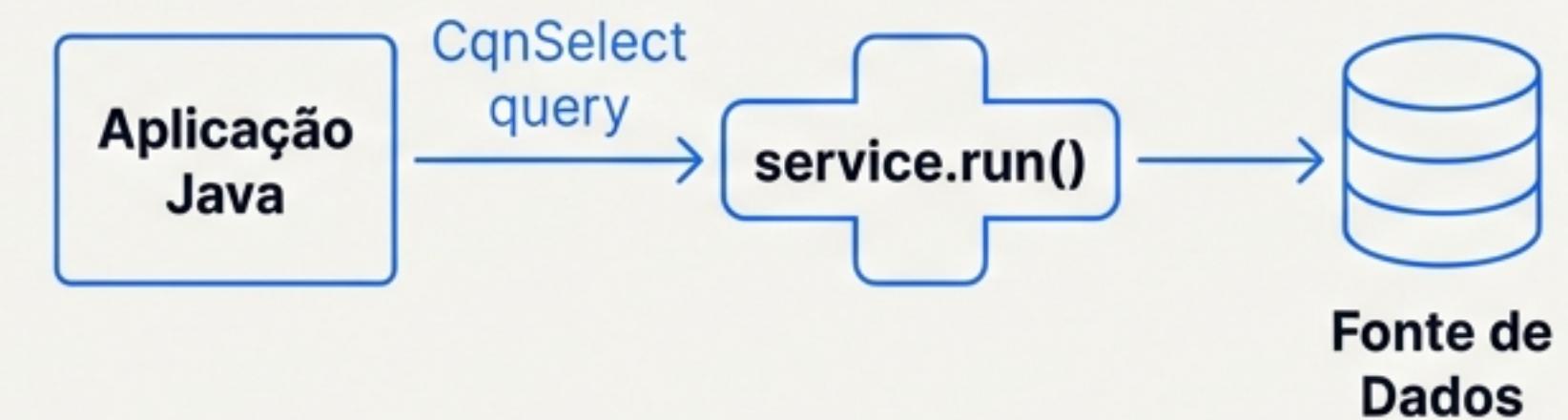


# Dominando a Manipulação de Dados com CAP Java SDK

Uma Jornada do Básico ao Avançado na Execução de Declarações CQL

# A Base de Tudo: Executando Consultas com service.run()

- O método **run** de um CqnService é o ponto de partida para executar qualquer declaração CQL (Consultas, Updates, Deletes).
- ❖ A parametrização é crucial para a segurança e flexibilidade. Apresentaremos duas abordagens principais: parâmetros nomeados e indexados.
- ❖ Demonstra a forma mais simples de buscar dados, preparando o terreno para adicionar complexidade de forma controlada.



```
// Consulta básica sem parâmetros
CqnService service = ...
CqnSelect query = Select.from("bookshop.Books")
    .columns("title", "price");
Result result = service.run(query);
```

# Parametrização em Foco: Nomeados vs. Indexados



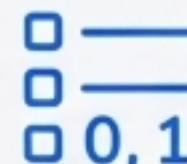
## Parâmetros Nomeados (`Named`)

Utilizam um `Map<String, Object>` para vincular nomes de parâmetros a valores. Ideal para clareza em consultas com múltiplos parâmetros.

```
// Usando param("nome") e um Map
CqnDelete delete = Delete.from("bookshop.Books")
    .where(b -> b.get("ID").eq(param("id1"))
        .or(b.get("ID").eq(param("id2"))));
Map<String, Object> paramValues = new HashMap<>();
paramValues.put("id1", 101);
paramValues.put("id2", 102);
Result result = service.run(delete, paramValues);
```



O mapa de parâmetros deve ser do tipo `Map<String, Object>`. Caso contrário, será interpretado como um único valor de parâmetro posicional, resultando em erro.



