

# Dominando o CQL em Java

Um Guia para a Construção Fluente e Type-Safe de Consultas

```
select().from()
    BatchStatement
        .where()
            .eq("tag")
        .limit()
    .limit() {
}
CqlSession
BatchStatement
TypeSafe().query()
}
```

# Além das Strings: A Vantagem da Construção Fluente

- A API CQL Builder é uma forma robusta e segura de construir instruções CQL, eliminando os riscos da concatenação de strings.
  - Prevenção de erros em tempo de compilação, legibilidade aprimorada e o poder do autocomplete da IDE, que acelera o desenvolvimento.

## O Jeito Antigo



```
// Propenso a erros e difícil  
de manter  
String cql = "SELECT title FROM  
bookshop.Books WHERE ID = 101";
```

## O Jeito Novo



```
// Seguro, legível e com  
suporte da IDE  
Select.from(BOOKS).columns(b ->  
b.title()).ById(101);
```

**Segurança, legibilidade e produtividade.**

# Dois Estilos, Um Objetivo: Flexibilidade e Segurança

## Estilo Dinâmico

Utiliza strings para referenciar entidades e elementos ("bookshop.Books"). Ideal para código genérico onde os nomes não são conhecidos em tempo de compilação.



## Estilo Estático

Utiliza constantes e interfaces geradas a partir do modelo CDS ('BOOKS'). Esta é a abordagem recomendada para a lógica de negócios, pois garante a verificação de tipos (*type-safety*) pelo compilador.

## Dinâmico

```
Select.from("bookshop.Books").columns("title").ById(101);
```

## Estático

```
import static bookshop.Bookshop_.BOOKS;
```

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

**Insight:** Use o estilo estático sempre que possível para um código mais robusto e fácil de manter.

# Lendo Dados: A Anatomia de um `SELECT`

Desconstrução das partes essenciais de uma consulta de leitura:

- **`from` (Fonte)**

A origem dos dados. Pode ser uma entidade ('BOOKS'), uma referência com filtro ('ORDERS', o -> o.filter(...)) ou até uma subquery.

- **`columns` (Projeção)**

Os elementos que você deseja retornar. Por padrão, todos os elementos são selecionados.

Expressões de caminho como 'b.author().name()' são automaticamente convertidas em `JOIN`s pelo runtime, simplificando drasticamente o código.

```
// Seleciona o título do livro e o nome do seu autor  
Select.from(BOOKS) ← Entidade  
    .columns( ← Fonte  
        b -> b.title(),  
        b -> b.author().name().as("authorName") ← Projeção  
    ) ← dos dados  
    .where(b -> b.author().name().startsWith("A"));
```

Path expression →  
`LEFT OUTER JOIN`

Entidade  
Fonte

Projeção  
dos dados

# Filtragem Inteligente e Precisa



## `where` para Condições Complexas

Use operadores lógicos (`and`, `or`, `not`) e uma vasta gama de predicados para construir filtros detalhados.

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



## `byId` para Chaves Primárias

O atalho mais eficiente para buscar uma única entidade por sua chave. Não suportado para chaves compostas.

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



## `matching` para Query-by-Example

Filtre usando um mapa de chave-valor. Suporta caminhos para entidades associadas.

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



## `byParams` para Consultas Parametrizadas

Simplifica a criação de filtros que serão preenchidos em tempo de execução.

```
// Alternativa a b.title().eq(param("title"))
Select.from(BOOKS).byParams("title", "author.name");
```

# Moldando o Resultado: De Listas Planas a Documentos Estruturados

## Conceito e Código

### `expand` para Resultados Aninhados

Cria subestruturas no JSON de resultado, ideal para associações *to-many*. Preserva a hierarquia dos dados.

