

Desconstruindo o CQN

Uma Análise Arquitetural da CAP CDS Query Notation

Para desenvolvedores, arquitetos e líderes técnicos que trabalham com o SAP Cloud Application Programming Model.



Três Caminhos, Um Objeto: A Flexibilidade do CQN

1. Com Strings CQL

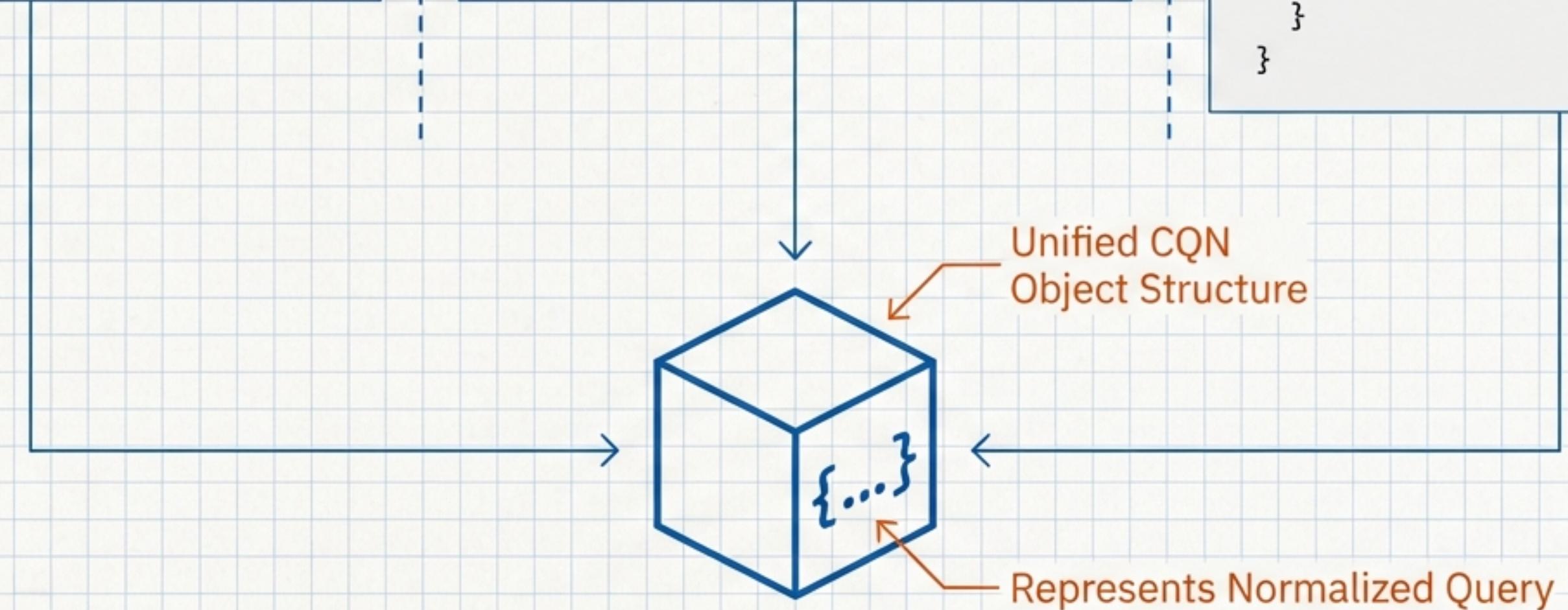
```
// Parsing CQL tagged template strings  
let query = cds ql`SELECT from Foo`
```

