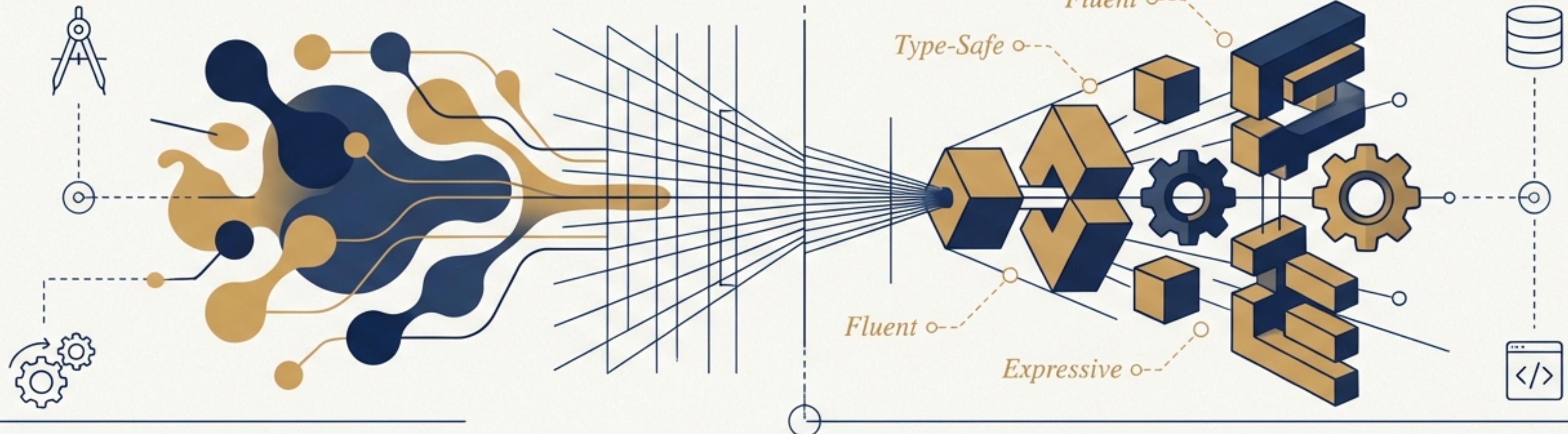


# Construindo Consultas com Maestria: Um Guia Visual para a API Java de Construção de CQL

De consultas simples a manipulações de dados complexas, domine a API fluente para criar código type-safe, legível e eficiente no ecossistema SAP CAP.



Esta apresentação é um guia destilado para desenvolvedores Java que buscam explorar todo o potencial da API de construção de CQL. Vamos percorrer uma jornada, começando pelos fundamentos da API, mergulhando na arte de construir consultas `SELECT` sofisticadas e, por fim, dominando as operações de modificação de dados (`INSERT`, `UPDATE`, `DELETE`). O objetivo é transformar a maneira como você interage com seus dados, trocando strings SQL frágeis por uma API robusta e elegante.



# Dois Estilos, Uma API: Estático vs. Dinâmico

A API oferece duas abordagens para construir statements. A escolha depende do seu caso de uso: lógica de negócio específica ou código genérico.



## Estilo Estático (Type-Safe)

Utiliza constantes e interfaces geradas a partir do modelo CDS. É a abordagem recomendada para lógica de negócio.

- Verificação de nomes de entidades e elementos em tempo de compilação.
- Suporte a code completion na IDE.
- Composição de predicados e expressões com tipagem segura.
- Código mais compacto e legível.

```
// Java CQL (static)
import static bookshop.Bookshop_.BOOKS;

Select.from(BOOKS)
    .columns(b -> b.title())
    .byId(101);
```



## Estilo Dinâmico (Flexível)

Refere-se a entidades e elementos usando strings. Ideal para código genérico que opera sobre entidades desconhecidas em tempo de compilação.

- Flexibilidade para construir consultas dinamicamente em runtime.
- Útil para frameworks e ferramentas genéricas.

```
// Java CQL (dynamic)
Select.from("bookshop.Books")
    .columns("title")
    .ById(101);
```

## CQL Equivalente (para ambos)

```
-- CQL
SELECT from bookshop.Books { title } where ID = 101
```

# A Expressividade das Lambdas: O Coração da API Fluente

A API utiliza intensivamente expressões lambda para compor cláusulas de forma fluida e type-safe. Elas são a chave para criar path expressions e predicados complexos de maneira intuitiva.

```
// Java CQL (static)
Select.from(BOOKS)
    .columns(b -> b.title().as("Book"))
    .where(b -> b.year().lt(2000));
```

- 1 A lambda `b` representa a entidade 'Books'. `b.title().as("Book")` projeta o elemento 'title' com um alias.
- 2 `b -> b.year().lt(2000)` define um predíicado complexo para a cláusula WHERE de forma type-safe.

## Pontos Chave

### 1. Projeções (`columns`)

A lambda define os elementos a serem retornados, permitindo o uso de métodos como `as()` para aliases.

### 2. Filtros (`where`)

A lambda constrói um predíicado que compara elementos da entidade, como `b.year().lt(2000)`.

