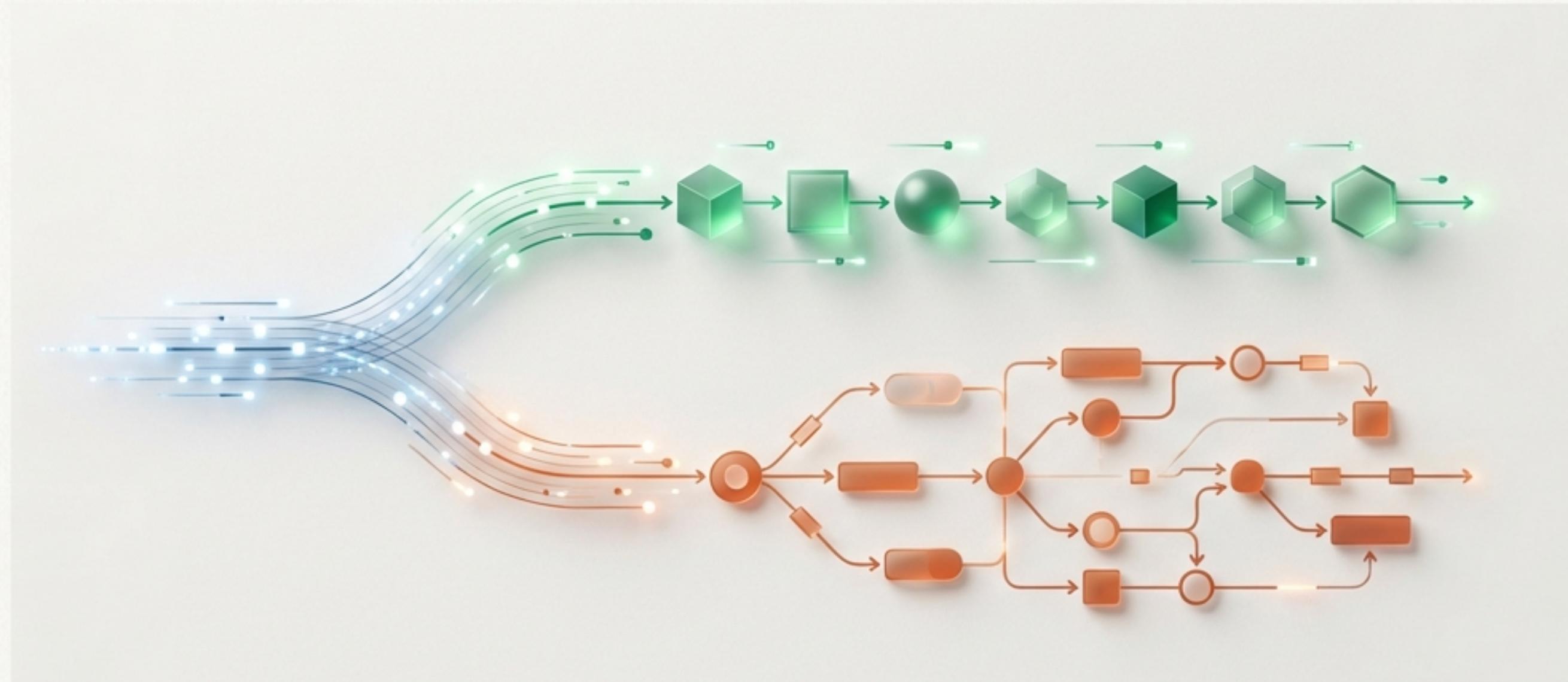


Introspecção de Declarações CQL em Java

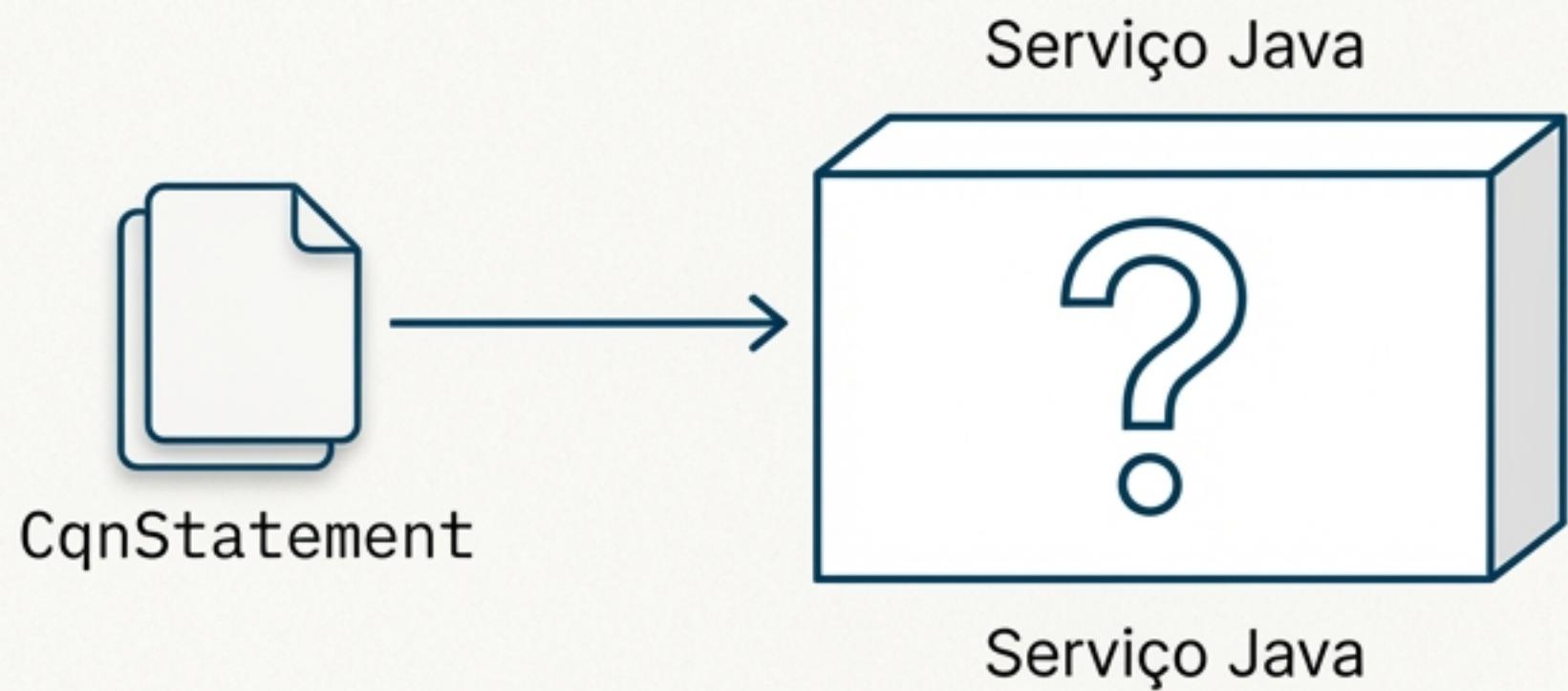
Escolhendo a Ferramenta Certa: CqnAnalyzer vs. CqnVisitor



Seu serviço precisa entender a query que recebe.

Handlers de serviços baseados em CQN frequentemente precisam inspecionar as declarações recebidas para aplicar validações, enriquecer dados ou implementar lógicas customizadas.

Mas como analisar essas declarações de forma eficaz em Java?



Dois Caminhos para a Introspecção



CqnAnalyzer

Uma API especializada para extrair valores de filtro e analisar a estrutura de **referências** de forma direta e inequívoca. Ideal para casos de uso simples e bem definidos.



CqnVisitor

Uma API de propósito geral, baseada no padrão de projeto **Visitor**, para percorrer árvores de tokens CQN. Oferece poder e flexibilidade para analisar queries complexas.

CqnAnalyzer: A Análise Direta e Especializada

O `CqnAnalyzer` foi projetado para analisar queries onde os valores de filtro podem ser **identificados sem ambiguidade**.