2. Com o Query Builder

```
// Query building  
let query = SELECT.from(ref`Foo`)
```

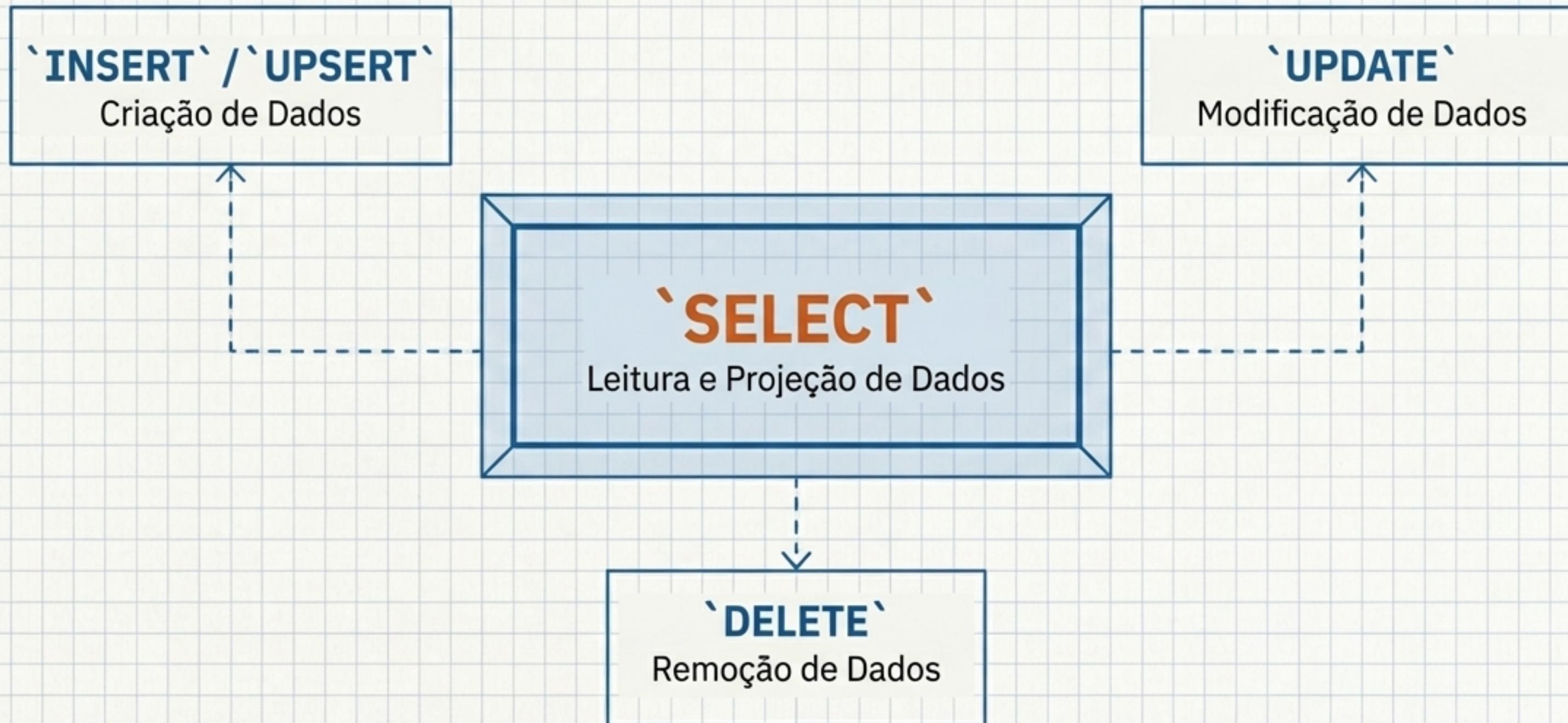
3. Com Objetos Puros

```
// Constructing plain CQN objects  
let query = {  
  SELECT: {  
    from: [{ ref: ['Foo'] }] }  
}
```



Todos os exemplos acima constroem o mesmo objeto de query, que pode ser executado com `await cds.run(query)`.

A Anatomia de uma Query: Os Quatro Verbos Principais



O CQN modela todas as interações com o banco de dados através de quatro tipos de objetos principais. Começaremos nossa análise pelo `SELECT`, o mecanismo central para a consulta de dados.