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

```

### `inline` para Resultados Achatados

Traz campos de entidades associadas para o nível superior, criando uma estrutura plana. Útil para relatórios tabulares.

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

```

## Visualização do JSON Resultante

### Resultado do `expand`

```
[
  {
    "author": "Bram Stoker",
    "books": [
      {"title": "Dracula", "year": 1897},
      {"title": "Miss Betty", "year": 1898}
    ],
    ...
]
```

### Resultado do `inline`

```
[
  {"name": "Bram Stoker", "book": "Dracula", "year": 1897},
  {"name": "Bram Stoker", "book": "Miss Betty", "year": 1898}
]
```

# Da Informação Bruta ao Insight

## Conceitos

 **Agregação**: Use funções como `count`, `sum`, `max`, `avg`, `min` dentro da projeção (`columns`) para resumir dados.

 **Agrupamento** (`groupBy`): Agrupa linhas para que as funções de agregação operem em cada grupo.

 **Filtragem de Grupos** (`having`): Filtra os resultados \*após\* o agrupamento e a agregação. Essencialmente, é um `where` para os grupos.

 **Ordenação** (`orderBy`): Controla a ordem dos resultados (`asc`, `desc`). Inclui controle sobre a posição de valores nulos (`ascNullsLast`, `descNullsFirst`).

## Consulta: Encontrar autores com mais de 2 livros

```
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));
```

1. Agrupar por nome

2. Filtrar grupos onde a contagem é > 2

## Resultado Esperado

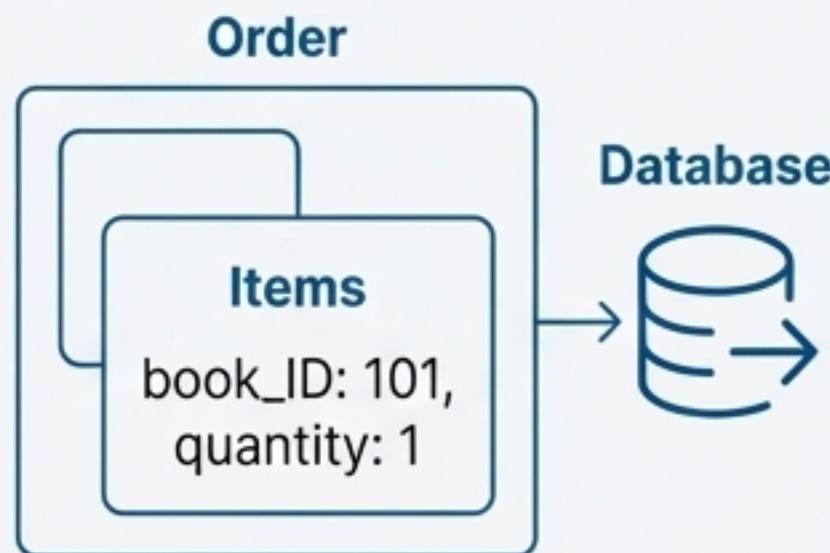
name	count
Smith	3

# Criando e Sincronizando Dados

## INSERT - Adicionando Novas Entradas

Suporta inserção de uma única entrada (`entry`) ou múltiplas (`entries`). Permite "deep inserts" em `Compositions`, criando uma árvore de objetos de uma só vez.

```
var items = List.of(Map.of(  
    "ID", 1,  
    "book_ID", 101,  
    "quantity", 1));  
  
var order = Map.of(  
    "OrderNo", "1000",  
    "Items", items);  
  
CqInsert insert =  
    Insert.into(ORDERS)  
    .entry(order);
```



## UPsert - A Operação Híbrida

Atualiza um registro se ele existir, ou insere um novo caso contrário. O principal caso de uso é a replicação de dados.

Opera com semântica de "PATCH" – apenas os campos fornecidos são alterados.

### ⓘ Aviso Importante

- Não gera valores de chave UUID.
- Não aplica valores padrão se não forem fornecidos.
- Não dispara anotações @cds.on.insert.
- Requer que todos os valores de chave e campos obrigatórios sejam fornecidos.

# Modificando Dados com Confiança

## Formas de Atualização

### Atualização Individual

Use `Update.entity(...)` com `data(...)` e filtre com `byId` ou `where`.

### Atualização com Expressões

Modifique valores com base em seu estado atual, perfeito para operações como decremento de estoque.