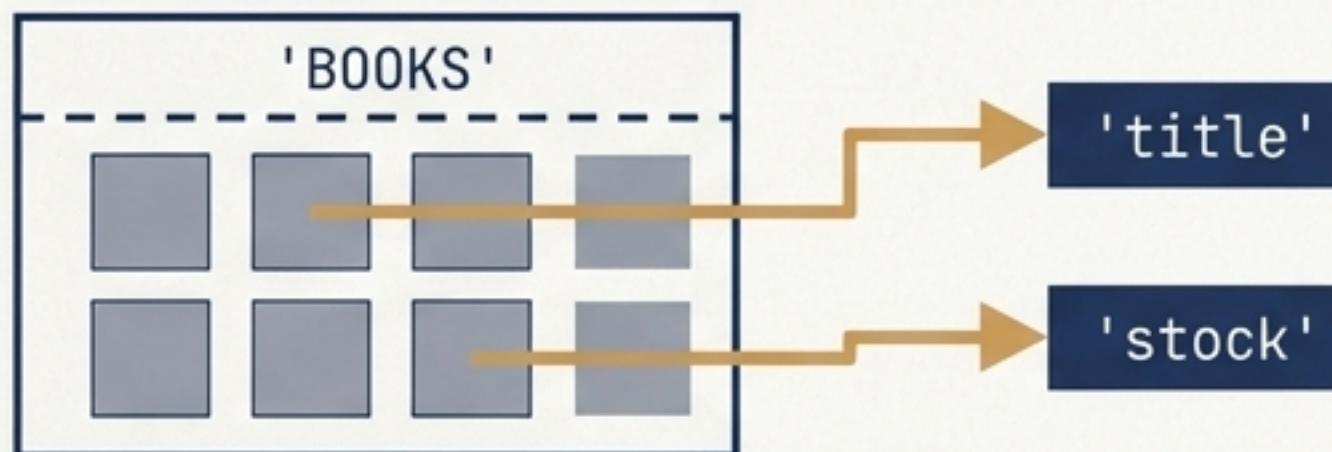
## CQL Equivalente

```
-- CQL
SELECT from bookshop.Books { title as Book } where year < 2000
```

# Modelando Resultados: De Projeções Planas a Path Expressions

Por padrão, um `SELECT` retorna todos os elementos da entidade. Use o método `columns` para definir uma projeção específica, moldando a estrutura do resultado.

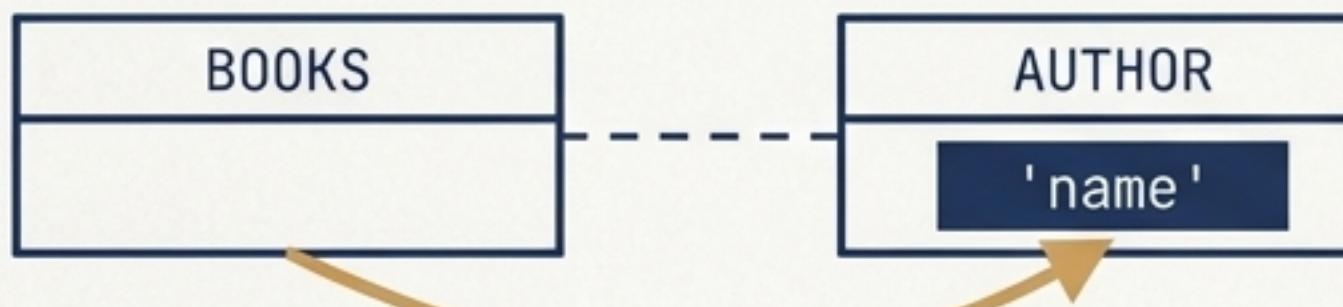
## Cenário 1: Projeção Simples



Selecione apenas os elementos necessários para otimizar a performance.

```
// Estático  
Select.from(BOOKS)  
.columns(b -> b.title(),  
        b -> b.stock());
```

## Cenário 2: Path Expressions para Acessar Associações



Navegue por associações para incluir dados de entidades relacionadas de forma simples. A API converte a path expression em um `LEFT OUTER JOIN` automaticamente em data stores SQL.

### Código (Estático)

```
// Seleciona o título do livro e o nome de seu autor associado.  
Select.from(BOOKS)  
.columns(b -> b.title(),  
        b -> b.author().name().as("authorName"));
```

### Código (Dinâmico)

```
Select.from("bookshop.Books")  
.columns(b -> b.get("title"),  
        b -> b.get("author.name").as("authorName"));
```

# Estruturas Profundas vs. Planas: O Poder do `expand` e `inline`

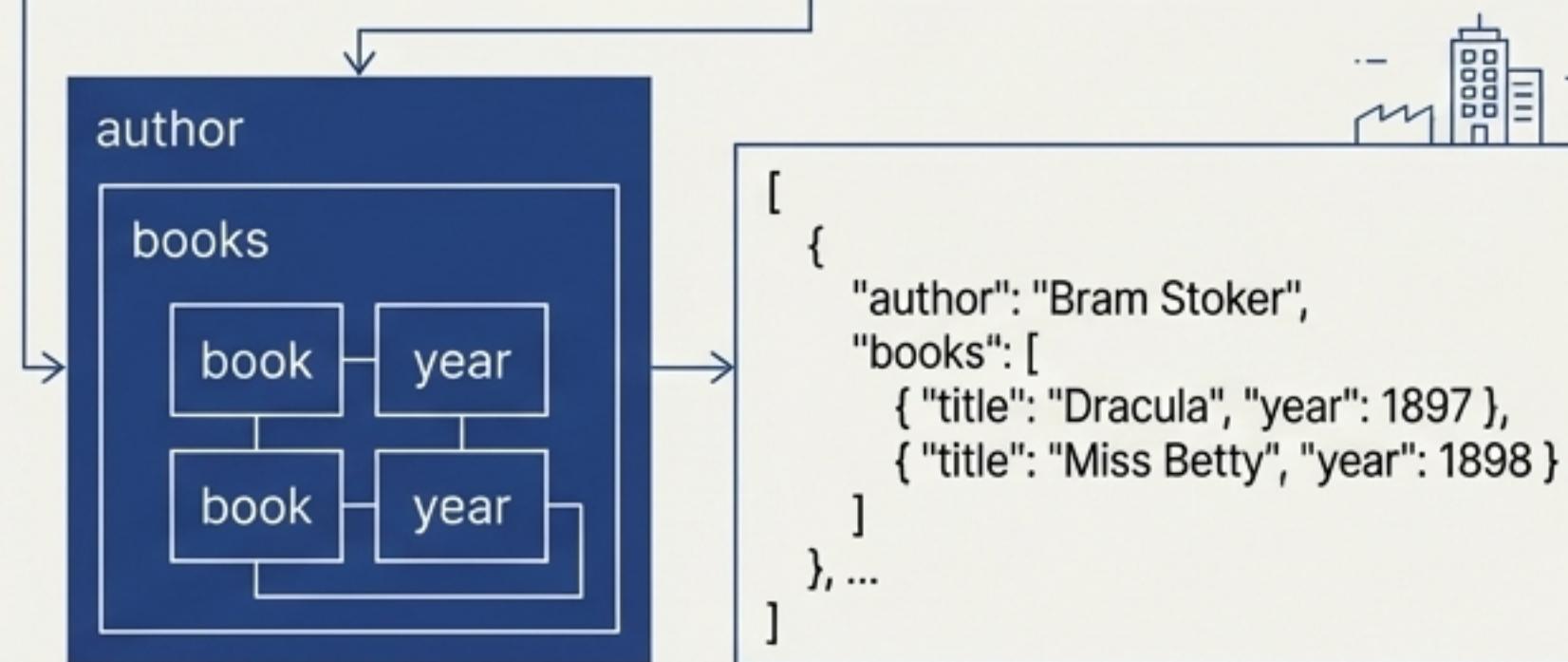
A API oferece controle total sobre a forma do resultado ao lidar com associações to-many. Escolha entre criar documentos aninhados (`expand`) ou achatar a estrutura (`inline`).

