ChatGPT

Seleção em Massa com Paginação: solução única com Java 21, Spring Boot 3.4.5 e Aurora PostgreSQL

Introdução

O cenário em questão utiliza uma **tabela paginada** que já está implementada no front-end e **não pode ser alterada** pela equipe de back-end. Essa tabela permite ao usuário:

- ordenar as colunas de forma crescente ou decrescente;
- clicar nos checkboxes para selecionar ou desmarcar individualmente cada linha;
- usar um **checkbox mestre** no cabeçalho para marcar ou desmarcar todos os itens da página corrente;
- usar um botão "**Selecionar tudo**" para marcar todos os itens que atendem ao filtro, independentemente da página;
- usar um botão "**Desmarcar tudo**" para limpar a seleção de todas as páginas;
- navegar entre páginas, avançar para uma tela de confirmação e voltar mantendo o estado das seleções.

Como o front-end é fixo, qualquer lógica de marcação e persistência do estado deve ser implementada no **back-end**. A solução precisa lidar com quatro tipos de ações enviados pelo front-end: marcar/desmarcar itens individualmente, marcar todos os itens de uma página, marcar todos os itens de todas as páginas e desmarcar todos os itens de todas as páginas. Também deve manter o estado de seleção quando o usuário avança para outra tela e retorna. A dúvida central é como representar esse estado no servidor: gravar a cada clique? Persistir no navegador? Ou manter um rascunho controlado pelo back-end? A solução escolhida deve respeitar as melhores práticas de UX (permitir seleção em várias páginas, exibir contagem de itens e confirmar antes de executar) [1] e, ao mesmo tempo, ser eficiente para o banco de dados.

Diretrizes de UX e motivação

Um bom design de paginação permite que o usuário selecione itens em diferentes páginas e tenha clareza sobre quantos serão afetados. O artigo de Andrew Coyle enfatiza que uma seleção em massa deve:

- permitir marcar itens em várias páginas;
- exibir uma contagem global (por exemplo, "78 itens selecionados em 3 páginas");
- oferecer uma opção de selecionar todos os resultados que atendem ao filtro;
- exibir um diálogo de confirmação antes de executar a ação [1].

Além disso, para conjuntos de dados grandes e dinâmicos é mais eficiente delegar paginação, filtros e lógica de negócio ao servidor. Uma publicação sobre melhores práticas de paginação ressalta que para **grandes conjuntos de dados**, **dados dinâmicos** ou **consultas complexas**, filtrar e paginar no back-end evita enviar todos os registros para o front e aproveita melhor as capacidades do banco. Essas

premissas fundamentam a decisão de não gravar cada clique no banco principal e manter o estado de seleção no servidor até que o usuário confirme a ação.

Visão geral da interface

Para contextualizar a solução de back-end, a figura abaixo ilustra de forma esquemática a tela de tabela paginada que o usuário utiliza. Cada linha possui uma caixa de seleção e, no cabeçalho da tabela, há uma **caixa de seleção mestre** que seleciona ou desmarca todos os itens da página atual. À direita dessa checkbox existem dois botões: um para selecionar todos os itens de todas as páginas e outro para desmarcar todos os itens de todas as páginas. O usuário pode ordenar as colunas e navegar entre páginas, e o estado das seleções deve permanecer consistente.

Solução recomendada

A proposta é centralizar o estado de seleção em uma **sessão de seleção** no servidor. Quando o usuário inicia uma seleção, o front-end envia os filtros desejados e recebe um `selectionId`. Essa sessão possui:

- **Filtro**: parâmetros usados para buscar os itens (ex.: situação, faixa de vencimento);
- **Modo**: indica se a seleção começa vazia (`NONE`) ou se considera todos os itens filtrados (`ALL`);
- **Listas de inclusão e exclusão**: IDs explicitamente marcados (quando o modo é `NONE`) ou desmarcados (quando o modo é `ALL`);
- **Contagem**: número de itens que serão afetados de acordo com o filtro e as exclusões/inclusões.

O front-end não mantém duas listas; ele apenas envia ao servidor as IDs visíveis quando o usuário marca ou desmarca uma página. A cada interação significativa (mudar de página, marcar/desmarcar um grupo ou acionar “selecionar tudo”), o front envia um `PATCH` para atualizar a sessão. O servidor atualiza o modo ou insere/retira IDs das listas e devolve a contagem atual. Apenas quando o usuário confirma a ação é que o back-end aplica as alterações em lote no banco de dados.