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

## O Poder do "Deep Update"

Ao atualizar entidades aninhadas (composições), você pode descrever as mudanças de duas maneiras:

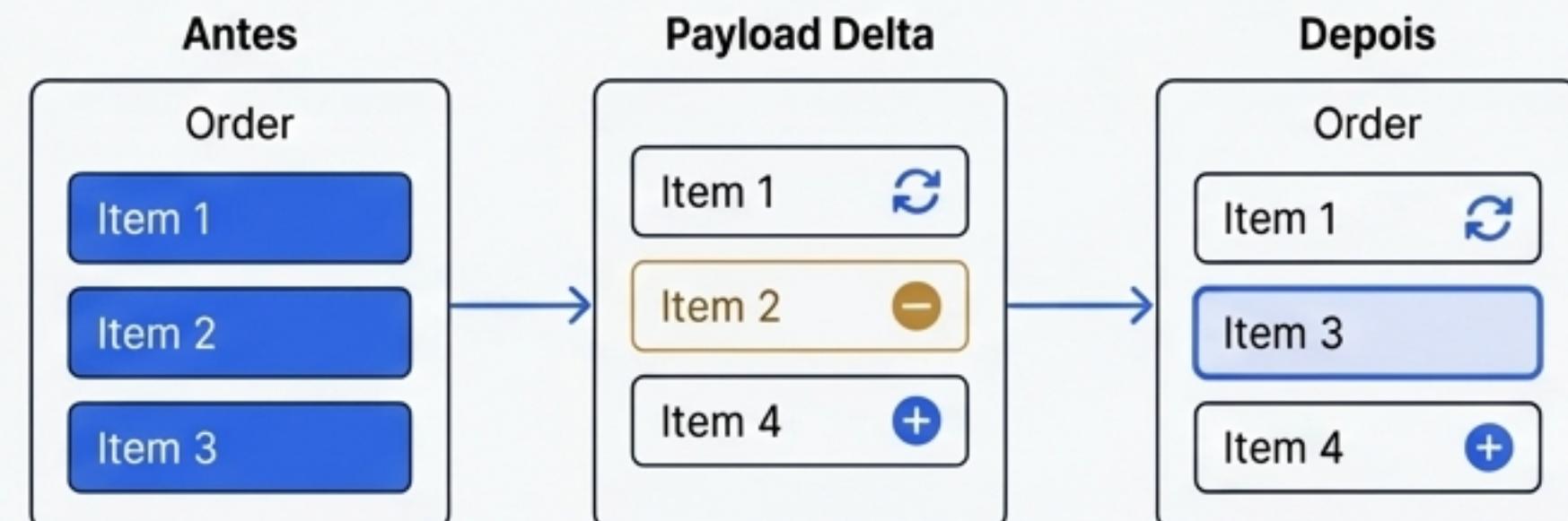
- **Full Set (Representação Completa)**: A coleção aninhada fornecida substitui completamente a existente. Itens não presentes na nova lista são excluídos.
- **Delta (Representação de Diferença)**: A coleção aninhada descreve apenas as mudanças. Itens existentes não mencionados permanecem intocados. Itens a serem removidos devem ser marcados explicitamente.

### Exemplo Visual (Delta)

Um item modificado (`item1`), um novo item adicionado (`item4`), e um item marcado para remoção (`item2.forRemoval()`).

```
order.setItems(delta(item1, item2.forRemoval(), item4));  
Update.entity(ORDER).data(order);
```

Mostra que o `OrderItem 3` do estado original (não mencionado no delta) permanece no resultado final, ao contrário do comportamento de 'full set'.



