate_id,
  payload->>'novo_status' AS status,
  ts AS started_at,
  LEAD(ts) OVER (PARTITION BY aggregate_id ORDER BY ts) AS ended_at,
  (LEAD(ts) OVER (PARTITION BY aggregate_id ORDER BY ts) - ts) AS duration
FROM eventos_reclamacao
WHERE aggregate_type = 'Reclamacao'
ORDER BY aggregate_id, ts;
```

> **Informação — função LEAD**
>
> A função SQL `LEAD()` permite acessar o valor de uma linha subsequente (por exemplo, a próxima linha) em relação à linha atual dentro do mesmo conjunto de resultados, sem necessidade de auto-join. Isso é especialmente útil para cálculos entre eventos sequenciais, como computar a diferença de tempo entre timestamps.
>
> Exemplo de uso:
>
> ```
> LEAD(ts) OVER (PARTITION BY aggregate_id ORDER BY ts) AS ended_at
> ```

> No exemplo acima, para cada `aggregate_id` é selecionado o timestamp do próximo evento (`ended_at`) ordenado por `ts`. A função complementar `LAG()` lê a enésima linha anterior.

respondendo as perguntas orientadoras:

> Como ficaria a consulta para saber o status atual de uma reclamação?

```
WITH reclamacoes_ordenadas AS (
  SELECT
    e.aggregate_id,
    e.payload->>'novo_status' AS status_atual,
    e.ts,
    ROW_NUMBER() OVER (PARTITION BY e.aggregate_id ORDER BY e.ts DESC) AS
rn
  FROM eventos_reclamacao e
  WHERE e.aggregate_type = 'Reclamacao'
)
SELECT aggregate_id, status_atual, ts
FROM reclamacoes_ordenadas
WHERE rn = 1;
```

aggregate_id	status_atual	ts
R1	Concluída	2025-11-23 09:01:07.674286+00
R2	Registrada	2025-11-23 08:59:30.415229+00
R3	Em Análise	2025-11-23 18:58:08.941179+00
R4	Em Judice	2025-11-23 18:58:48.941179+00
R5	Em Análise	2025-11-23 18:58:53.941179+00

> É possível responder "quanto tempo uma reclamação levou em cada etapa"?

- Agora as coisas mudaram. Como o event sourcing vai armazenando coda etapa do ciclo de vida - e aqui vale observar que diferentemente das procedures que construí no CRUD, aqui eu apenas insiro eventos na tabela. Assim eu consigo rastrear o tempo que cada reclamação levou em cada etapa do ciclo de vida.