Regras Fundamentais

- 1. O operador do predicado de comparação deve ser `eq` ou `is`.
- 2. Apenas a conjunção `and` pode ser usada para conectar predicados.

```
// Válido: operador `eq`
Select.from("bookshop.Book")
    .where(b -> b.get("ID").eq(42));
```

```
// Válido: operadores 'eq' e 'is' conectados por 'and'
Select.from("bookshop.Book")
    .where(b -> b.get("ID").eq(42)
        .and(b.get("title").is("Capire")));
```

Nota: Esta regra se aplica a todos os segmentos de todas as referências da query.

CqnAnalyzer em Ação: Resolvendo Entidades

```
// 1. Crie o Analyzer a partir do seu CdsModel
CqnAnalyzer cqnAnalyzer = CqnAnalyzer.create(context.getModel());

// 2. Analise a referência da declaração CQN
AnalysisResult result = cqnAnalyzer.analyze(cqn.ref());

// 3. Acesse as entidades raiz e alvo
CdsEntity order = result.rootEntity();      // Resolvido para 'Orders'
CdsEntity item  = result.targetEntity();       // Resolvido para 'OrderItems'
```

Contexto

Modelo CDS (simplificado)

```
entity Orders {
    key OrderNo: String;
    Items: Composition of many OrderItems...
}
entity OrderItems {
    key ID: Integer;
    book: Association to Books; ...
}
```

Query CQL

```
SELECT from Orders[OrderNo = '42']:items[ID = 1]
```

Extraindo Valores de Filtro com `targetKeys()` e `targetValues()`

Se os valores de filtro podem ser determinados sem ambiguidade, o `CqnAnalyzer` pode extraí-los em um `Map<String, Object>`.

Exemplo 1: Extraindo Chaves de Múltiplos Segmentos

```
SELECT from Orders[OrderNo = '42']:items[ID = 1]  
  
Map<String, Object> rootKeys = result.rootKeys();  
Map<String, Object> rootKeys = result.rootKeys();  
String orderNo = (String) rootKeys.get("OrderNo"); // "42"  
  
Map<String, Object> targetKeys = result.targetKeys();  
Integer itemId = (Integer) targetKeys.get("ID"); // 1
```

Exemplo 2: Extraindo Valores de uma Cláusula `where`

```
SELECT from Orders.items where ID = 3 and status = 'open'  
  
Map<String, Object> filterValues = result.targetValues();  
String status = (String) filterValues.get("status"); // "open"
```

Navegando por Múltiplos Segmentos com o Iterator

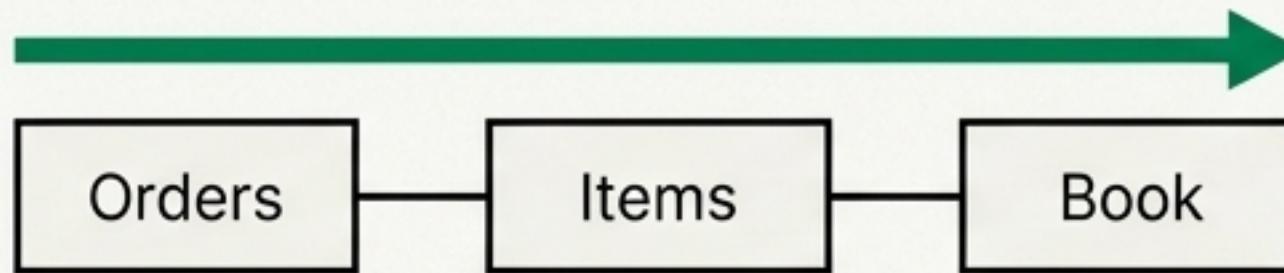
O que acontece quando a referência tem mais de dois segmentos?

```
SELECT from Orders[OrderNo = '42']:items[ID = 1].book
```