Armazenamento e modelo de dados

Para persistir a sessão de seleção no servidor é utilizada uma tabela simples em Aurora PostgreSQL chamada `selection`. Essa tabela armazena os campos principais e as listas de IDs em colunas do tipo `jsonb`, o que dispensa criar tabelas auxiliares. Uma possível estrutura em SQL é:

```
create table selection (
  id uuid primary key,
  user_id uuid not null,
  mode varchar(10) not null,              -- 'ALL' ou 'NONE'
  filter_json jsonb not null,            -- filtros aplicados na consulta
  include_ids jsonb default '[]'::jsonb, -- lista de inclusões quando
mode='NONE'
  exclude_ids jsonb default '[]'::jsonb, -- lista de exclusões quando
mode='ALL'
  version integer not null default 0,    -- controle de concorrência otimista
  created_at timestamp with time zone not null default now(),
  expires_at timestamp with time zone     -- TTL para limpeza automática
);

-- Índice para acelerar buscas pela seleção
create index on selection (user_id);
```

As listas `include_ids` e `exclude_ids` armazenam arrays JSON de IDs de pagamentos. A escolha por PostgreSQL se justifica pela flexibilidade do tipo `jsonb` e pelo suporte a operações em lote e controle de concorrência avançado; Aurora PostgreSQL oferece compatibilidade total com essas funções.

API REST proposta

Os endpoints são reduzidos para favorecer clareza e minimizar chamadas desnecessárias. Todos utilizam JSON como formato de troca.

Método & caminho	Descrição	Corpo da requisição
`POST /selections`	Cria uma nova sessão de seleção. Retorna `selectionId` e contagem inicial.	`{ "filter": { … }, "mode": "NONE" }`
`PATCH /selections/{id}`	Atualiza a sessão: altera o modo (`ALL` / `NONE`), marca ou desmarca individualmente (através de listas `includeIds` e `excludeIds`), marca todos (`mode="ALL"`) ou desmarca todos (`mode="NONE"`).	`{ "mode": "ALL" }`, `{ "mode": "NONE" }` ou `{ "includeIds": [1,2,3], "excludeIds": [4] }`
`POST /selections/{id}/apply`	Confirma a ação em lote (por exemplo, pagar ou alterar status).	`{ "action": "pay" }`

Fluxo de uso

1. **Criação**: O front-end chama `POST /selections` com os filtros (por exemplo, boletos com situação “A PAGAR” e vencimento até determinada data) e informa o modo inicial (`NONE` para começar sem nenhuma seleção ou `ALL` para considerar todos os filtrados). O servidor salva a sessão, calcula a contagem de itens do filtro e devolve o `selectionId` e a contagem.

2. **Marcar ou desmarcar**: Existem duas operações distintas que o front-end pode enviar:

3. **Selecionar página**: quando o usuário marca o **checkbox mestre** no cabeçalho, o front-end coleta as IDs de todos os registros visíveis (por exemplo, 10 itens) e chama `PATCH /selections/{id}`. Se a sessão estiver em modo `NONE`, essas IDs são adicionadas a `includeIds`; se estiver em modo `ALL`, essas IDs são removidas de `excludeIds` (pois em modo `ALL` supõe-se que todos estão marcados, e desmarcar uma página significa excluí-los da seleção global).

4. **Selecionar ou desmarcar individualmente**: ao marcar ou desmarcar um único checkbox, o front envia a ID correspondente no array apropriado (`includeIds` ou `excludeIds`) seguindo a mesma lógica acima.

O servidor atualiza o JSON correspondente, incrementa a versão e calcula a nova contagem.

- **Selecionar tudo**: quando o usuário clica em “Selecionar tudo”, o front envia `{ "mode": "ALL" }`. O servidor altera o modo para `ALL`, limpa a lista de inclusões e passa a usar `excludeIds` para registrar itens explicitamente desmarcados.

- **Desmarcar tudo**: se o usuário clica em “Desmarcar tudo”, o front envia `{ "mode": "NONE" }`. O servidor altera o modo para `NONE` e limpa as listas de inclusão e exclusão, retornando ao estado inicial onde nenhum item está selecionado.