## Parâmetros Indexados (`Indexed`)

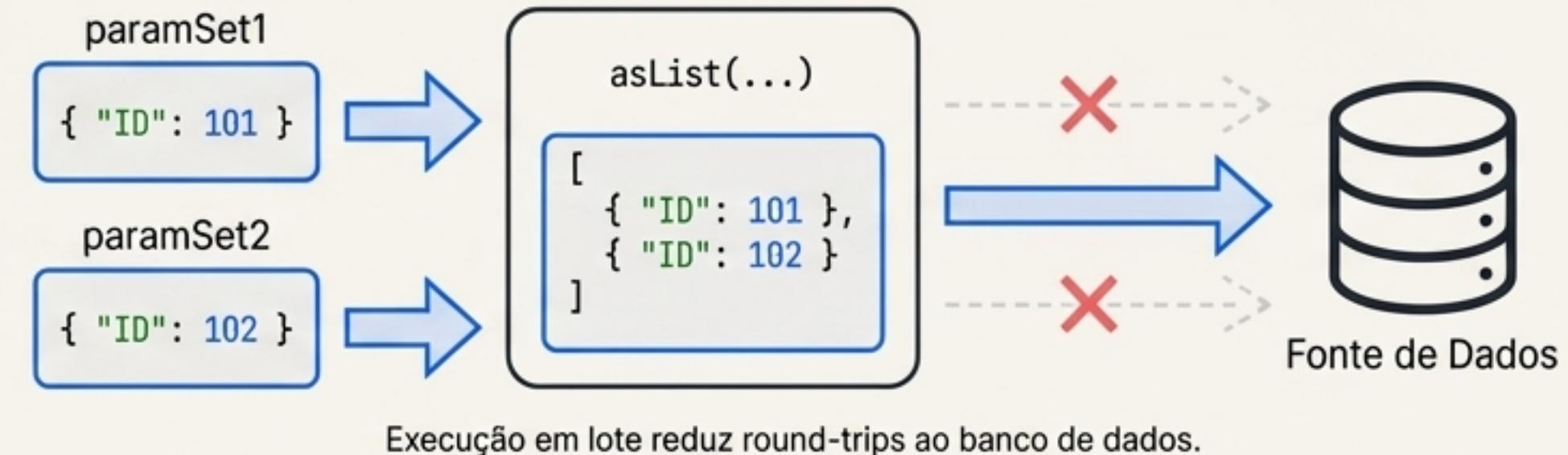
Utilizam a posição (índice) para vincular valores. Mais conciso para consultas simples com poucos parâmetros.

```
// Usando param(i) e varargs
CqnDelete delete = Delete.from("bookshop.Books")
    .where(b -> b.get("ID").in(param(0), param(1)));
Result result = service.run(delete, 101, 102);
```

# Eficiência em Larga Escala: Execução em Lote (`Batch`)

## Conceito

Updates e deletes com parâmetros nomeados podem ser executados em modo `batch` com múltiplos conjuntos de parâmetros, reduzindo a sobrecarga de comunicação com o banco de dados.



```
// Batch delete usando byParams("ID") e uma lista de mapas
CqnDelete delete = Delete.from("bookshop.Books").byParams("ID");

Map<String, Object> paramSet1 = singletonMap("ID", 101);
Map<String, Object> paramSet2 = singletonMap("ID", 102);

Result result = service.run(delete, asList(paramSet1, paramSet2));
```

## Processando o Resultado do Batch

- **result.rowCount()**: Retorna o número total de linhas afetadas.
- **result.rowCount(batchIndex)**: Retorna o número de linhas para um conjunto de parâmetros específico.
- **result.batchCount()**: Retorna o número de lotes executados.



### Pro-Tip

O tamanho máximo do lote para `update` e `delete` pode ser configurado via `cds.sql.max-batch-size` (padrão: 1000).

# O Poder da Modificação: `Update`, `Insert` e `Upsert`

A API CQN permite manipular dados através de declarações `insert`, `update`, `delete` ou `upsert`. Inter Regular.



## Análise do Resultado

- O `Result` de um `update` contém os dados que foram escritos, incluindo valores gerados para campos gerenciados (ex: `modifiedAt`) e chaves estrangeiras.
- `rowCount` indica o número de linhas afetadas. Um `rowCount` de 0 com uma execução bem-sucedida significa que nenhuma linha correspondeu à condição.

## Observação

Para `updates` com expressões (ex: `stock = stock + 1`), o resultado dessas expressões é avaliado no banco de dados e não está contido no `updateResult`.

# Dados Estruturados: Entendendo Operações em Cascata

## Comportamento Padrão

Por padrão, `insert`, `update` e `delete` operam em cascata **apenas sobre `compositions`**.