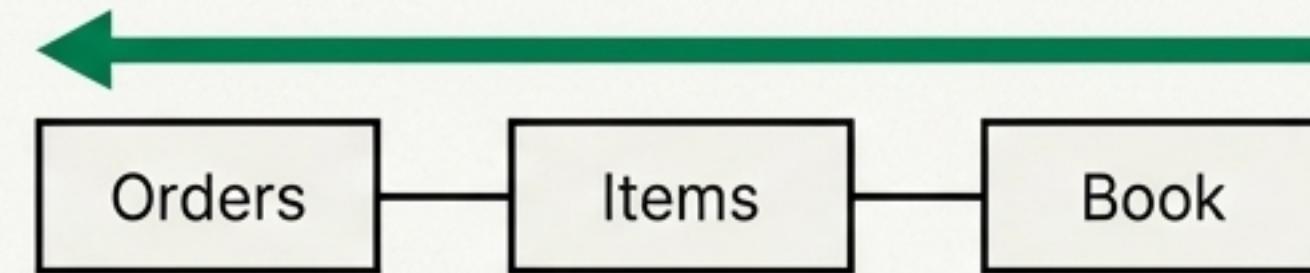
Segmentos Resolvidos

Use o `AnalysisResult.iterator()` para percorrer cada segmento resolvido na ordem.

Iterator Direto



Iterator Reverso



```
Iterator<ResolvedSegment> iterator = result.iterator();
CdsEntity order = iterator.next().entity(); // Orders
CdsEntity item = iterator.next().entity(); // OrderItems
CdsEntity book = iterator.next().entity(); // Books
```

```
Iterator<ResolvedSegment> iterator = result.reverse();
CdsEntity book = iterator.next().entity(); // Books
CdsEntity item = iterator.next().entity(); // OrderItems
CdsEntity order = iterator.next().entity(); // Orders
```

Quando o CqnAnalyzer Atinge seus Limites

‘CqnAnalyzer’ é poderoso para filtros simples e diretos.

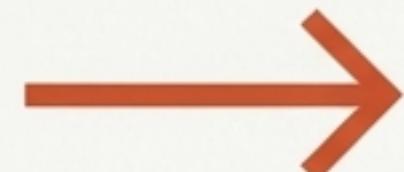
No entanto, ele não é adequado para analisar queries que envolvem lógica mais complexa, onde os valores não são mapeados de forma única.

Cenários que Exigem uma Ferramenta Mais Poderosa:



- ‘lt’, ‘gt’, ‘le’, ‘ge’, ‘ne’, ‘isNot’
- Uso de predicados ‘in’
- Negação com ‘not’
- Predicados de busca (search)
- Uso de funções ou subqueries

Para esses casos, precisamos de uma abordagem diferente: o ‘CqnVisitor’.