# Excluindo Dados e Suas Dependências

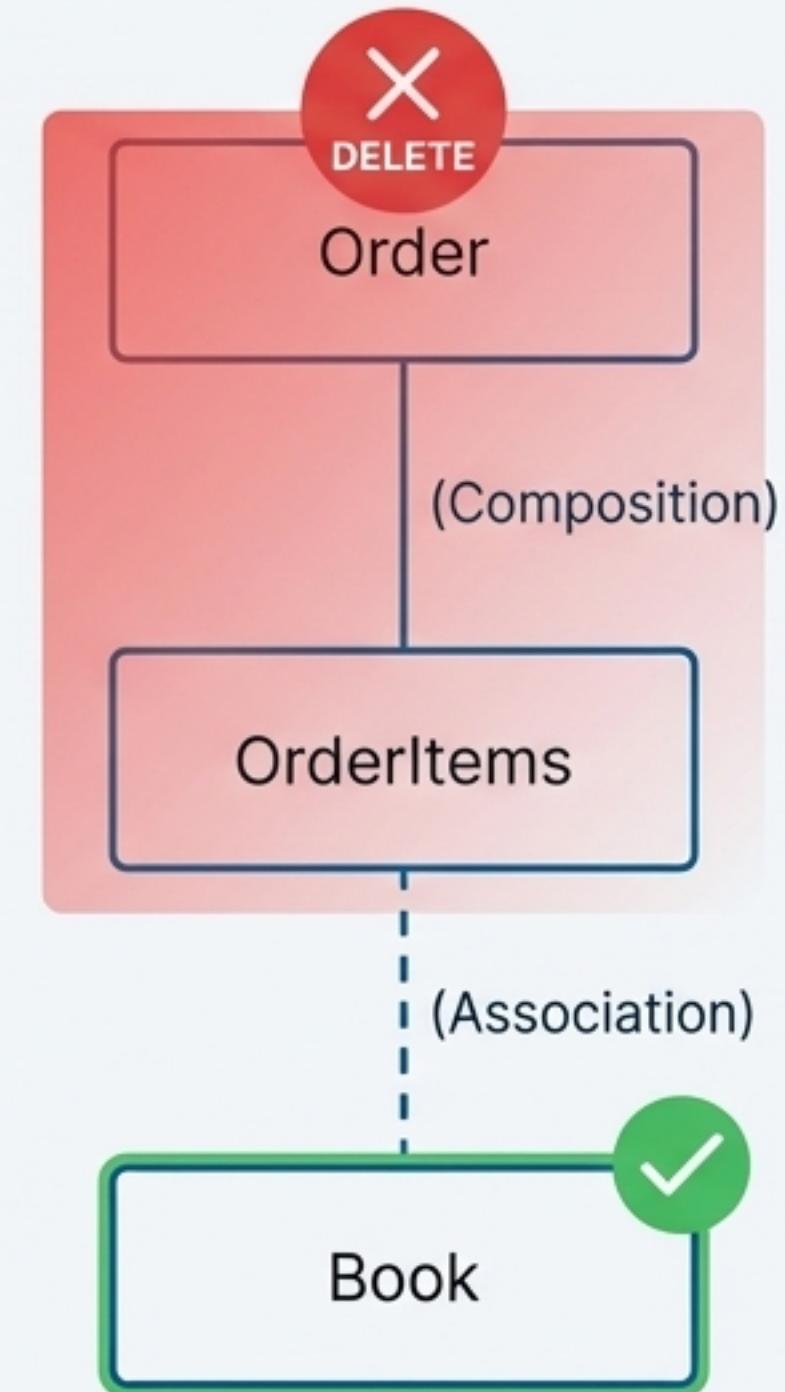
## Filtros

A especificação do que deletar utiliza os mesmos métodos de filtragem do `SELECT`: `where`, `matching` e `byParams`.

```
CqnDelete delete = Delete.from(ORDERS)
    .where(b -> b.OrderNo().eq(1000));
```

## A Regra da Cascata

- **Compositions:** A exclusão se propaga automaticamente.  
Deletar uma `Order` deleta seus `OrderItems`.
- **Associations:** A exclusão **não** se propaga por padrão.  
Deletar um `OrderItem` não deleta o `Book` associado.
- **Exceção:** A propagação pode ser forçada em associações com a anotação `@cascade`.



# Da Construção à Execução

## A Interface Universal

Todas as instruções CQL (Select, Insert, Update, Delete) são executadas através do método `service.run(cqn)`.