## `expand` – Criando Hierarquias

Expande associações para criar subestruturas aninhadas no resultado JSON. Ideal para retornar documentos completos.

```
Select.from(AUTHORS)
    .columns(a -> a.name().as("author"),
             a -> a.books().expand(
                 b -> b.title().as("book"),
                 b -> b.year()
             ));

```

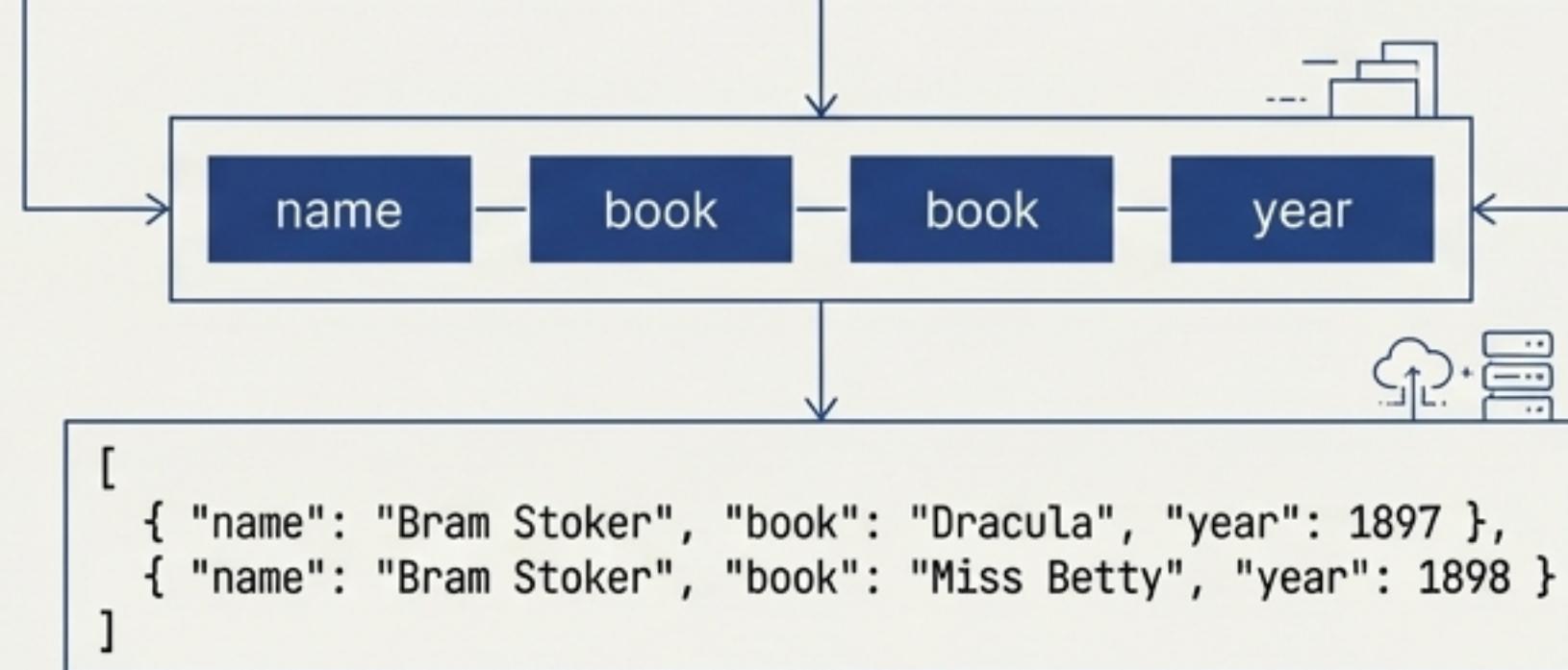


## `inline` – Achatando Estruturas

Inclui elementos de entidades associadas em uma estrutura de resultado plana. É uma notação curta para múltiplas path expressions.

```
Select.from(AUTHORS)
    .columns(a -> a.name(),
             a -> a.books().inline(
                 b -> b.title().as("book"),
                 b -> b.year()
             ));

```



# Filtragem Precisa: Atalhos Convenientes e o Poder do `where`

A API oferece múltiplos métodos para filtrar o conjunto de resultados, cada um adequado a um cenário diferente.



## 1. `byId(key)`

A forma mais simples de buscar uma entidade por sua chave primária (não suportado para chaves compostas).