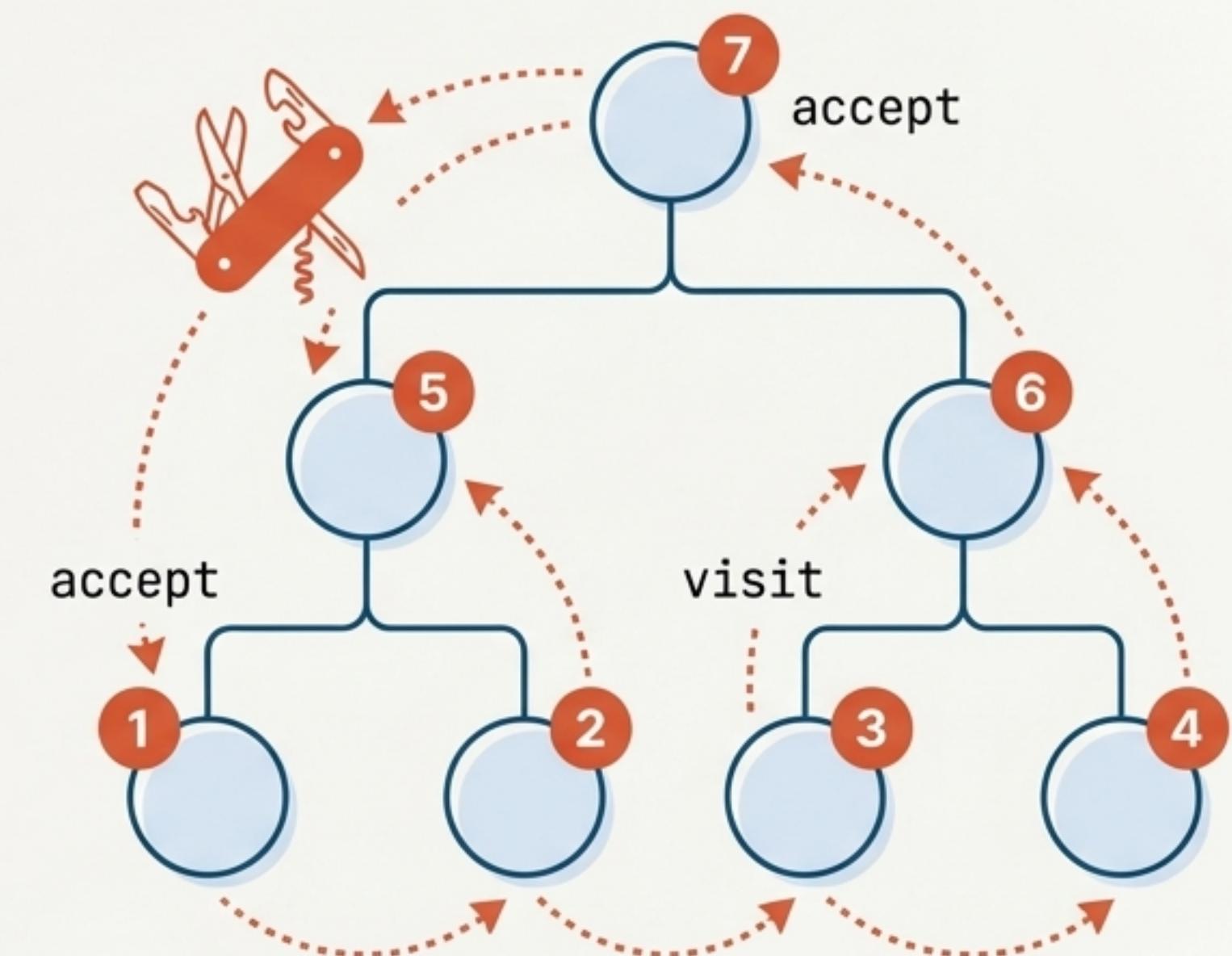


CqnVisitor: O Poder da Travessia de Árvores

O `CqnVisitor` permite percorrer árvores de tokens CQN (expressões, predicados, valores).

Como Funciona (em 3 passos)

1. `accept(visitor)`: Você passa uma implementação do `CqnVisitor` para o método `accept` de um token CQN.
2. **Travessia Depth-First**: O método `accept` navega pela árvore de expressão, chamando primeiro os `accept` dos filhos.
3. `visit(token)`: Após visitar os filhos, o método `visit` mais específico para o token atual é invocado na sua implementação do visitor.