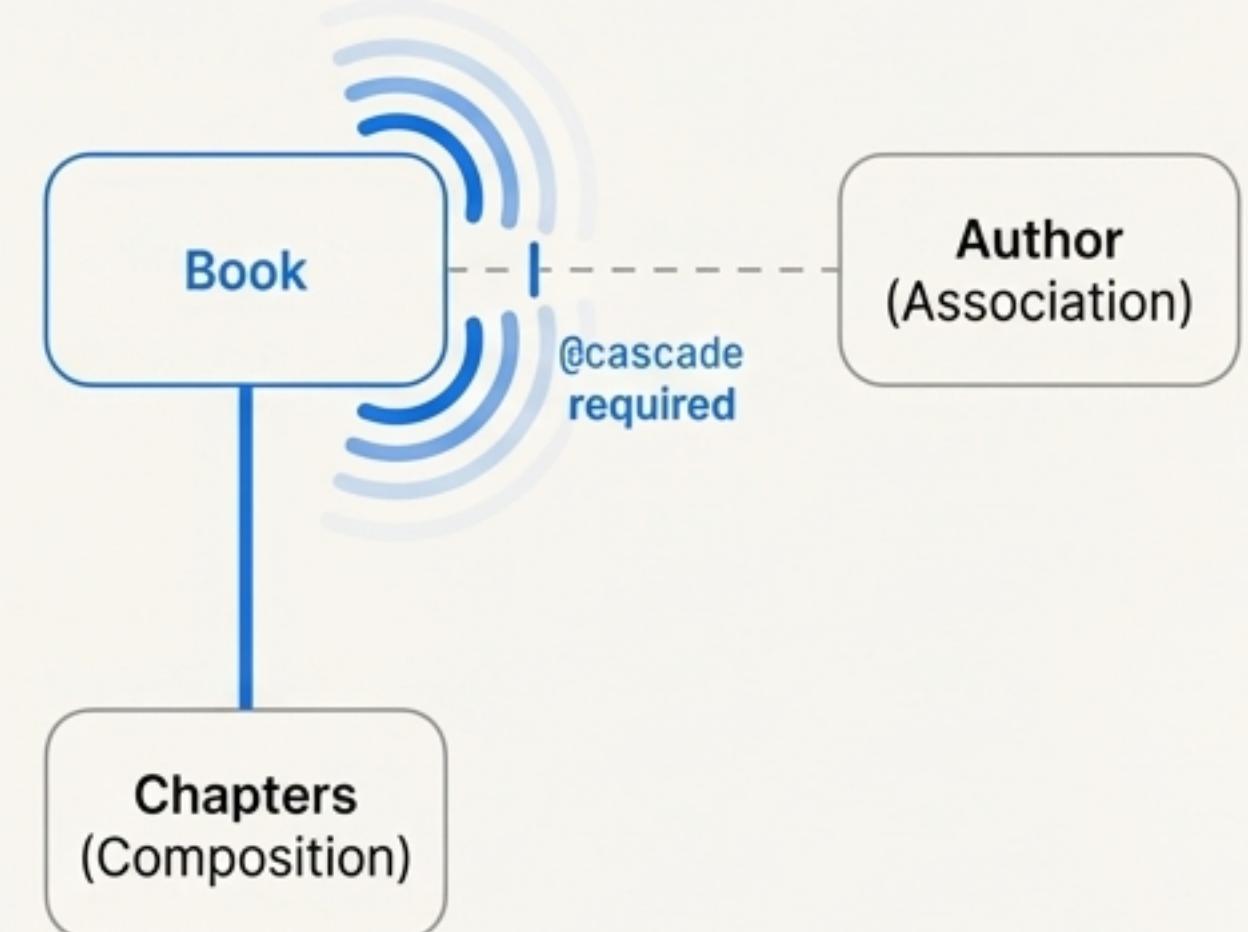
## Ativando para `Associations`

A anotação `@cascade` pode habilitar a cascata para associações.

```
entity Book {  
    key ID : Integer;  
    title : String;  
    @cascade: {insert, update} // Apenas insert e update em cascata  
    author : Association to Author;  
}  
  
entity Author { ... }
```

i A anotação `@cascade` é ignorada para entidades em modo `draft`.

! Pode ser necessário desabilitar constraints de chave estrangeira com `@assert.integrity:false` para evitar conflitos.

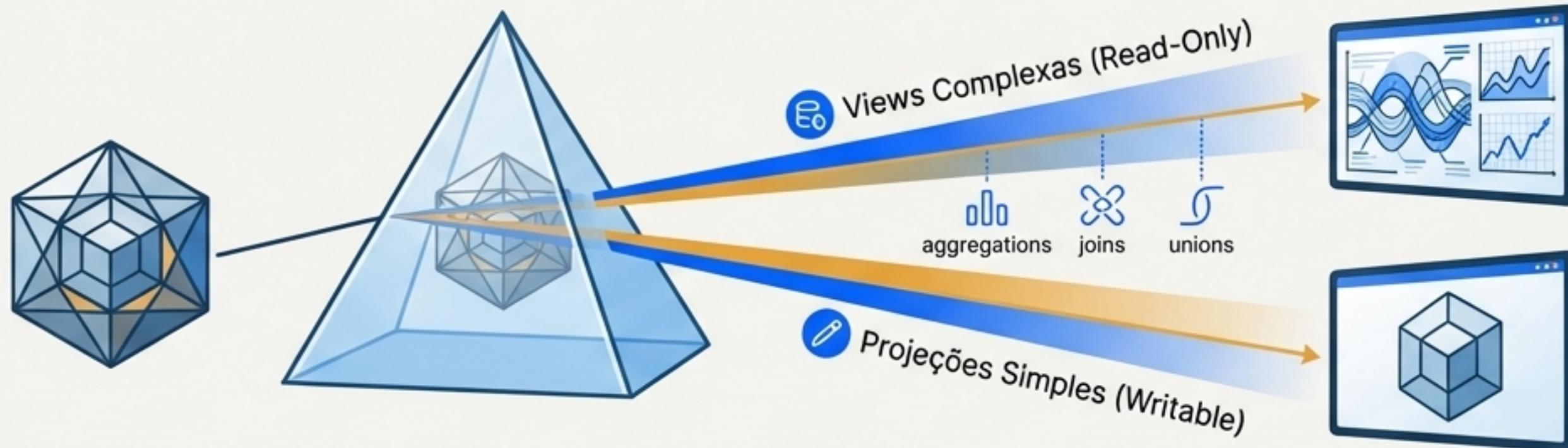


## Evite a Cascata sobre Associações

Operações em cascata sobre associações não são consideradas uma boa prática e devem ser evitadas. Elas podem levar a comportamento inesperado e dificultar a manutenção.

# Abstração e Projeção: Trabalhando com CDS Views

O que são Views? Permitem derivar novas entidades a partir de existentes, renomeando/excluindo elementos ou adicionando elementos virtuais.