```
CREATE OR REPLACE VIEW duracao_etapas_reclamacao AS
SELECT
```

```
  aggregate_id,
  payload->>'novo_status' AS status,
  ts AS started_at,
  LEAD(ts) OVER (PARTITION BY aggregate_id ORDER BY ts) AS ended_at,
  (LEAD(ts) OVER (PARTITION BY aggregate_id ORDER BY ts) - ts) AS duration
FROM eventos_reclamacao
WHERE aggregate_type = 'Reclamacao'
ORDER BY aggregate_id, ts;
```

- Aqui estão os resultados da view criada acima:

> OBS: No print podemos obersver que algumas reclamações tem um tempo muito distinto da outras, isso porque durante a realização das atividades tive um espaçamento - eu dormi.

aggregate_id	status	started_at	ended_at	duration
R1	Registrada	2025-11-23 08:52:07.674286+00	2025-11-23 08:55:07.674286+00	00:03:00
R1	Em Análise	2025-11-23 08:55:07.674286+00	2025-11-23 08:59:07.674286+00	00:04:00
R1	Em Judice	2025-11-23 08:59:07.674286+00	2025-11-23 09:01:07.674286+00	00:02:00
R1	Concluída	2025-11-23 09:01:07.674286+00	NULL	NULL
R2	Registrada	2025-11-23 08:59:30.415229+00	NULL	NULL
R3	Registrada	2025-11-23 09:01:30.415229+00	2025-11-23 18:58:08.941179+00	09:56:38.52595
R3	Em Análise	2025-11-23 18:58:08.941179+00	NULL	NULL
R4	Registrada	2025-11-23 09:02:30.415229+00	2025-11-23 18:58:23.941179+00	09:55:53.52595
R4	Em Análise	2025-11-23 18:58:23.941179+00	2025-11-23 18:58:48.941179+00	00:00:25
R4	Em Judice	2025-11-23 18:58:48.941179+00	NULL	NULL
R5	Registrada	2025-11-23 09:03:30.415229+00	2025-11-23 18:58:53.941179+00	09:55:23.52595
R5	Em Análise	2025-11-23 18:58:53.941179+00	NULL	NULL

Parte 3 - Coparativo CRUD vs Event Sourcing

> Faça um pequeno experimento e compare as duas abordagens quanto ao crescimento do volume de dados ao longo do tempo e desempenho em atualizações.

O que já sabemos da anterior (3):

INSERT VS INSERT + UPDATE:

- Já vimos que o `UPDATE` no geral é uma operação mais custosa que o `INSERT` - atividade 3 - porque o banco precisa primeiro localizar o registro a ser atualizado, que involve leitura de indices, e depois modificar o registro existente, escalando isso para um grande volume de dados com alta concorrencia, o desempenho pode ser impactado devido a bloqueios e contenção de recursos. No caso do `INSERT`, o banco simplesmente adiciona um novo registro, o que é geralmente mais rápido e eficiente a depender da indexação e estrutura da tabela.

desempenho em atualizações

- No crud da nossa situação temos 4 estados, 4 transições possíveis sendo 1 `INSERT` inicial e 3 `UPDATE` para cada reclamação, então supondo uma distribuição uniforme de transições temos 1/4 de `INSERT` e 3/4 de `UPDATE` - ou seja, 75% das operações são teoricamente mais custosas vs 100% de `INSERT` no event sourcing.
- Considerando que o volume de dados cresça linearmente em ambos os casos, o event sourcing pode ter uma vantagem de desempenho em cenários com alta frequência de atualizações, pois evita a sobrecarga associada às operações de atualização devido ao overhead de leitura prévia do update - principalmente considerando uma indexação inadequada.

crescimento do volume de dados ao longo do tempo

- O CRUD armazena apenas o estado atual dos dados, resultando em um crescimento de dados mais controlado e previsível. Cada registro é atualizado no local mantendo a relação 1 - 1 com o número de reclamações abertas.
- O Event Sourcing, por outro lado, armazena cada mudança de estado como um evento separado. Isso leva ao crescimento rápido do volume tendo. Tendo em vista nosso escopo fechado de estados possíveis a 4 estados por reclamação, a memória consumida pode a chegar a ser 4 vezes maior que o CRUD para o mesmo número de reclamações.

Como experimento podemos usar o proprio codigo supracitado ja durante a tarefa.

Eu tentei elaborar algo mais complexo para medir o tempo mas devido as limitacoes de ambiente e tempo, acabei optando por essa abordagem qualitativa. O supabase nao permite comandos de monitoramento mais avançados como wall clock time em trechos dentro de procedures, por exemplo.

Parte 4 – Plot Twist

Inicialmente, a agência só precisava guardar o estado final da reclamação. Contudo, após a promulgação de uma nova lei regulatória, são exigidos relatórios completos sobre as reclamações, contendo:

(i) Histórico de todos os status de cada reclamação

(ii) Tempo médio gasto em cada etapa do processo.

Pergunta final:

> Como essa mudança de requisito impacta a escolha entre CRUD e Event Sourcing?

- O CRUD tradicional não é adequado para atender aos novos requisitos por armazenar apenas o estado (visao foto) atual das reclamações, tornando impossível rastrear o histórico completo de status e calcular o tempo médio gasto em cada etapa do processo, por isso o event sourcing cai como uma luva.

> Qual abordagem traz menos esforço de adaptação? Por quê?

- A abordagem de Event Sourcing traz menos esforço de adaptação para atender aos novos requisitos regulatórios. Isso ocorre porque o Event Sourcing já armazena todas as mudanças de estado como

eventos imutáveis, facilitando a geração de relatórios completos sobre o histórico de cada reclamação e o cálculo do tempo médio gasto em cada etapa do processo

> Considerando a arquitetura que você identificou como a de maior esforço de adaptação, quais modificações você precisaria realizar para adequá-la à exigência regulatória? Você pode combinar sua resposta com trechos de código (SQL) para ilustrar.

- Seria necessário implementar um mecanismo para rastrear o histórico completo de status de cada reclamação. Isso poderia ser feito criando uma tabela adicional para armazenar os eventos de mudança de status OU criando novas colunas com status de alteracao, mas isso poderia tornar a modelagem teoricamente incorreta por armazenar mais coisas do que deveria em apenas uma relacao. Entao a abordagem de uma tabela a mais para armazenar as mudancas seria talvez a melhor abordagem