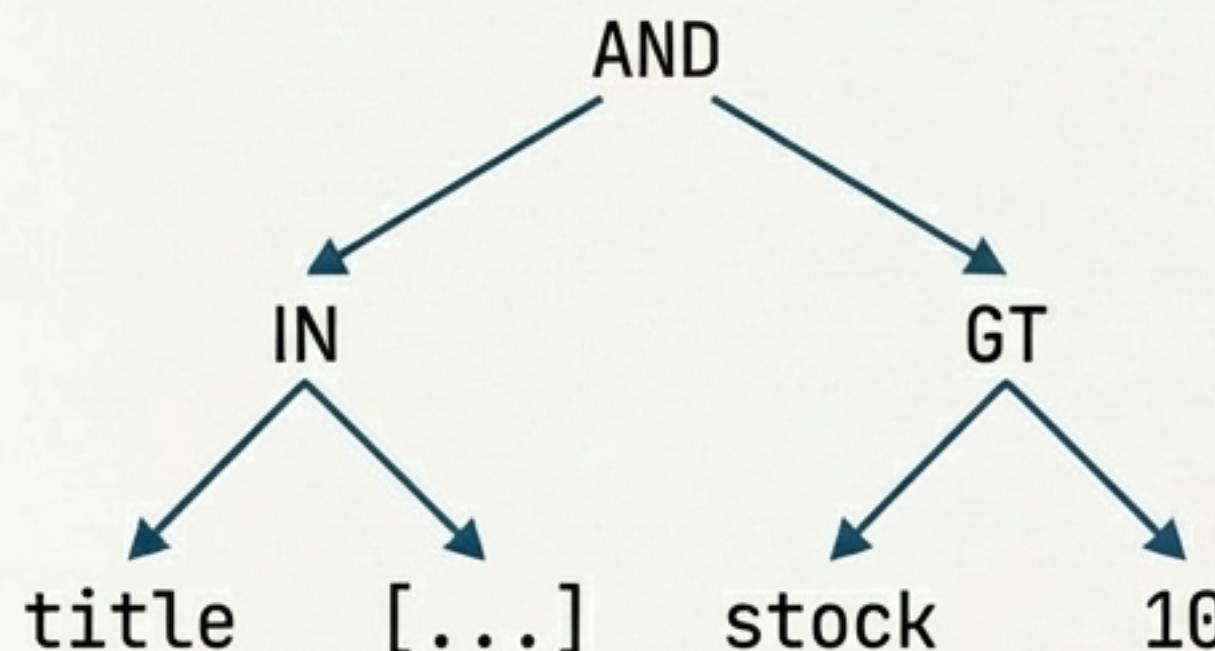


Decodificando uma Query Complexa com CqnVisitor

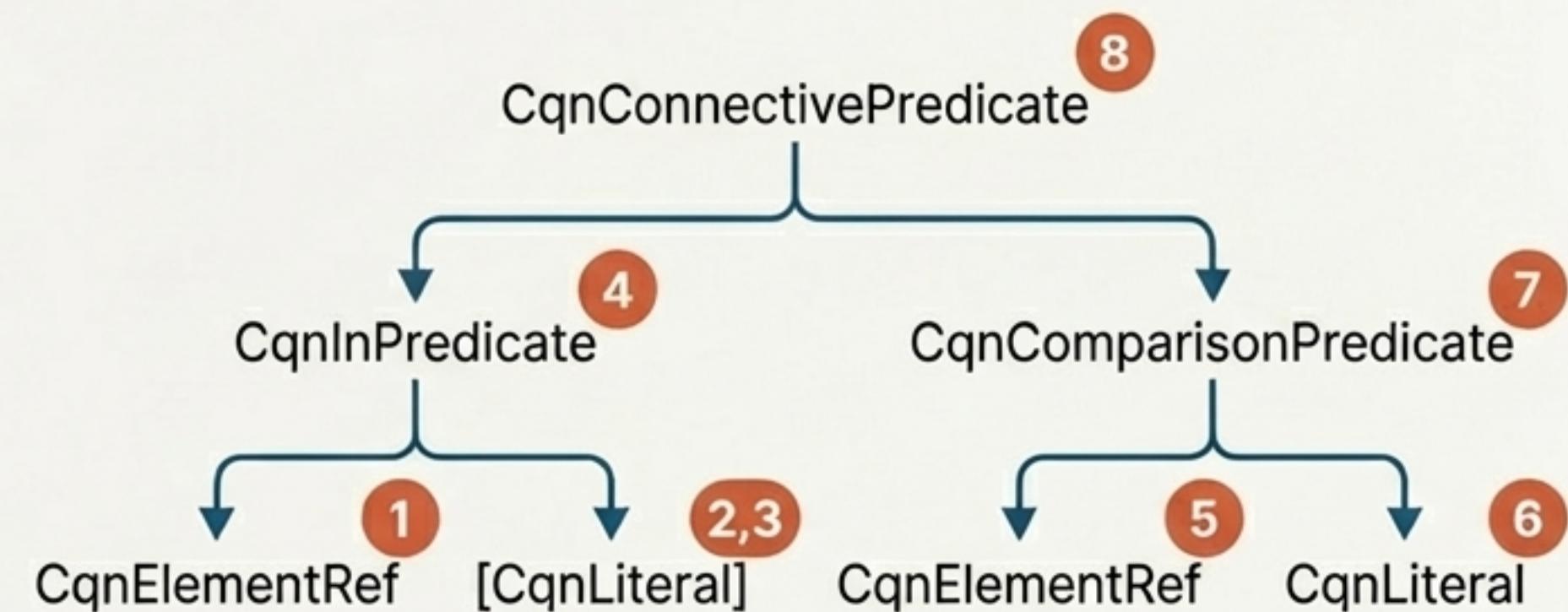
Avaliar se um conjunto de dados corresponde a um filtro complexo.

```
title IN ('Catweazle', 'The Raven') AND stock > 10
```