## Execução Parametrizada

A maneira segura e performática de executar consultas. Use `param()` ou `byParams` para criar placeholders que são preenchidos em tempo de execução. Isso previne SQL Injection.

## Execução em Lote (Batch)

Execute uma mesma instrução `UPDATE` ou `DELETE` com múltiplos conjuntos de parâmetros de uma só vez, otimizando a comunicação com o banco de dados.



## Exemplo de Código: Batch Update

```
CqnUpdate update = Update.entity(BOOKS).data("stock", 0)
    .byParams("title", "author.name");

Map<String, Object> paramSet1 = Map.of(
    "author.name", "Victor Hugo",
    "title", "Les Misérables"
);
Map<String, Object> paramSet2 = Map.of(
    "author.name", "Emily Brontë",
    "title", "Wuthering Heights"
);

// Executa a mesma atualização para dois conjuntos de parâmetros
Result result = service.run(update, List.of(paramSet1, paramSet2));
```

# Operando em Abstrações e Ambientes Concorrentes

## Views e Projeções

O runtime do CAP tenta resolver escritas e deleções em Views para a entidade de persistência subjacente, mas isso só é possível para projeções simples (sem agregações, joins, etc.).

- **Runtime Views (`@cds.persistence.skip`)**

Permitem alterar a definição da view sem reimplantar o schema do banco, resolvendo a projeção em tempo de execução.

## Controle de Concorrência



### Otimista (`@odata.etag`)

Previne 'lost updates'. O cliente envia a versão ('ETag') do dado que ele leu. A atualização só ocorre se a versão no banco for a mesma.



### Pessimista (`.lock()`)

Bloqueia os registros no banco, impedindo que outras transações os modifiquem até que a transação atual seja concluída.

```
// Adquire um bloqueio exclusivo (write) com timeout de 5 segundos
Select.from("bookshop.Books").byId(1).lock(5);
```