O Coração da Leitura: A Estrutura do `SELECT`

```
class SELECT { SELECT: {  
    distinct? : true  
    count? : true  
    one? : true  
    from : source  
    columns? : column[]  
    where? : xo[]  
    groupBy? : expr[]  
    orderBy? : order[]  
    having? : xo[]  
    // ...  
}}
```

Anotações

As queries 'SELECT' do CQN aprimoram o SQL com adições notáveis:

- **'one?'**: Retorna um único objeto em vez de um array.
- **'count?'**: Solicita a contagem total de registros, similar ao '\$count' do OData.
- **Projeções Aninhadas**: A cláusula 'columns' suporta projeções profundas de dados.
- **Sintaxe Minimalista**: Um 'SELECT' é válido apenas com a propriedade 'from', equivalente a 'SELECT * from ...'

De Onde Vêm os Dados? A Cláusula `from`

A propriedade `from` especifica a origem da query. Pode ser uma tabela, uma view, uma subquery ou um `JOIN` entre fontes.

A Definição do Tipo `source`

```
type source = ref & as | SELECT | {
  join: 'inner' | 'left' | 'right'
  args: [ source, source ]
  on?: expr
}
```

Exemplos Práticos

Referência Simples (`ref`)

```
SELECT.from('Books')
```



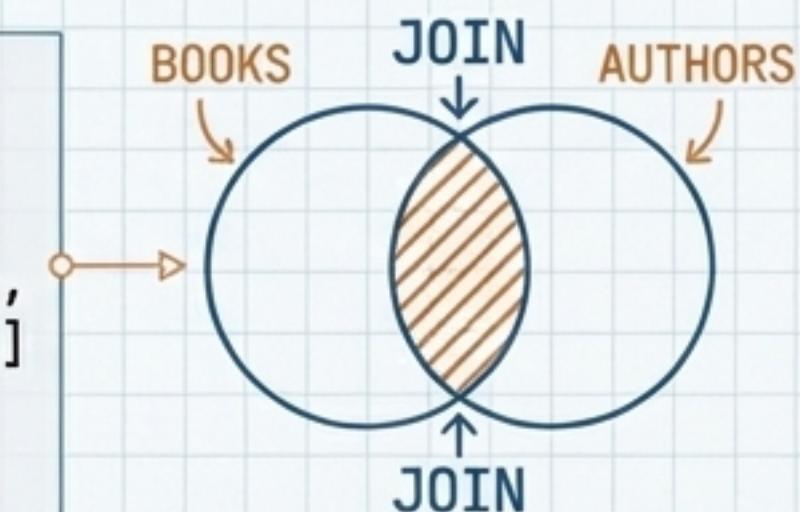
Subquery (`SELECT`)

```
SELECT.from( SELECT.from(...) )
```



Join

```
{ SELECT: { from: {
  join: 'inner',
  args: [ 'Books', 'Authors' ],
  on: [ {ref:['book','author']}, '=',
    {ref:['author','ID']} ] }
}}}
```