- **Confirmação**: Quando o usuário clica em “Pagar” ou outra ação, o front chama `POST /selections/{id}/apply`. O serviço lê o filtro e as listas de exclusões/inclusões, gera um `UPDATE` em lote no banco de pagamentos, e só então remove a sessão. Para grandes volumes, essa aplicação pode ser delegada a um worker assíncrono que processa em blocos.

Implementação em Java com Spring Boot 3.4.5

A seguir um exemplo simplificado de como implementar essa solução usando Java 21 e Spring Boot 3.4.5 com Aurora PostgreSQL. Os pacotes e importações foram omitidos para foco na lógica.

Entidades e enumerações

```
@Entity
public class Selection {
    @Id
    private UUID id;
    private UUID userId;
    @Enumerated(EnumType.STRING)
```

```
    private Mode mode;              // ALL ou NONE
    @Column(columnDefinition = "jsonb")
    private String filterJson;      // JSON serializado dos filtros
    @Column(columnDefinition = "jsonb")
    private String includeIds;      // JSON array de IDs incluídos
    @Column(columnDefinition = "jsonb")
    private String excludeIds;      // JSON array de IDs excluídos
    private Integer version;
    private Instant createdAt;
    private Instant expiresAt;
    // getters/setters omitidos
}

public enum Mode { ALL, NONE }

public enum Action { PAY, CANCEL }
```

DTOs para a API

```
// Requisição de criação
public record CreateSelectionRequest(
    Map<String, Object> filter,
    Mode mode
) {}

// Requisição de atualização de seleção
public record UpdateSelectionRequest(
    Mode mode,
    List<Long> includeIds,
    List<Long> excludeIds
) {}

// Requisição de aplicação da ação em lote
public record ApplySelectionRequest(
    Action action
) {}

// Resposta com a contagem de itens
public record SelectionResponse(
    UUID selectionId,
    long selectedCount
) {}
```

Repositório JPA

```
public interface SelectionRepository extends JpaRepository<Selection, UUID> {
    Optional<Selection> findByIdAndUserId(UUID id, UUID userId);
}
```

Serviço de seleção

```
@Service
public class SelectionService {
    private final SelectionRepository repository;
    private final PaymentRepository paymentRepository; // hipotético

    public SelectionResponse create(CreateSelectionRequest req, UUID userId)
{
        Selection sel = new Selection();
        sel.setId(UUID.randomUUID());
        sel.setUserId(userId);
        sel.setMode(req.mode());
        sel.setFilterJson(toJson(req.filter()));
        sel.setIncludeIds("[]");
        sel.setExcludeIds("[]");
        sel.setVersion(0);
        sel.setCreatedAt(Instant.now());
        sel.setExpiresAt(Instant.now().plus(Duration.ofHours(4)));
        repository.save(sel);
        long count = countByFilter(req.filter());
        return new SelectionResponse(sel.getId(), count);
    }

    @Transactional
    public SelectionResponse update(UUID id, UUID userId,
UpdateSelectionRequest req) {
        Selection sel = repository.findByIdAndUserId(id, userId)
            .orElseThrow(() -> new NotFoundException("Selection not found"));
        // mudança de modo (pode ser enviado "ALL" para selecionar todos ou
"NONE" para desmarcar todos)
        if (req.mode() != null) {
            sel.setMode(req.mode());
            sel.setIncludeIds("[]");
            sel.setExcludeIds("[]");
        }
        // inclui IDs se estiver em modo NONE (selecionar página ou registros
individuais)
        if (req.includeIds() != null && sel.getMode() == Mode.NONE) {
            sel.setIncludeIds(mergeJsonArray(sel.getIncludeIds(),
req.includeIds()));
        }
        // exclui IDs se estiver em modo ALL (desmarcar página ou registros
individuais)
        if (req.excludeIds() != null && sel.getMode() == Mode.ALL) {
            sel.setExcludeIds(mergeJsonArray(sel.getExcludeIds(),
req.excludeIds()));
        }
        sel.setVersion(sel.getVersion() + 1);
        repository.save(sel);
        long count = recalcCount(sel);
```

```
        return new SelectionResponse(sel.getId(), count);
    }

    // Aplica a ação em lote
    @Transactional
    public void apply(UUID id, UUID userId, ApplySelectionRequest req) {
        Selection sel = repository.findByIdAndUserId(id, userId)
            .orElseThrow(() -> new NotFoundException("Selection not found"));
        if (req.action() == Action.PAY) {
            applyPayments(sel);
        }
        repository.delete(sel);
    }

    private void applyPayments(Selection sel) {
        Map<String, Object> filter = fromJson(sel.getFilterJson());
        List<Long> excluded = fromJsonArray(sel.getExcludeIds());
        List<Long> included = fromJsonArray(sel.getIncludeIds());
        if (sel.getMode() == Mode.ALL) {
            // Atualiza todos que se enquadram no filtro menos os excluídos
            paymentRepository.bulkUpdateByFilter(filter, excluded);
        } else {
            // Atualiza apenas os incluídos
            paymentRepository.bulkUpdateByIds(included);
        }
    }

    // Métodos utilitários (toJson, mergeJsonArray, countByFilter,
recalcCount, etc.) omitidos
}
```