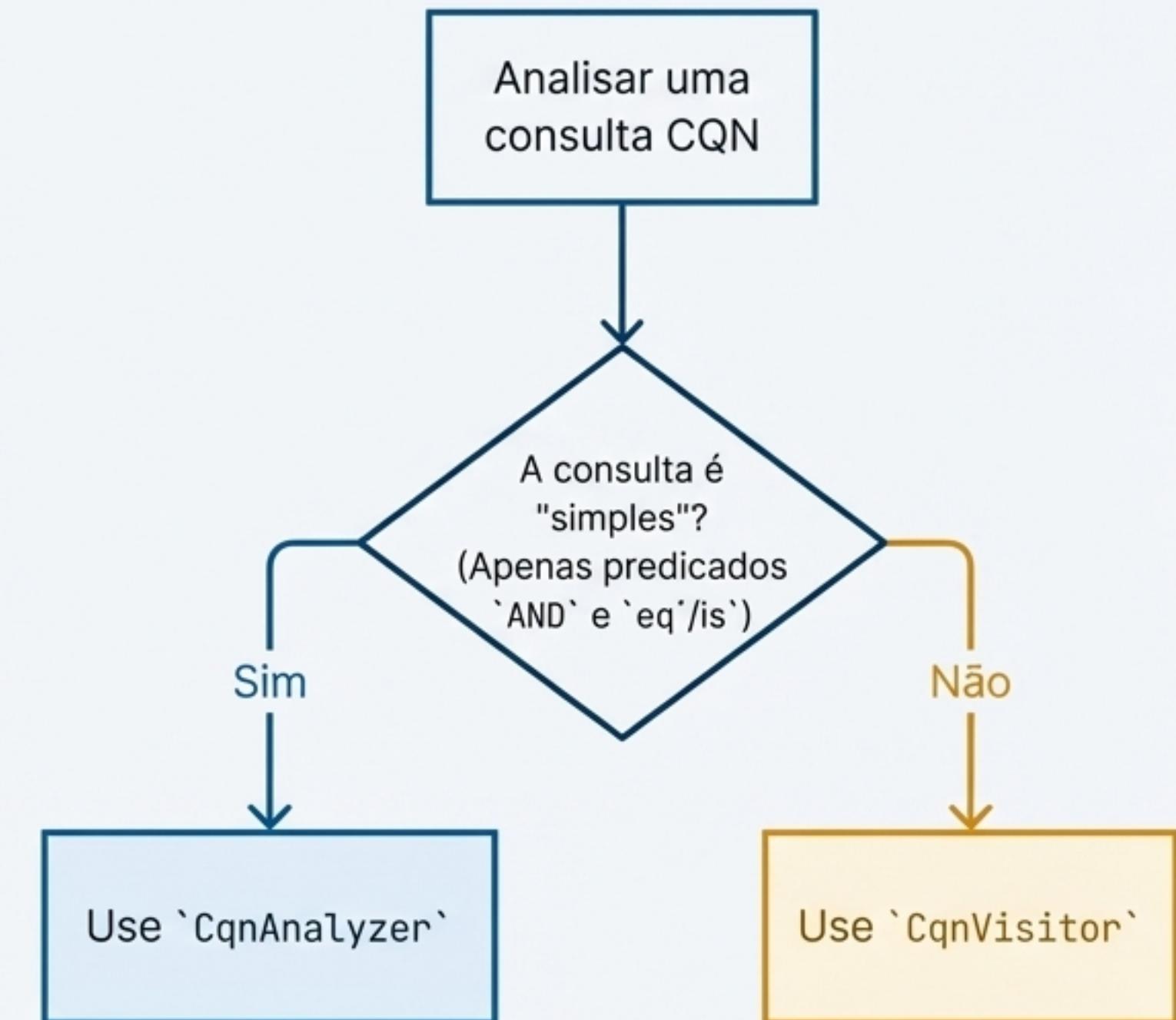
# Introspecção: Analisando a Consulta

## Caso de Uso

Necessário para handlers genéricos, validações dinâmicas, auditoria ou para entender a estrutura de uma consulta recebida.

## As Ferramentas

- `CqnAnalyzer`: A via rápida. Ideal para extrair valores de filtros simples, onde predicados são conectados apenas por `AND` e usam operadores `eq` ou `is`.
- `CqnVisitor`: A ferramenta poderosa. Necessária para atravessar árvores de expressão complexas com operadores `OR`, `gt`, `lt`, funções, subqueries, etc.



# Processando os Resultados com `CdsResult`

## Conceitos

### Acesso aos Dados

O `CdsResult` é um `Iterable` de `CdsData` (que é um `Map`). Pode ser percorrido com um loop `for` ou processado com a API de Streams (`.stream()`).

### Acesso Tipado

Obtenha resultados diretamente como uma lista ou um único objeto de uma interface gerada, garantindo segurança de tipo e clareza no código.

```
// Retorna um único objeto do tipo 'Book'  
Book book = result.single(Book.class);
```