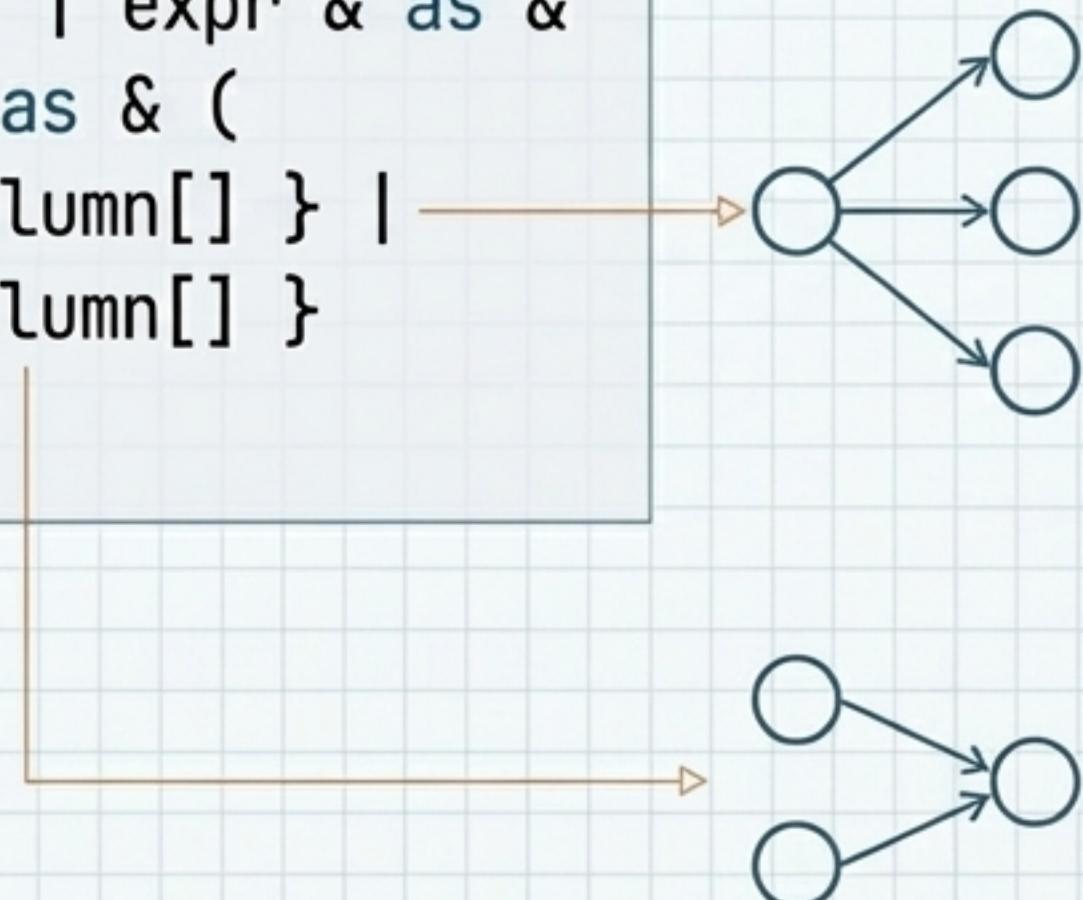


Projetando o Resultado: A Cláusula `columns`

A propriedade `columns` define as colunas a serem selecionadas. Ela vai além de uma simples lista de campos, permitindo projeções aninhadas (`expand`) e achatadas (`inline`).

A Definição do Tipo `column`

```
type column = '*' | expr & as &
  & cast | ref & as & (
    { expand?: column[] } | --
    { inline?: column[] }
  ) & infix
```

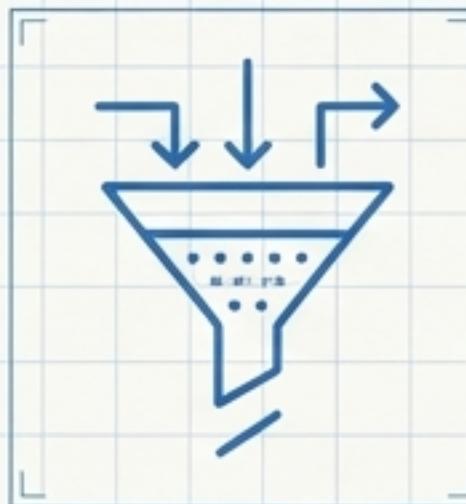


Exemplo de `expand` para buscar um autor e seus livros em uma única query:

```
SELECT.from('Authors').columns(a => {
    a.name,
    a.books(b => {
        b.title,
        b.stock
    })
})
```

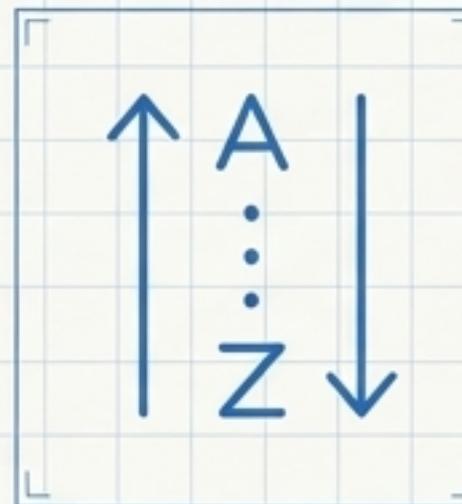
Refinando a Busca: `where`, `orderBy` e Mais

As cláusulas `where`, `having` e `search` aplicam predicados de filtro, enquanto `orderBy` define a ordem dos resultados.