## Distinção Crucial

**Views Complexas (Read-Only):** Usam 'aggregations', 'joins', 'unions'. São poderosas para leitura, mas não são graváveis e exigem redeploy do esquema se alteradas.

**Projeções Simples (Writable):** O runtime do CAP Java tenta resolver a view para a entidade de persistência subjacente, reescrevendo a declaração. São a base para operações de escrita.

## Callouts de Boas Práticas



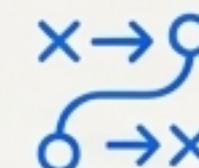
### Anote Views e Elementos Read-Only

Use `@readonly` para indicar explicitamente que uma view ou elemento não é gravável.



### Prefira Views Simples

Aplique o Princípio de Segregação de Interface. Crie múltiplas views simples para casos de uso específicos em vez de uma view complexa para múltiplos cenários.



### Evite Caminhos sobre Associações To-Many

Bloqueia operações de escrita e pode causar problemas de performance.

# Escrevendo e Deletando Através de Views

## Requisitos para uma View ser “Resolvível” (Gravável pelo Runtime):

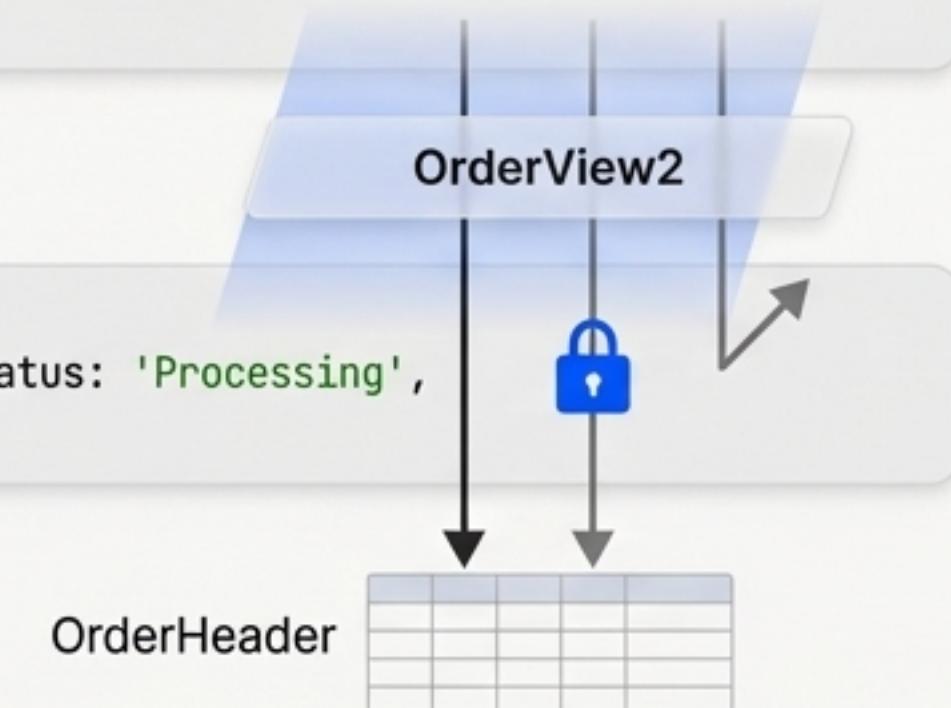
1. É uma projeção simples (sem `aggregations`, `join`, `union`, `where`).
2. Inclui todos os elementos `not null` (chaves, etc.), a menos que tenham valor padrão ou gerado.
3. Não inclui expressões de caminho usando associações `to-many`.

## Exemplo Prático - CDS View

```
// View que projeta uma Order com caminhos  
entity OrderView2 as projection on OrderView1 {  
    key ID,  
    header.status      as headerStatus, // Caminho gravável (composition)  
    header.customer.name as customerName @readonly, // Caminho read-only (association)  
    items              as lineItems,     // Composição gravável  
    toUpper(shipToCountry) as country : String // Ignorado na escrita  
};
```

## SQL-like Update

```
// Exemplo de Update na View  
UPDATE entity OrderView2 { ID: 42, headerStatus: 'Processing',  
    lineItems: [{ID: 1, book:251}] }
```



### Exclusão em Cascata

A exclusão em cascata é aplicada apenas no nível da entidade de persistência. Composições definidas apenas na view são ignoradas.