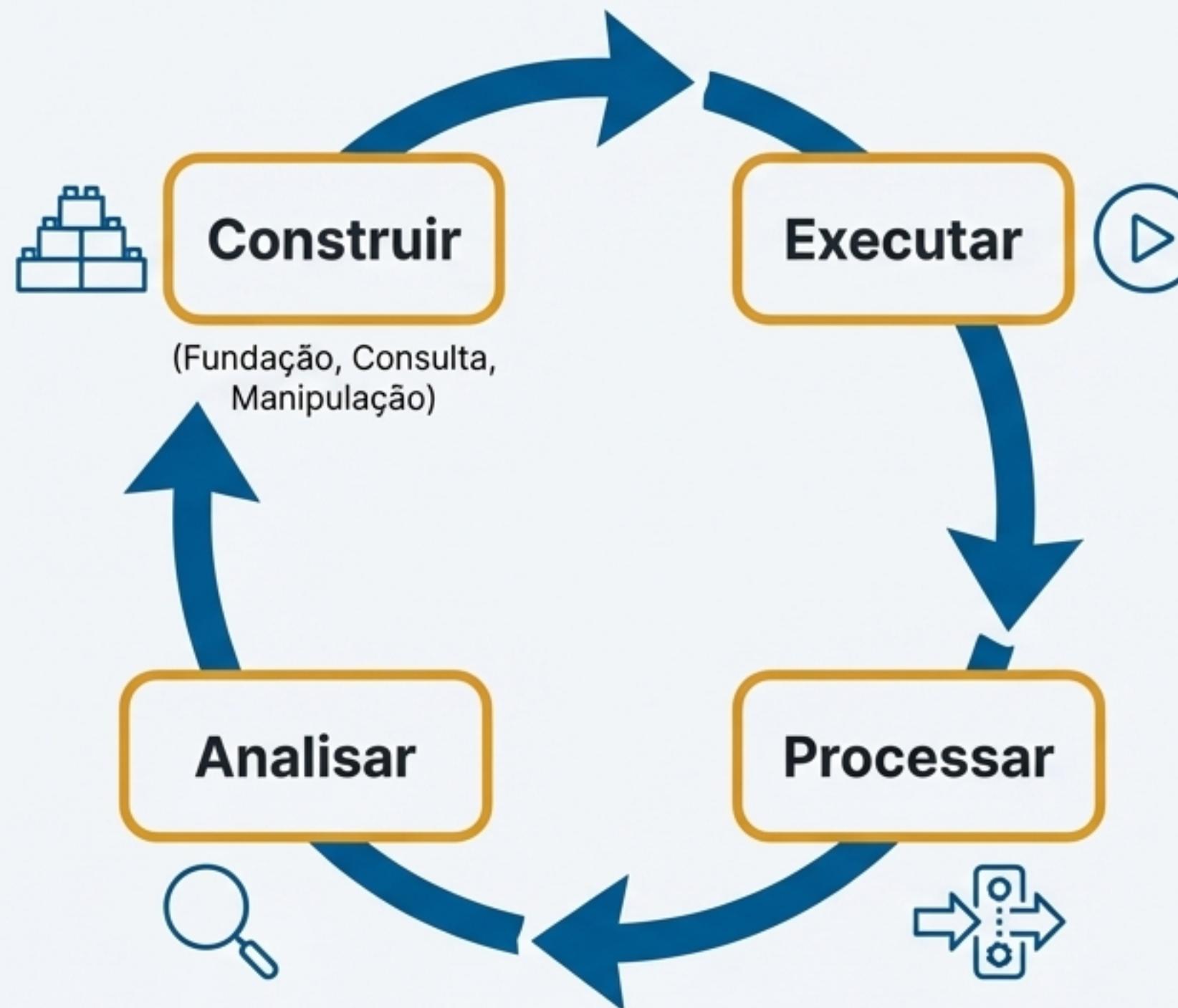
### Referências de Entidade (`.ref()`)

A partir de uma linha de resultado (`CdsData`), obtenha uma referência tipada que encapsula as chaves daquela entidade. Essa referência pode ser usada para construir novas consultas (`UPDATE`, `DELETE`, etc.) para aquele registro específico.

## Exemplo de Código

```
// 1. Executa uma consulta para encontrar um autor  
Author authorData = service.run(query).single(Author.class);  
  
// 2. Obtém uma referência tipada para aquele autor  
Author_ authorRef = authorData.ref();  
  
// 3. Usa a referência para construir uma nova consulta  
CqnInsert insertBook = Insert.into(authorRef.books()).entry(...)  
service.run(insertBook);
```

# Sua Jornada com o CQL Builder



## Principais Takeaways

- Prefira o **estilo estático** para robustez e manutenibilidade.
- Domine as projeções (`expand`, `inline`) para controlar a estrutura dos dados retornados.
- Entenda a diferença crucial entre '**full set**' e '**delta**' em `UPDATE`'s profundos.
- Sempre use **parâmetros** (`byParams`) para segurança e performance.

**Construa com confiança.**

[docs.capire.sap/java/cql-builder](https://docs.capire.sap/java/cql-builder)