`where` / `having` / `search`

Especificam predicados de filtro para as linhas selecionadas ou agrupadas. A sintaxe usa uma sequência de expressões (`xo[]`).



`orderBy`

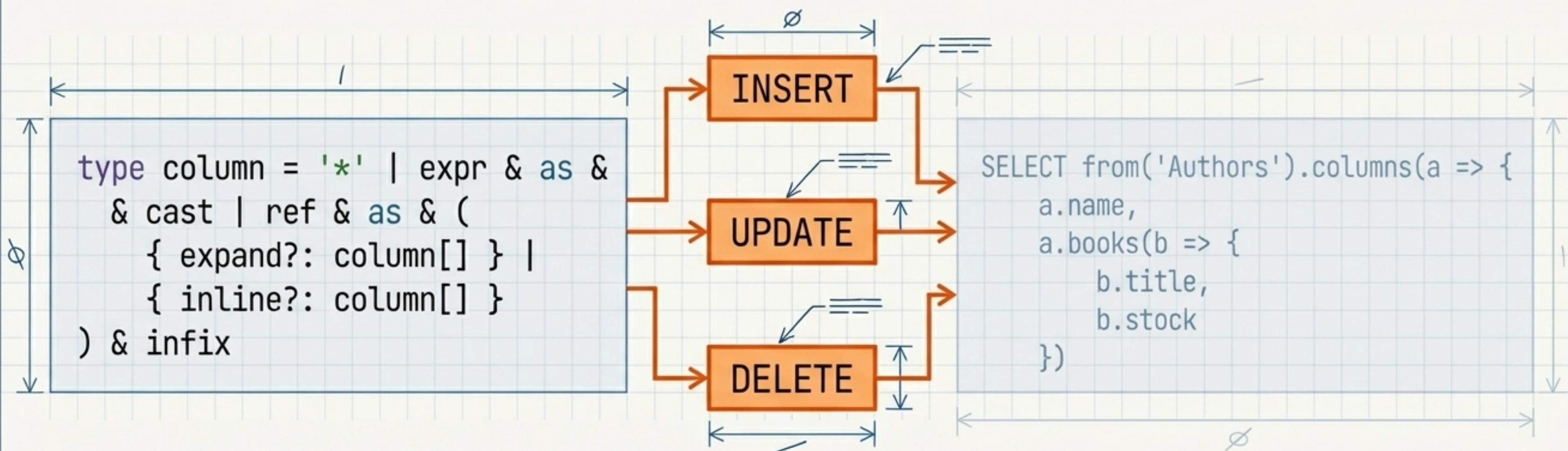
Define a ordenação. Suporta a especificação de `nulls` como `first` ou `last`.

```
type order = expr & {  
    sort: 'asc' | 'desc'  
    nulls: 'first' | 'last'  
}
```

Exemplo de Combinação

```
SELECT.from('Books').columns('title', 'stock')  
.where({ stock: { '>': 100 } })  
.orderBy({ stock: 'desc' })
```

Modificando Dados: A Arquitetura de `INSERT`, `UPDATE` e `DELETE`



Após explorar a leitura de dados, vamos analisar os mecanismos do CQN para criar, atualizar e remover registros. Essas operações compartilham uma estrutura elegante e poderosa, focada na manipulação de dados.

Criando Registros: A Versatilidade do `INSERT` e `UPsert`

As representações de `INSERT` e `UPsert` no CQN são essencialmente idênticas. A diferença está na semântica: `UPsert` atualiza o registro se ele já existir.