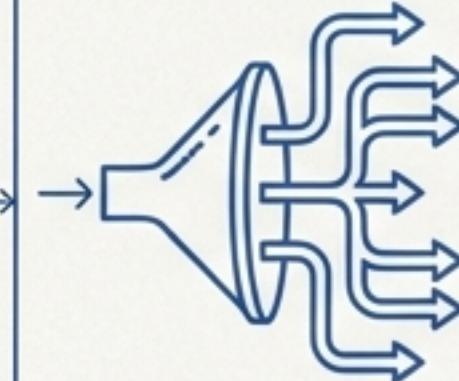
```
Select.from("bookshop.Authors").ById(101);
```



## 2. `matching(map)`

Um estilo 'query-by-example'. Filtra entidades onde os valores dos elementos correspondem a um mapa de chave-valor. Suporta paths.

```
Map<String, Object> filter = new HashMap<>();  
filter.put("author.name", "Edgar Allen Poe");  
filter.put("stock", 0);  
Select.from("bookshop.Books").matching(filter);
```



## 3. `where(lambda)`

O método mais poderoso e flexível. Permite a construção de predicados complexos usando operadores lógicos ('and', 'or', 'not') e funções.

```
Select.from(BOOKS)  
.where(b -> b.author().name().eq("Twain")  
.and(b.title().startsWith("A")) // `and  
.or(b.title().endsWith("Z")) //  
);
```

# Filtros Dinâmicos: Parametrização e Busca Textual

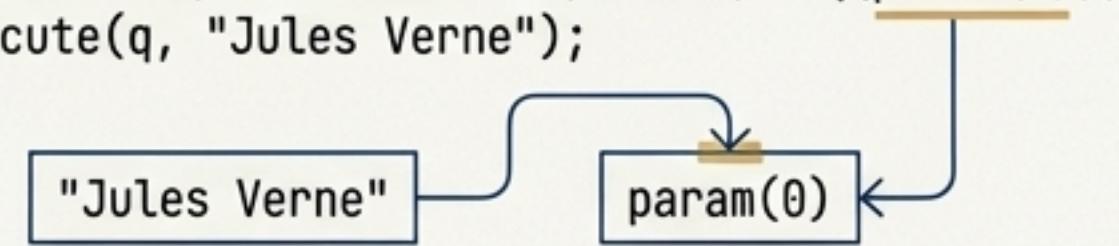
Prepare suas consultas para execução dinâmica e adicione capacidades de busca textual de forma integrada.

## Consultas Parametrizadas

Use `param()` para criar placeholders em suas consultas, que são preenchidos de forma segura em tempo de execução. Essencial para prevenir SQL injection.

`byParams: Uma alternativa mais simples para filtros de igualdade baseados em parâmetros.

```
import static com.sap.cds.ql.CQL.param;  
  
CqnSelect q = Select.from(BOOKS)  
    .where(b -> b.author().name().eq(param(0)));  
dataStore.execute(q, "Jules Verne");
```



```
CqnSelect q = Select.from(BOOKS).byParams("author.name");  
dataStore.execute(q, singletonMap("author.name", "Jules Verne"));
```

## Busca Textual com `search`

Adiciona um predicado que filtra entidades onde qualquer elemento 'pesquisável' contém um termo. Elementos `String` são pesquisáveis por padrão, mas isso pode ser customizado com a anotação `@cds.search`.



E una cor por livros que contenham tamida qu contenstonande Allen aue tin:tia rasso. Elos comeostenemi estario, apara deviento e de Heights sumolario consecida a conlica alemem.

```
// Busca por livros que contenham "Allen" OU "Heights"  
Select.from("bookshop.Books")  
    .columns("id", "name")  
    .search(term -> term.has("Allen").or(term.has("Heights")));
```

# Aggregando Dados: De Grupos a Cálculos Sobre Associações

A API suporta agregação de dados de duas formas principais: agrupando o resultado com `groupBy` e calculando agregações sobre associações to-many diretamente na projeção.

## Agregação com `groupBy` e `having`

Agrupe linhas baseadas em um ou mais elementos e use funções de agregação (count, sum, avg, max, min). Use `having` para filtrar os grupos resultantes.

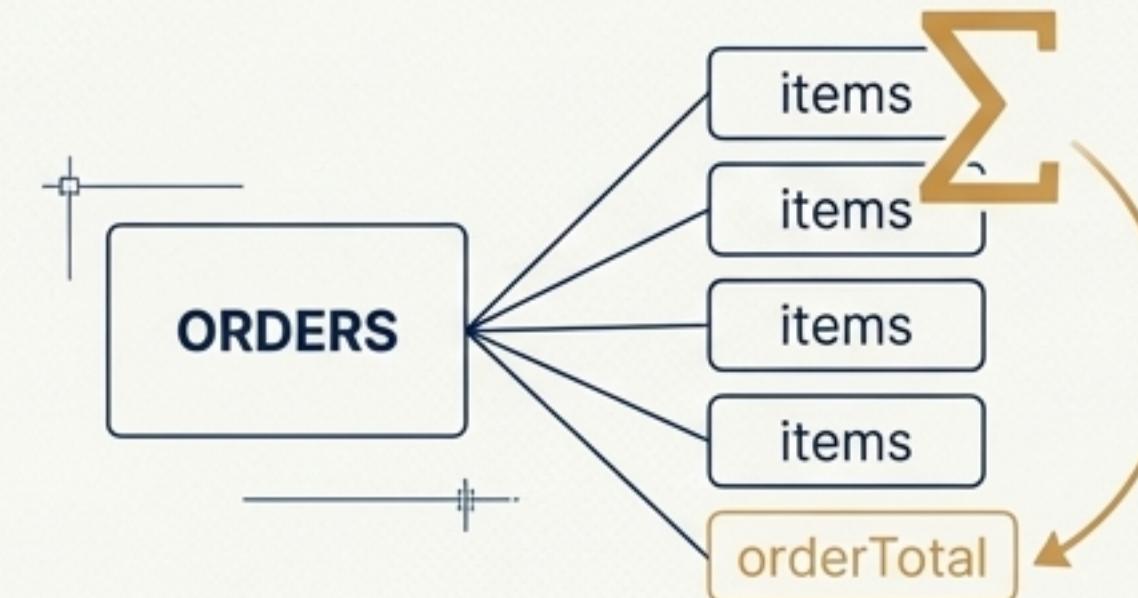


Contar autores com o mesmo nome e filtrar aqueles com mais de 2 ocorrências.

```
Select.from("bookshop.Authors")
  .columns(c -> c.get("name"), c -> func("count", c.get("name")).as("count"))
  .groupBy(c -> c.get("name"))
  .having(c -> func("count", c.get("name")).gt(2));
```

## Aggregando Sobre Associações (Beta)

Calcule `min`, `max`, `sum` e `count` sobre elementos de coleções associadas diretamente, sem subqueries. Uma feature extremamente poderosa.