# Agilidade no Desenvolvimento: Runtime Views

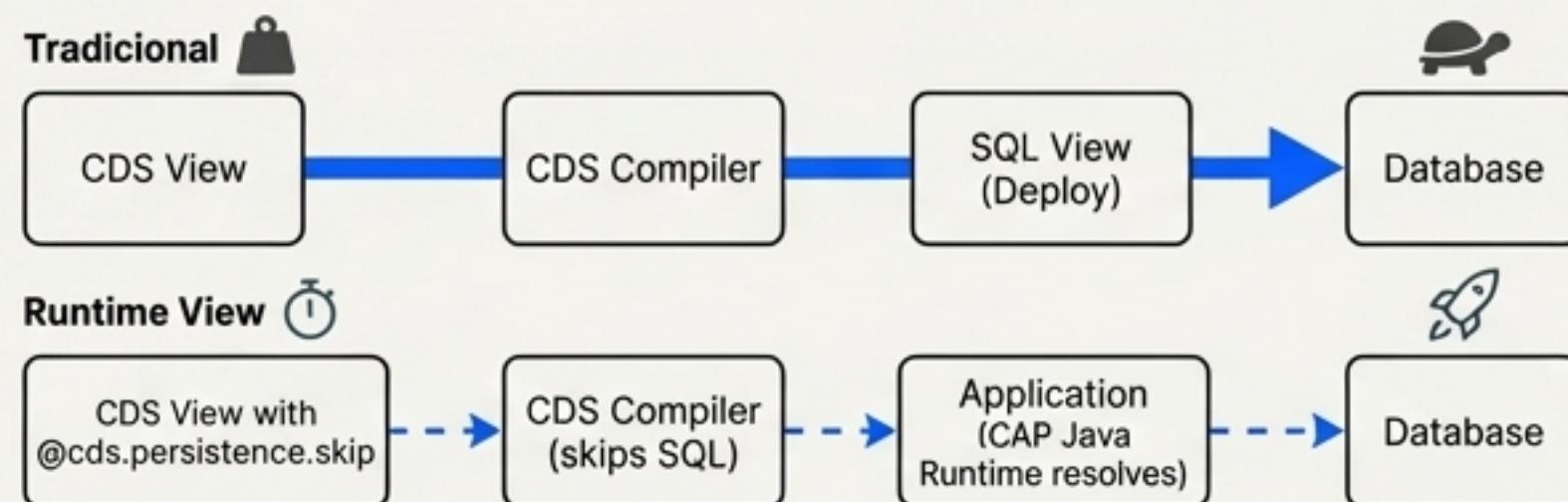
## Conceito e Funcionamento

### O Conceito

Anote views com `@cds.persistence.skip` para que o compilador CDS não gere uma view SQL no banco de dados.

### Como Funciona

Em vez disso, o CAP Java resolve a view em tempo de execução a cada requisição.



## Prós e Contras

### Benefícios

- ✓ Permite adicionar ou atualizar views sem a necessidade de um novo deploy do esquema do banco de dados.
- ✓ É um pré-requisito para extensibilidade leve.

### Restrições

- ⚠ Devem ser projeções simples.
- ⚠ Podem ter uma cláusula 'where' para leitura.
- ⚠ Se o runtime não conseguir resolver a view durante uma operação de escrita, a declaração falha (não há fallback para uma view de banco de dados).

# Modos de Resolução de Runtime Views: `cte` vs. `resolve`

Para operações de leitura, o CAP Java oferece dois modos de resolver as views em tempo de execução.

## Modo `cte` (Padrão)

O runtime traduz a definição da view em uma *Common Table Expression (CTE)* e a envia junto com a consulta para o banco de dados.

```
WITH BOOKSWITHLOWSTOCK_CTE AS (
    SELECT B.ID, B.TITLE, A.NAME AS "AUTHOR"
    FROM BOOKS B LEFT OUTER JOIN AUTHOR A ...
    WHERE B STOCK < 10
)
```

```
SELECT ID, TITLE, AUTHOR AS "author"
FROM BOOKSWITHLOWSTOCK_CTE
WHERE A.NAME = ?
```

## Modo `resolve`

O runtime resolve a definição da view para as entidades de persistência subjacentes e executa a consulta diretamente nas tabelas correspondentes.

```
SELECT B.ID, B.TITLE, A.NAME AS "author"
FROM BOOKS AS B LEFT OUTER JOIN AUTHORS AS A ...
WHERE B STOCK < 10 AND A NAME = ?
```

### Limitação

 O modo `resolve` não suporta associações definidas apenas na view e consultas `draft` complexas.

# Garantindo a Integridade dos Dados: Controle de Concorrência

## O Problema

Em aplicações multiusuário, modificações concorrentes podem levar à perda de atualizações e dados inconsistentes.



## A Solução

O CAP Java oferece mecanismos para proteger seus dados contra essas alterações inesperadas.

## Duas Estratégias Principais

### 1. Bloqueio Otimista (Optimistic Locking)



**Quando usar:** Para detectar modificações concorrentes entre requisições.

**Como funciona:** Utiliza um `ETag` (Entity Tag) que muda a cada atualização da entidade. A operação de escrita falha se o `ETag` esperado não corresponder ao valor atual.

### 2. Bloqueio Pessimistico (Pessimistic Locking)



**Quando usar:** Para garantir que os dados lidos não sejam modificados por outra transação até que a transação atual seja concluída.