Árvore de Predicados (Visual)



Árvore de Tokens CQN (com ordem de visitação)



O `CqnAnalyzer` não é adequado aqui, pois nem `title` nem `stock` estão restritos a um único valor.

A Lógica do Visitor: Construindo o `CheckDataVisitor`

Estratégia: Para respeitar a ordem de travessia *depth-first*, usamos uma `Deque` (pilha) para armazenar resultados intermediários.

Passo 1: Estrutura Base e Folhas da Árvore



Handles nodes like `title` and `stock`.
Pushes the actual data value (e.g.,
'The Raven' or 42) onto the stack.

Handles nodes like `10` or
'Catweazle'. Pushes the literal
value from the query onto the stack.

```
class CheckDataVisitor implements CqnVisitor {  
    private final Map<String, Object> data;  
    private final Deque<Object> stack = new ArrayDeque<>();  
    // Construtor e método `matches()`...  
  
    // Visita uma referência de elemento (folha)  
    @Override  
    public void visit(CqnElementRef ref) {  
        // Empurra o valor do dado correspondente para a pilha  
        stack.push(data.get(ref.displayName()));  
    }  
  
    // Visita um valor literal (folha)  
    @Override  
    public void visit(CqnLiteral<?> literal) {  
        // Empurra o valor literal da query para a pilha  
        stack.push(literal.value());  
    }  
}
```

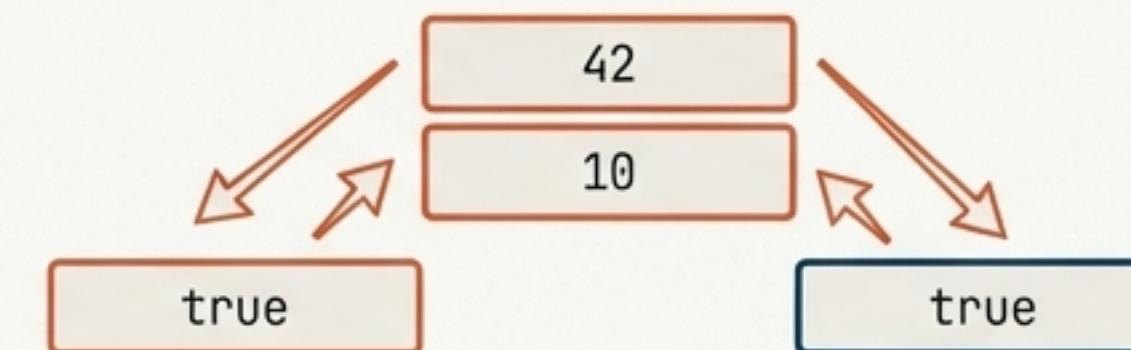
A Lógica do Visitor: Avaliando Predicados

Estratégia: Os métodos `visit` para predicados desempilham os valores (das folhas), realizam a comparação e empilham o resultado booleano.

Passo 2: Avaliando os Nós de Comparação



```
@Override  
public void visit(CqnInPredicate in) {  
    List<Object> values = ... desempilha os valores literais  
    Object value = stack.pop(); // desempilha o valor do dado  
    stack.push(values.stream().anyMatch(value::equals));  
}
```