Selecionar o ID de cada pedido e o valor total (soma de `amount \* price` dos seus itens).

```
Select.from(ORDERS).columns(
  o -> o.id(),
  o -> o.items().sum(i -> i.amount().times(i.price())).as("orderTotal")
);
```

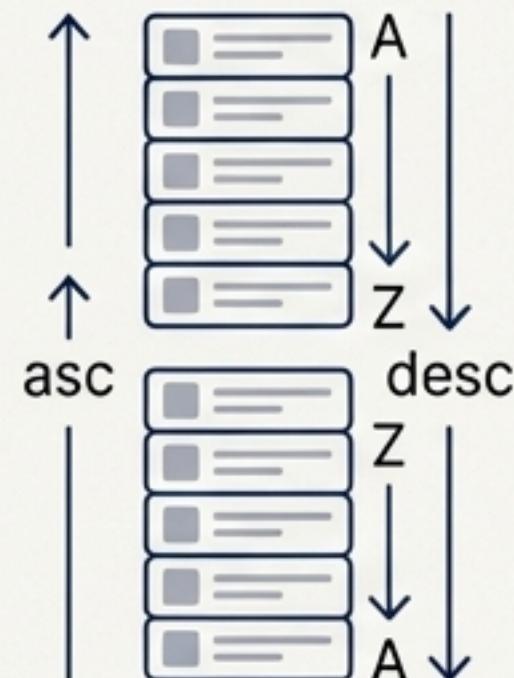
# Controle Fino: Ordenação e Paginação de Resultados

Garanta uma ordem de resultados consistente e gerencie grandes conjuntos de dados de forma eficiente com as cláusulas `orderBy`, `limit` e `offset`.

## Ordenação com `orderBy`

Especifique um ou mais elementos para ordenar o resultado, em ordem ascendente (`asc`) ou descendente (`desc`).

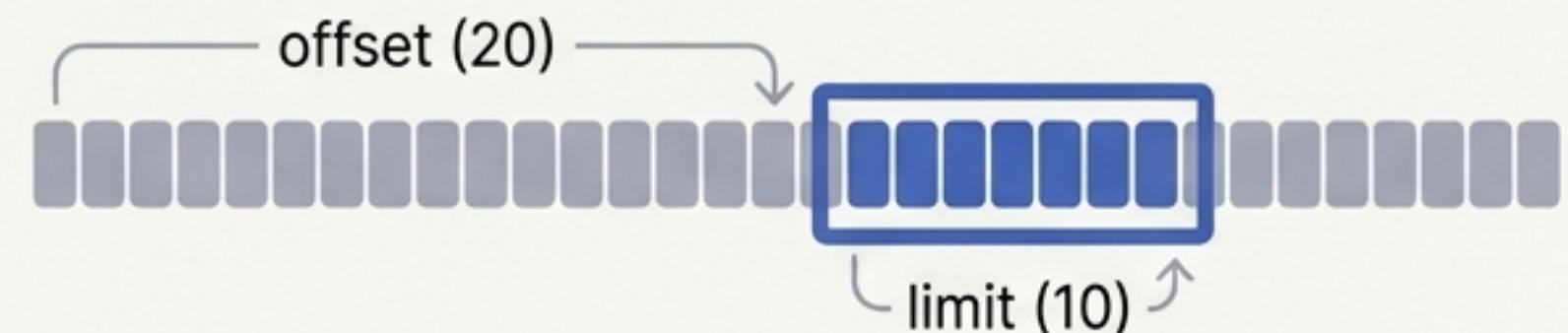
Controle de Nulos: Use `ascNullsLast` ou `descNullsFirst` para customizar o comportamento da ordenação de valores nulos.



```
Select.from("bookshop.Books")
    .orderBy(c -> c.get("ID").desc(),
             c -> c.get("title").asc());
```

## Paginação com `limit` e `offset`

Divida o resultado em "páginas" discretas. `limit(rows)` define o número máximo de linhas. `limit(rows, offset)` pula um número de linhas (`offset`) antes de retornar as `rows`.



Selecionar 10 livros, pulando os primeiros 20.

```
Select.from("bookshop.Books").limit(10, 20);
```

Nota: A paginação não é stateful. Inserções ou deleções entre requisições podem afetar as páginas subsequentes.

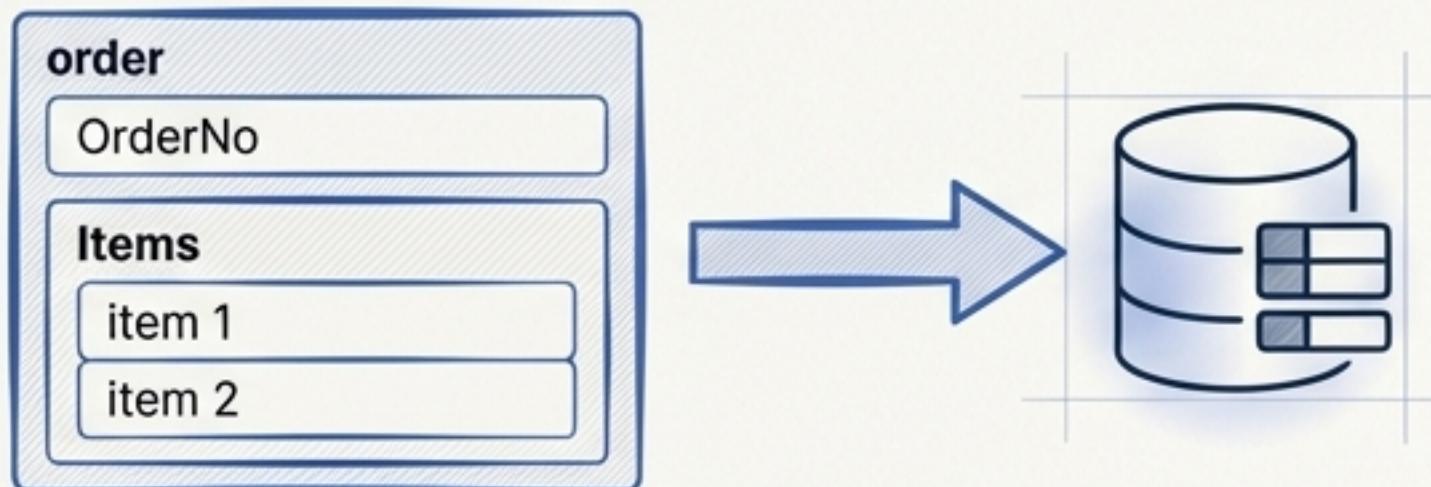
# Criando e Atualizando Dados: `Insert` e `Upser`