**Como funciona:** Utiliza bloqueios (`locks`) no nível do banco de dados.

# Bloqueio Otimista: Detectando Conflitos com ETags

## Implementação para OData

A anotação `@odata.etag` em um elemento (tipicamente um timestamp `modifiedAt`) informa ao adaptador OData para usá-lo na detecção de conflitos via cabeçalhos HTTP `If-Match`.

```
entity Order : cuid {  
    @odata.etag  
    @cds.on.update : $now  
    modifiedAt : Timestamp;  
    ...  
}
```

## Uso Programático

Use `CqnEtagPredicate` para especificar o valor esperado do ETag em uma operação de `update` ou `delete`.

```
CqnUpdate update = Update.entity(ORDER).entry(newData)  
CqnUpdate update = Update.entity(ORDER).entry(newData)  
    .where(o -> o.id().eq(85).and(  
        CQL.eTag(expectedLastModification))); // Predicado  
Result rs = db.execute(update);
```

### O Aplicativo DEVE Verificar o Resultado

! Nenhuma exceção é lançada se a validação do ETag falhar. A execução do `update`/`delete` é bem-sucedida, mas não aplica nenhuma alteração. Verifique se `rs.rowCount() == 0` para detectar um conflito de concorrência e implementar sua lógica de tratamento de erro.

# ETags Avançados: Versões Gerenciadas pelo Runtime (beta)

## Conceito

A Alternativa: Em vez de `@cds.on.update`, use `@cds.java.version` para que o runtime gerencie exclusivamente os valores de versão.

## Vantagens sobre `@cds.on.update`

- Suporta tipos integrais (`Int32`, `Int64`, etc.), que são incrementados automaticamente, além de `Timestamp` e `UUID`.
- Não permite a definição do valor em código customizado, garantindo que o runtime tenha controle total.

## Fluxo de Código Convenient

O valor da versão no objeto de dados passado para um `update` é usado como o valor esperado para a verificação.

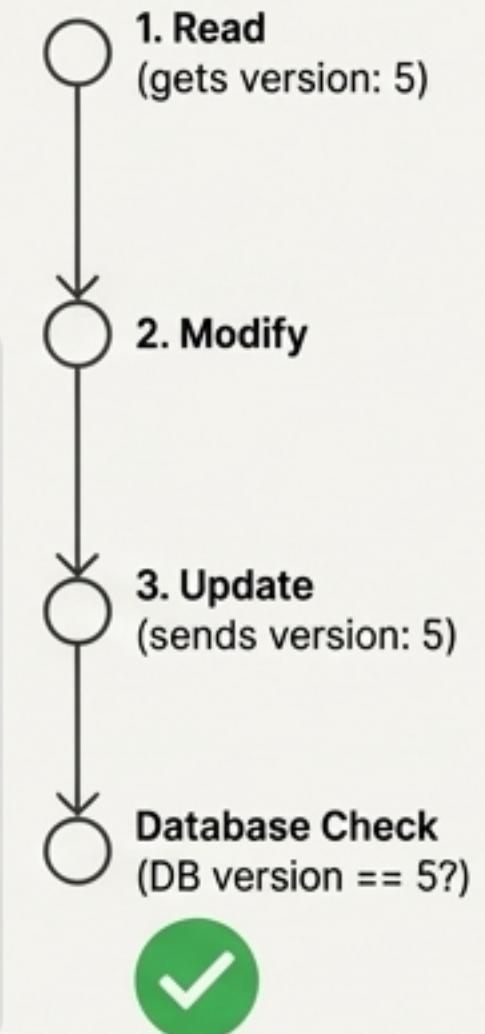
```
// 1. Ler a entidade (incluindo o campo 'version')
Order order = db.run(select).single(Order.class);

// 2. Modificar a entidade
order.setAmount(5800);

// 3. Executar o update com a entidade modificada
CqnUpdate update = Update.entity(ORDER).entry(order);

Result rs = db.execute(update); // O runtime verifica se a 'version' no BD
                                // corresponde à 'version' no objeto 'order'

if (rs.rowCount() == 0) { /* Conflito! */ }
```



# Bloqueio Pessimista: Garantindo Exclusividade com `lock()`

## Fluxo e Parâmetros

### Quando Usar

Para impedir que dados lidos sejam modificados por uma transação concorrente até que a sua seja concluída.

### Fluxo de Trabalho

1. Inicie uma transação.
2. Consulte os dados e aplique um bloqueio usando `lock()`.
3. Processe e modifique os dados (se o bloqueio for exclusivo) dentro da mesma transação.
4. Faça `commit` (ou `rollback`) da transação para liberar o bloqueio.