1. Com `entries` (Deep Inserts)

Permite a inserção de registros como objetos, incluindo associações aninhadas.

```
{ INSERT: { into: 'Authors', entries: [
  { ID:150, name:'Edgar Allen Poe', books: [
    { ID:251, title:'The Raven' },
    { ID:252, title:'Eleonora' }
  ]}
]} }
```

Deep Insert: Associações aninhadas incluídas

2. Com `columns` e `values`

Para inserir uma única linha, como no SQL padrão.

```
{ INSERT: { into: 'Books',
  columns: [ 'ID', 'title' ],
  values: [ 201, 'Wuthering Heights' ]
}}
```

Para inserir uma única linha, como no SQL

3. Com `columns` e `rows`

Para inserir múltiplas linhas de forma eficiente.

```
{ INSERT: { into: 'Books',
  columns: [ 'ID', 'title' ],
  rows: [ [ 201, 'Wuthering Heights' ], [ 252, 'Eleonora' ] ]
}}
```

Atualizando Dados: A Diferença Crucial entre `data` e `with`

UPDATE ... data

Para atualizações com valores literais (`scalar`). A estrutura é idêntica à de `INSERT.entries`.

```
interface data { [element:name]: scalar | data  
  | data[]  
}
```

```
UPDATE('Books').where({ ID: 201 }).with({  
  data: { title: 'New Title' }  
})
```

UPDATE ... with

Mais poderoso. Permite o uso de expressões (`expr`) como valores, possibilitando, por exemplo, operações aritméticas ou referências a outras colunas.

```
interface changes { [element:name]:  
  scalar | expr | changes | ...  
}
```

```
UPDATE('Books').where({ ID: 201 }).with({  
  with: { stock: { xpr: ['stock', '+', 1] } }  
})
```

Use `data` para dados puros. Use `with` quando precisar de expressões do banco de dados.

Removendo Dados: A Simplicidade do `DELETE`

‘DELETE’ é a operação mais direta. Ela especifica o alvo (`from`) e, opcionalmente, uma condição (`where`) para determinar quais registros devem ser removidos.

Inter, Regular

A Definição da Classe ‘DELETE’

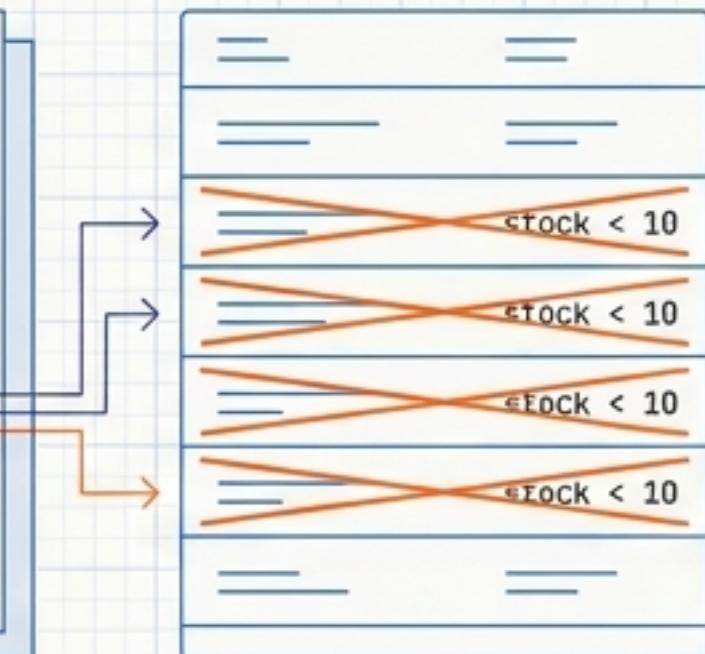
IBM Plex Sans, Medium

```
JetBrains Mono  
  
class DELETE { DELETE: {  
  from: ref  
  where?: expr  
}}  
  
;
```