Adicione novos dados à sua base com as operações `Insert` e `Upser`, com suporte para operações em lote e estruturas de documentos complexas (deep inserts).

## `Insert`

Insere novos dados. Suporta inserção única (`entry`), em lote (`entries`) e aninhada (`deep insert`).

**Deep Insert:** Ao passar um mapa com valores que são listas de mapas (ex: um Pedido com seus Itens), a API realiza um `deep insert`, propagando a operação através de `Compositions`.



```
var items = List.of(Map.of("ID", 1, "book_ID", 101, "quantity", 1));  
var order = Map.of("OrderNo", "1000", "Items", items);
```

```
CqnInsert insert = Insert.into(ORDERS).entry(order);
```

## `Upser`

Atualiza um registro existente ou o insere se ele não existir. Principal caso de uso: replicação de dados.

**Semântica de "PATCH":** Apenas os valores fornecidos são atualizados ou inseridos. Elementos não informados mantêm seus valores existentes (ou `null` se for uma inserção).



**Atenção:** `Upser` não gera UUIDs, não inicializa valores default e não dispara handlers genéricos como audit logging. Os dados devem conter valores para todos os elementos chave e obrigatórios.

```
Books book = Books.create();  
book.setId(101); // Chave para encontrar o registro  
book.setTitle("CAP for Beginners");
```

```
CqnUpser upsert = Upser.into(BOOKS).entry(book);
```

# Modificando Estruturas: `Update` Individual e `Deep Update`

O statement `Update` permite modificar registros existentes com semântica de "patch". Vá além de atualizações simples e modifique documentos aninhados inteiros com `deep updates`.

## Update de Entidades Individuais



Use `Update.entity(...)` para especificar o alvo. O filtro pode ser definido pela chave no mapa de dados (`data`) ou explicitamente com `byId` ou `where`.

```
// Filtro隐式 pelo chave 'ID' dentro do objeto 'book'
Books book = Books.create();
book.setId(100);
book.setTitle("CAP Matters");
CqnUpdate update = Update.entity(BOOKS).data(book);

// Filtro explícito com byId
Update.entity(BOOKS).data("title", "CAP Matters").ById(100);
```

## Deep Update



Atualize estruturas de documentos que contêm `Compositions`. Por padrão, a operação é propagada apenas para `Compositions`, mas pode ser estendida para `Associations` com a anotação @cascade.

JSON Antes	Dados do Update	JSON Depois
{ "order_ID": "321", "status": "new", "Items": [ {"item_ID": "1", "quantity": 10}, {"item_ID": "2", "quantity": 2}, {"item_ID": "3", "quantity": 2} ] }	{ "order_ID": "321", "status": "in process", "Items": [ {"item_ID": "2", "quantity": 8}, {"item_ID": "4", "quantity": 1} ] }	{ "order_ID": "321", "status": "in process", "Items": [ {"item_ID": "2", "quantity": 8}, {"item_ID": "4", "quantity": 1} ] }

# O Dilema do Deep Update: `Full Set` vs. `Delta`

Ao atualizar coleções aninhadas (to-many), a API suporta duas estratégias distintas com implicações importantes.

## Representação `Full Set`



A coleção no payload do update representa o **estado final completo**. Quaisquer entidades pré-existentes que **não estejam** na coleção são **deletadas**.

Comportamento: Requer queries adicionais para determinar o que deletar, sendo menos eficiente.

**Antes**

```
items:  
[item1,  
 item2,  
 item3]
```

**Payload**

```
items:  
[item1_modificado,  
 item4_novo]
```

**Depois**

```
items:  
[item1_modificado,  
 item4_novo]
```

item2   
item3

## Representação `Delta`



A coleção no payload contém apenas as **diferenças** a serem aplicadas. Entidades não mencionadas permanecem intocadas. Entidades a serem removidas devem ser marcadas explicitamente.

Comportamento: Mais eficiente, pois descreve apenas as mudanças.

```
import static com.sap.cds.CdsList.delta;  
// ...  
// item2 é marcado para remoção  
order.setItems(delta(item1, item2.forRemoval(), item4));  
Update.entity(ORDER).data(order);
```

**Resultado:** `item1` é atualizado, `item2` é removido, `item3` permanece inalterado, e `item4` é adicionado.



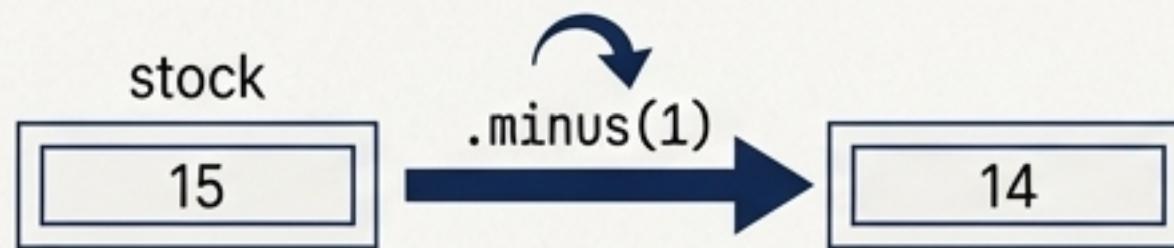
# Updates Dinâmicos: Expressões e Modificações em Lote

Vá além de fornecer valores estáticos. Use expressões para calcular novos valores no banco de dados e aplique a mesma mudança a múltiplos registros com um único statement.