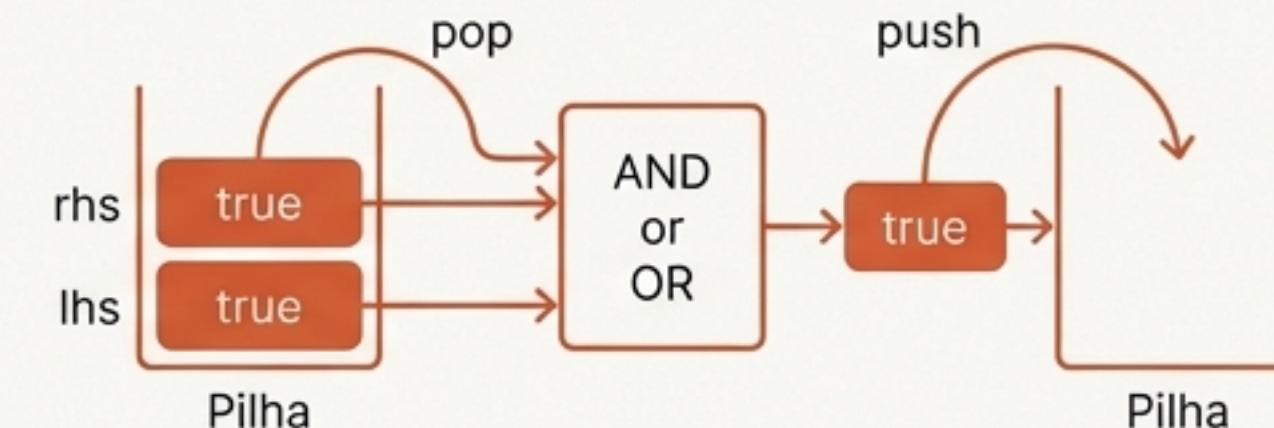


```
@Override  
public void visit(CqnComparisonPredicate comparison) {  
    Comparable rhs = (Comparable) stack.pop(); // valor literal  
    Comparable lhs = (Comparable) stack.pop(); // valor do dado  
    // ... switch para o operador (GT, EQ, etc.)  
    stack.push(cmp > 0); // empilha o resultado  
}
```

A Lógica do Visitor: Reduzindo a Árvore

Estratégia: O método `visit` para predicados conectivos (`AND`, `OR`) desempilha os resultados booleanos de seus filhos e aplica o operador lógico.

Passo 3: Reduzindo a Expressão (o Nó Raiz)



```
@Override  
public void visit(CqnConnectivePredicate connect) {  
    Boolean rhs = (Boolean) stack.pop(); // resultado da direita  
    Boolean lhs = (Boolean) stack.pop(); // resultado da esquerda  
    switch (connect.operator()) {  
        case AND: stack.push(lhs && rhs); break;  
        case OR: stack.push(lhs || rhs); break;  
    }  
}
```

Execução e Resultado:

```
// ... loop sobre os dados  
filter.accept(v); // Inicia a travessia e avaliação  
System.out.println(book.get("title") + " " + (v.matches() ? "match" : "no match"));
```

Saída:

Catweazle no match
The Raven match
Dracula no match

O Ponto de Decisão: CqnAnalyzer vs. CqnVisitor

Use **CqnAnalyzer** quando...

- Referências de elemento são mapeadas sem ambiguidade para um valor por:
- Um predicado de comparação usando eq ou is.
- Uma cláusula byId ou matching.
- Predicados são conectados apenas pela conjunção and.

****Em Resumo**:** Para extração rápida e direta de valores em queries simples e restritivas.



Use **CqnVisitor quando...**

- A query envolve comparações com lt, gt, le, gt, le, ge, ne, isNot.
- A query utiliza predicados in ou search.
- A lógica inclui negação com not.
- A query contém funções ou subqueries.

****Em Resumo**:** Para análise profunda e lógica customizada em queries de qualquer complexidade.

Introspecção como uma Operação de Redução

A introspecção de queries CQN é um processo de redução da complexidade a uma informação acionável.



- 🔎 `CqnAnalyzer`: Reduz uma query simples diretamente a um mapa de valores-chave. É uma otimização para o caso comum.
- `CqnVisitor`: Reduz uma árvore de expressão complexa, passo a passo, a um único resultado. É a ferramenta fundamental para o caso geral.

Escolha a ferramenta que melhor se alinha com a complexidade da sua query para um código mais limpo, eficiente e robusto.