Exemplo de Uso

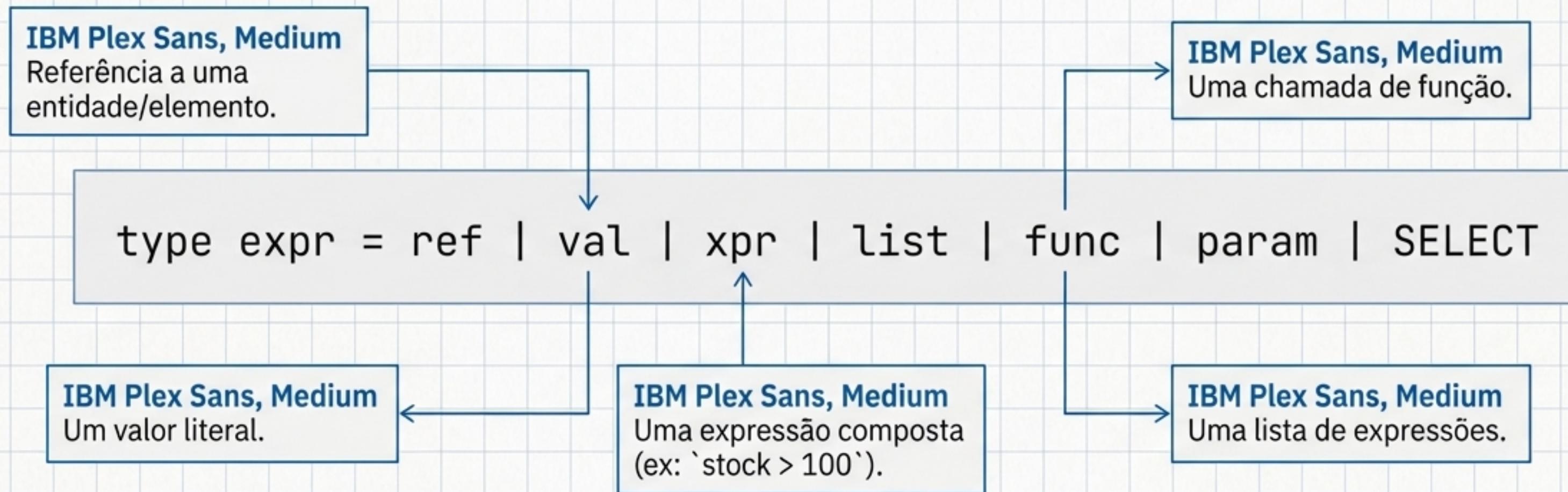
IBM Plex Sans, Medium

```
JetBrains Mono  
  
const q = {  
  DELETE: {  
    from: { ref: ['Books'] },  
    where: [ { ref: ['stock'] }, '<', { val: 10 } ]  
  }  
};  
  
;
```



Os Blocos Fundamentais: A Linguagem Universal das Expressões (`expr`)

Todas as partes de uma query CQN — de referências a colunas, valores literais, funções e condições — são construídas a partir de um conjunto fundamental de tipos de expressão.



“

O CQN, por intenção, não ‘entende’ expressões; palavras-chave e operadores são representados como strings. Isso nos permite traduzir para e de qualquer outra linguagem de query, incluindo suporte a recursos nativos de SQL.

”

Referência Rápida: Primitivos e Tipos Essenciais do CQN

Tipos de Expressão (`expr`)

```
type expr = ref | val | xpr | list | ...  
  
type ref = { ref: name[] } ← Estrutura de Referência  
  
type val = { val: scalar } ← Valor Escalar Embutido  
  
type func = { func: string, args: expr[] }
```

Expressões Compostas (`xo`)

```
type xo = expr | keyword | operator  
  
type xpr = { xpr: xo[] } ← Sequência de Operandos/Operadores  
  
type operator = '=' | '!=' | '<' | '>' | ...  
type keyword = 'in' | 'like' | 'and' | 'or' | ... ← Strings de Operação e Lógica
```

Tipos Escalares e de Dados

```
type scalar = number | string | boolean | null ← Tipos Primitivos Básicos  
  
type name = string  
  
interface data { ... } ← Interfaces de Estrutura de Dados  
interface changes { ... }
```

Para a especificação completa, consulte o arquivo `cqn.d.ts` na documentação oficial do CAP.