## Update com Expressões (`set`)

O método `set` permite que o novo valor seja uma 'CqlValue', incluindo expressões aritméticas. Isso permite, por exemplo, incrementar ou decrementar um valor atomicamente.



Exemplo: Diminuir o estoque do livro 101 em 1.

```
Update.entity(BOOKS).ById(101)  
    .set(b -> b.stock(), s -> s.minus(1));
```

Estático:

```
Update.entity("bookshop.Books").ById(101)  
    .set("stock", CQL.get("stock").minus(1));
```

Dinâmico:



## Update em Lote (`Searched Update`)

Use a cláusula 'where' para aplicar a mesma atualização a todos os registros que correspondem ao filtro.

```
Update.entity(BOOKS).  
    data("stock", 100).where(b ->  
        b.title().contains("CAP"));
```

BOOKS

ID	TITLE	STOCK
101	"CAP Matters"	5
102	"Other Book"	20
103	"CAP in Depth"	8
104	"Java Basics"	30

Exemplo: Definir o estoque como 100 para todos os livros cujo título contém 'CAP'.

```
Update.entity(BOOKS).data("stock", 100)  
    .where(b -> b.title().contains("CAP"));
```

# Remoção Cirúrgica de Dados: A Operação `Delete`

O statement `Delete` remove registros de uma entidade. Por padrão, a operação é propagada em cascata através de `Compositions`.

## Métodos de Filtragem

### 1. `where(lambda)`

A forma mais comum e flexível de especificar quais registros deletar.

```
// Estático
import static bookshop.Bookshop_.ORDERS;
Delete.from(ORDERS).where(o -> o.OrderNo().eq("1000"));
```

### 2. `matching(map)`

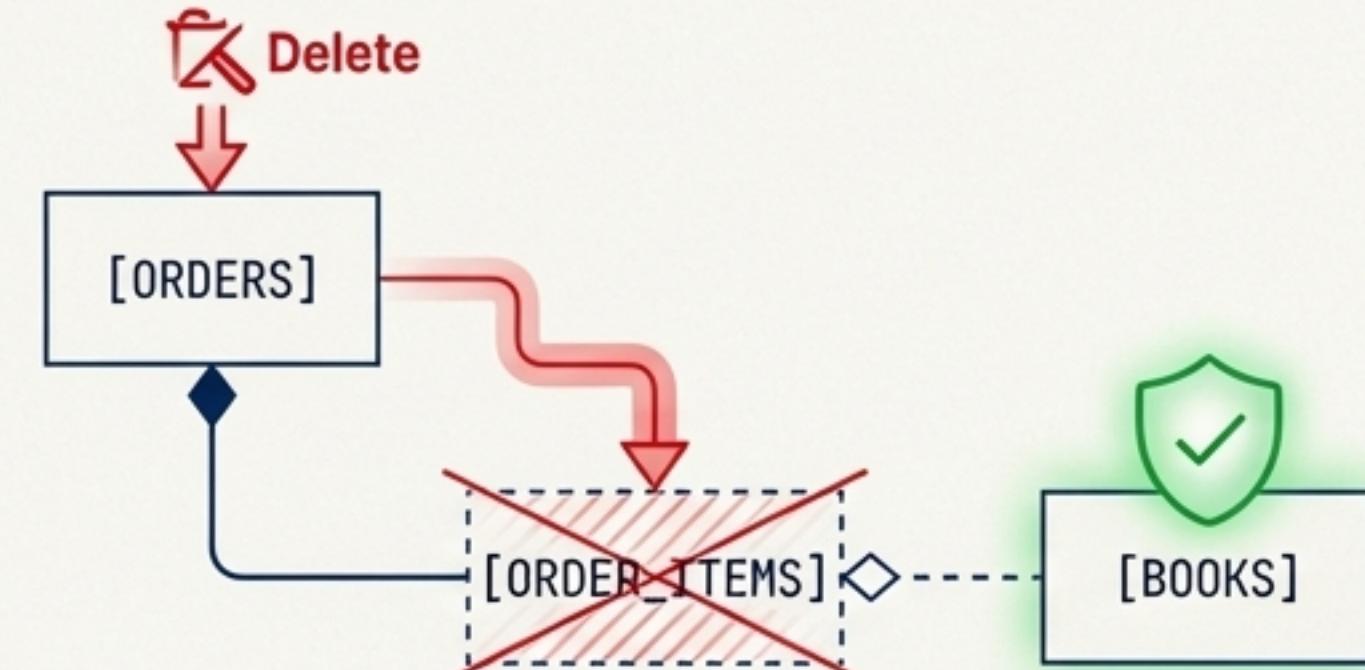
Útil para deletar baseado em um conjunto de valores, especialmente para entidades com chaves compostas.

```
Map<String, Object> params = new HashMap<>();
params.put("ID", 1);
params.put("journalID", 101);
CqnDelete delete = Delete.from("bookshop.Article").matching(params);
```

### 3. `byParams(...)`

Para deleções em lote parametrizadas, onde múltiplos conjuntos de chaves são passados na execução.

## Cascading Deletes



**Comportamento Padrão:** Ao deletar um `Order`, todos os seus `OrderItems` (ligados por `Composition`) são automaticamente deletados. Livros (ligados por `Association`) não são afetados.