Repositório de pagamentos e atualização em lote

O `PaymentRepository` deve fornecer métodos específicos para atualizar os pagamentos em lote com base nos filtros e exclusões. Em Spring Data JPA é possível definir consultas nativas para maior eficiência:

```
public interface PaymentRepository extends JpaRepository<Payment, Long> {

    @Modifying
    @Transactional
    @Query(value = """
        UPDATE payment
           SET status = 'PAID', updated_at = NOW()
         WHERE status = 'A_PAGAR'
           AND (vencimento <= :#{#filter['vencimentoAte']})
           AND NOT (id = ANY(:excluded))
        """, nativeQuery = true)
    int bulkUpdateByFilter(@Param("filter") Map<String, Object> filter,
                           @Param("excluded") List<Long> excluded);
```

```
    @Modifying
    @Transactional
    @Query(value = """
        UPDATE payment
           SET status = 'PAID', updated_at = NOW()
         WHERE id IN (:ids)
           AND status = 'A_PAGAR'
        """, nativeQuery = true)
    int bulkUpdateByIds(@Param("ids") List<Long> ids);
}
```

Para volumes muito grandes é possível segmentar a atualização em blocos, usando `LIMIT` e `OFFSET` ou `SELECT ... FOR UPDATE SKIP LOCKED` em consultas temporárias, garantindo que cada worker atualize um subconjunto sem competir por locks. Aurora PostgreSQL oferece suporte a esse padrão, evitando bloqueios ao processar lotes consecutivos.

Controlador REST

```
@RestController
@RequestMapping("/selections")
public class SelectionController {
    private final SelectionService service;
    private final UserService userService;

    @PostMapping
    public SelectionResponse create(@RequestBody CreateSelectionRequest req)
{
        UUID userId = userService.getCurrentUserId();
        return service.create(req, userId);
    }

    @PatchMapping("/{id}")
    public SelectionResponse update(@PathVariable UUID id,
                                    @RequestBody UpdateSelectionRequest req)
{
        UUID userId = userService.getCurrentUserId();
        return service.update(id, userId, req);
    }

    @PostMapping("/{id}/apply")
    public void apply(@PathVariable UUID id,
                      @RequestBody ApplySelectionRequest req) {
        UUID userId = userService.getCurrentUserId();
        service.apply(id, userId, req);
    }
}
```

Considerações finais

Ao centralizar o estado da seleção em uma única tabela `selection` e expor apenas três endpoints, a solução cumpre os requisitos de UX de permitir seleção em várias páginas, exibir contagem de itens e confirmar ações em massa [1]. Ela cobre inclusive o caso de **desmarcar todos os itens**, bastando enviar `mode: "NONE"` para a API e limpar as listas internas. Como o front-end não pode ser modificado, ele continua controlando a interface e apenas **envia as IDs dos registros selecionados ou desmarcados** ou o modo desejado. O back-end mantém toda a lógica de negócio, calcula a contagem em tempo real e persiste o estado de seleção de forma que o usuário possa avançar para outras telas e voltar sem perder as escolhas. A utilização de Aurora PostgreSQL e colunas `jsonb` simplifica o modelo de dados, reduz a necessidade de tabelas auxiliares e permite atualizações em lote eficientes. Esse padrão oferece uma implementação robusta e escalável para cenários em que as seleções atravessam páginas paginadas, sem exigir mudanças na camada de apresentação.

---

[1] Design Better Pagination by Andrew Coyle
https://www.andrewcoyle.com/blog/design-better-pagination