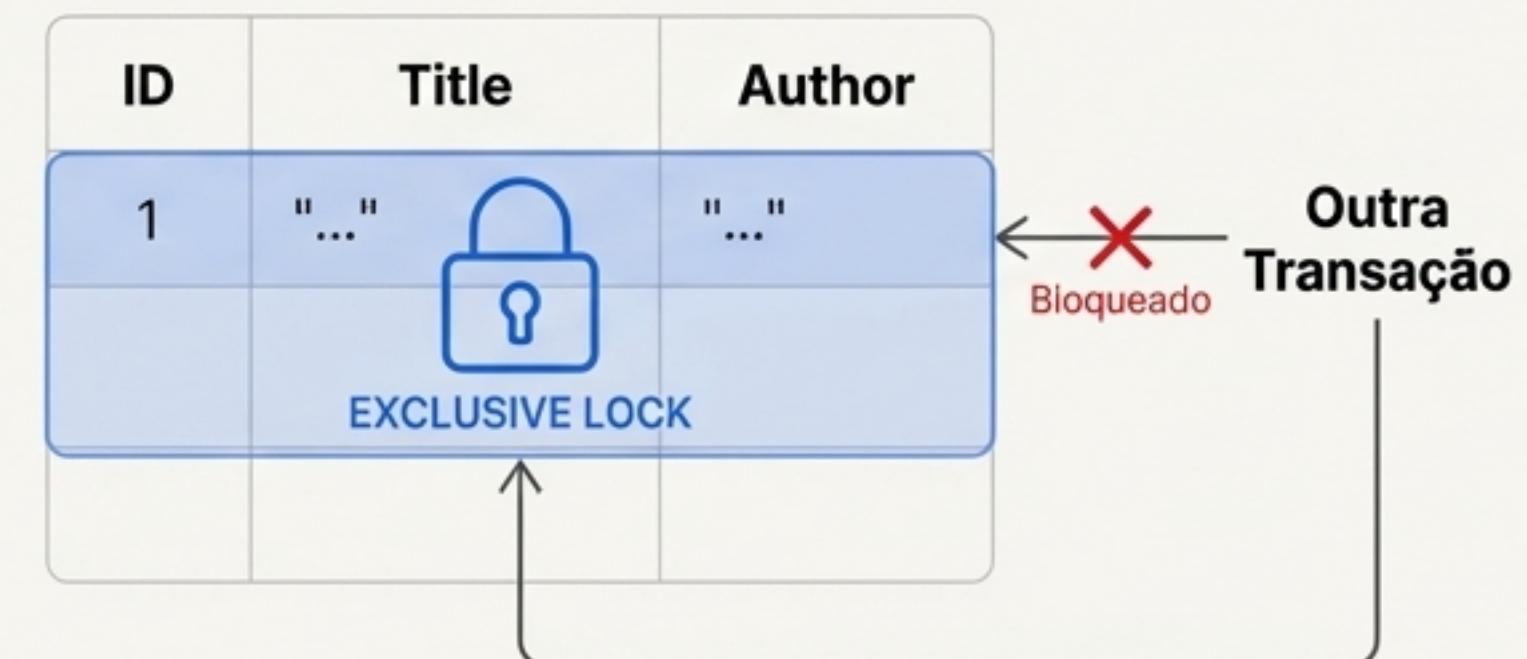
### Parâmetros do `lock()`

- ⌚ `timeout`: Segundos para esperar pelo bloqueio antes de lançar `CdsLockTimeoutException`.
- 🔗 `mode`: `EXCLUSIVE` (bloqueia outras leituras e escritas) ou `SHARED` (permite outras leituras com `SHARED lock`, mas bloqueia escritas).

```
// Dentro de uma transação...
// 1. Seleciona e bloqueia o livro com id 1
service.run(Select.from("bookshop.Books").byId(1).lock());

// ... processamento ...

// 2. Atualiza o livro que foi bloqueado
Map<String, Object> data = ...
service.run(Update.entity("bookshop.Books").data(data).byId(1));
// Fim da transação (commit/rollback)
```



# A Colheita: Processando os Resultados da Consulta

## O Objeto `CdsResult`

É um `Iterable` de `CdsData` (que é um `Map<String, Object>` com métodos de conveniência).

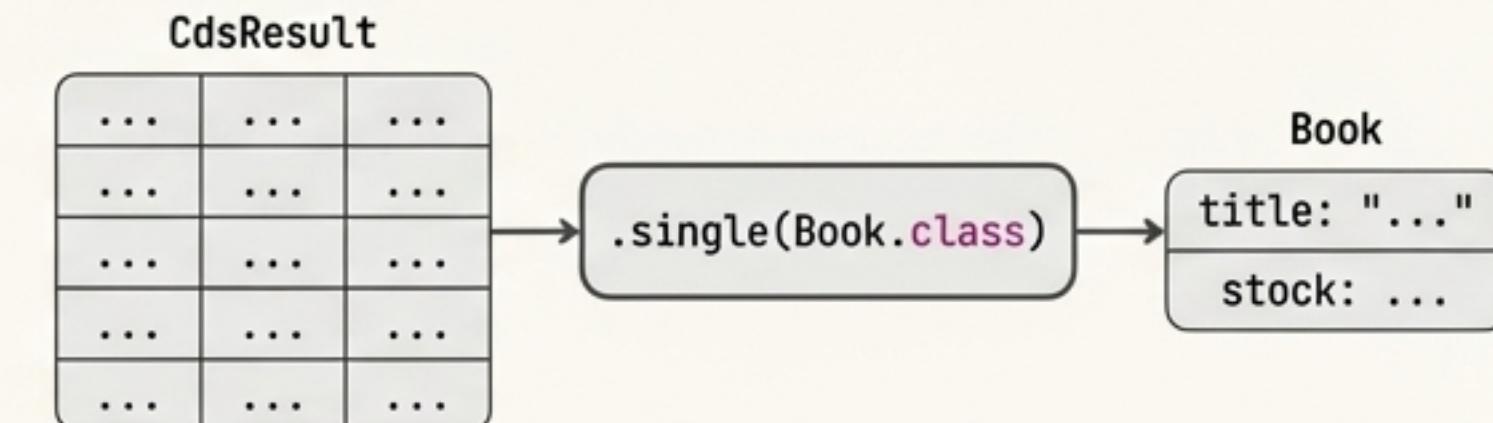
## Métodos de Acesso Comuns

- Iteração: `for (CdsData row : result)`
- Streams: `result.stream().map(...)`
- Resultado Único: `CdsData row = result.single();` (Lança exceção se 0 ou >1 linha)
- Primeiro Resultado: `Optional<CdsData> row = result.first();`