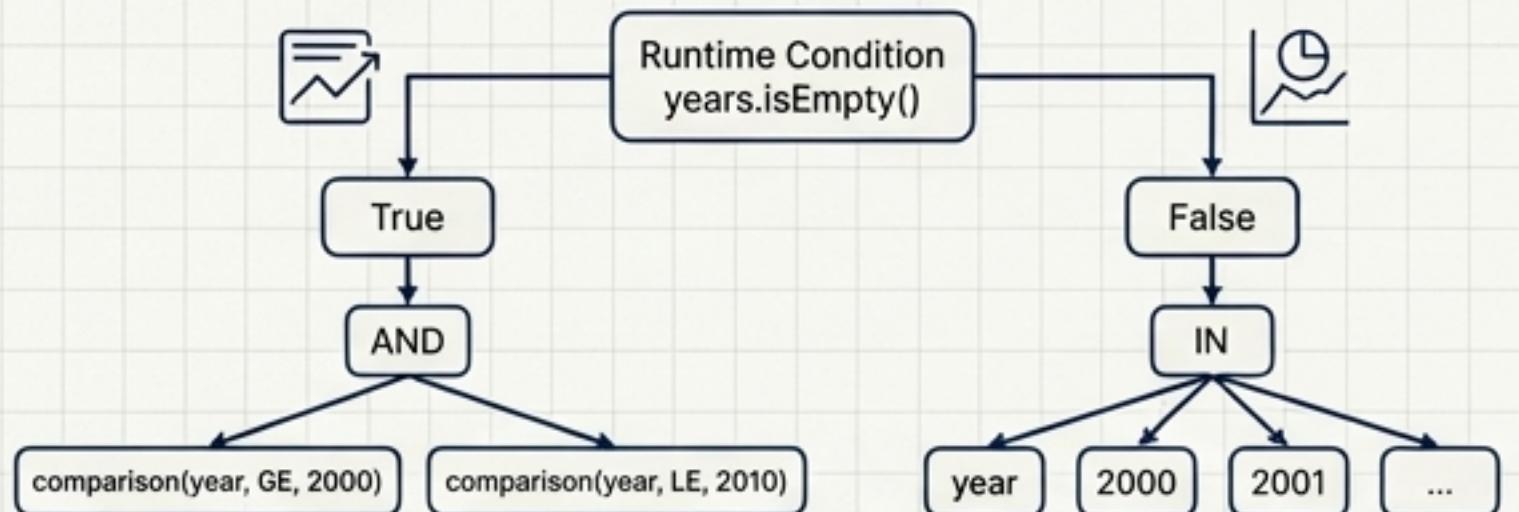
 **Customização:** Use a anotação `@cascade` para estender o comportamento de deleção em cascata para `Associations` selecionadas.

# Além do Fluente: Construindo Árvores de Expressão com a `CQL Interface`

Para o controle dinâmico máximo, a `CQL Interface` permite construir, modificar e reutilizar partes de um statement CQL como uma árvore de objetos. Isso é ideal para cenários onde a estrutura da consulta precisa ser montada programaticamente.

## Construindo Predicados Dinamicamente

Use `CQL.and()`, `CQL.or()`, `CQL.comparison()` para montar filtros complexos baseados em condições de runtime.



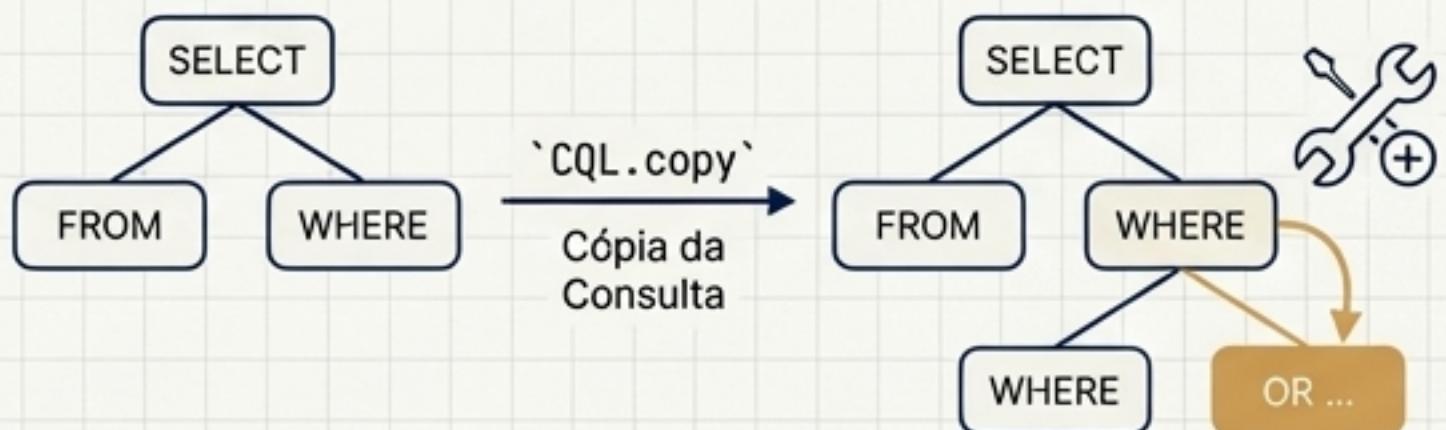
```
List<Integer> years = ...;  
CqnElementRef year = CQL.get("year");  
CqnPredicate filter;  
  
if (years.isEmpty()) {  
    filter = and(comparison(year, GE, val(2000)),  
                comparison(year, LE, val(2010)));  
} else {  
    List<Value<Integer>> yearValues = ...;  
    filter = CQL.in(year, yearValues);  
}  
Select.from("bookshop.Books").where(filter);
```

**Exemplo:** Construir um filtro que muda com base em uma lista de anos.

## Modificando Consultas Existentes com `CQL.copy`

Use a interface `Modifier` para criar uma cópia modificada de uma consulta existente, permitindo adicionar/remover colunas, alterar filtros ou substituir referências.

```
CqnSelect copy = CQL.copy(query, modifier);
```



**Aplicação:** Adicionar um `OR` a uma cláusula `where` existente, injetar paginação em uma consulta base, etc.

A API fluente é a sua ferramenta para o dia a dia. A `CQL Interface` é o seu canivete suíço para os desafios mais complexos, oferecendo controle total sobre cada nó da sua consulta.