## Processamento Tipado (Type-Safe)

Use interfaces para obter acesso tipado aos dados.

```
// Definindo a interface
interface Book {
    String getTitle();
    Integer getStock();
}
// Usando a interface para obter resultados tipados
CdsResult<?> result = service.run(query);
Book book = result.single(Book.class); // Conversão automática
List<Book> books = result.listOf(Book.class);
```



 **Pro-Tip:** Use o Plugin Maven do CDS para gerar automaticamente as interfaces a partir do seu modelo de dados, simplificando o processo de obtenção de resultados tipados.

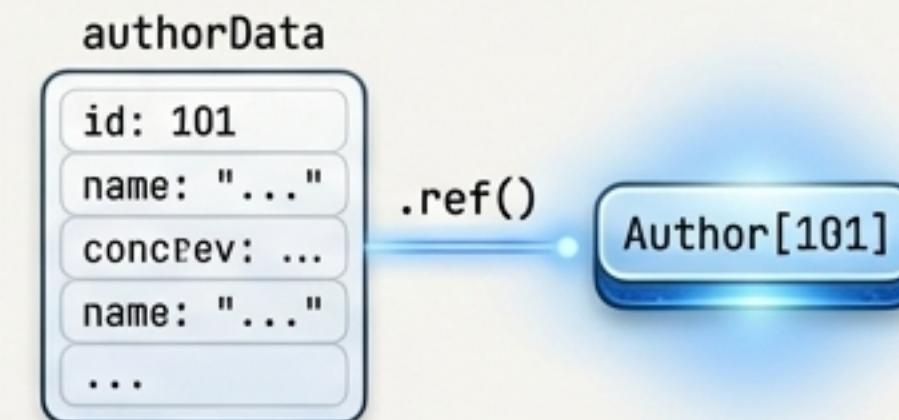
# Além dos Dados: O Poder das Referências de Entidade (`ref()`)

## O Conceito

Para linhas de resultado que contêm todos os valores-chave de uma entidade, o método `ref()` retorna uma referência que aponta para aquela instância específica da entidade (ex: `Author[101]`)

```
// SELECT from Author[101]
CqnSelect query = Select.from(AUTHOR).byId(101);
Author authorData = service.run(query).single(Author.class);
String authorName = authorData.getName();

// Acesso aos dados
Author_ authorRef = authorData.ref(); // Obtém uma referência tipada para Author[101]
```

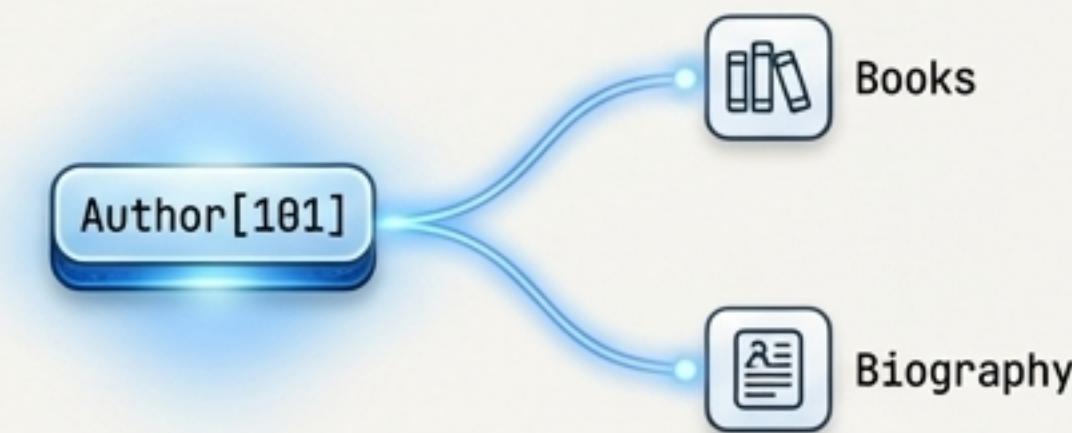


## A Mágica Acontece Aqui

Use a referência para construir novas declarações CQL de forma fluida e intuitiva, navegando pelo modelo de dados.

```
// SELECT from Author[101].books { sum(stock) as stock }
CqnSelect q = Select.from(authorRef.books())
    .columns(b -> func("sum", b.stock()).as("stock"));

// INSERT into Author[101].books
CqnInsert i = Insert.into(authorRef.books()).entry("title", "Ulysses");
```



Dominar as referências de entidade transforma seu código, tornando-o mais expressivo, legível e alinhado com o seu modelo de dados CDS.