

O Playbook de Performance para ABAP Moderno

Dominando Tabelas Internas, Expressões e Chaves Secundárias para a Era S/4HANA



O Desafio: De Milhares para Milhões de Registros

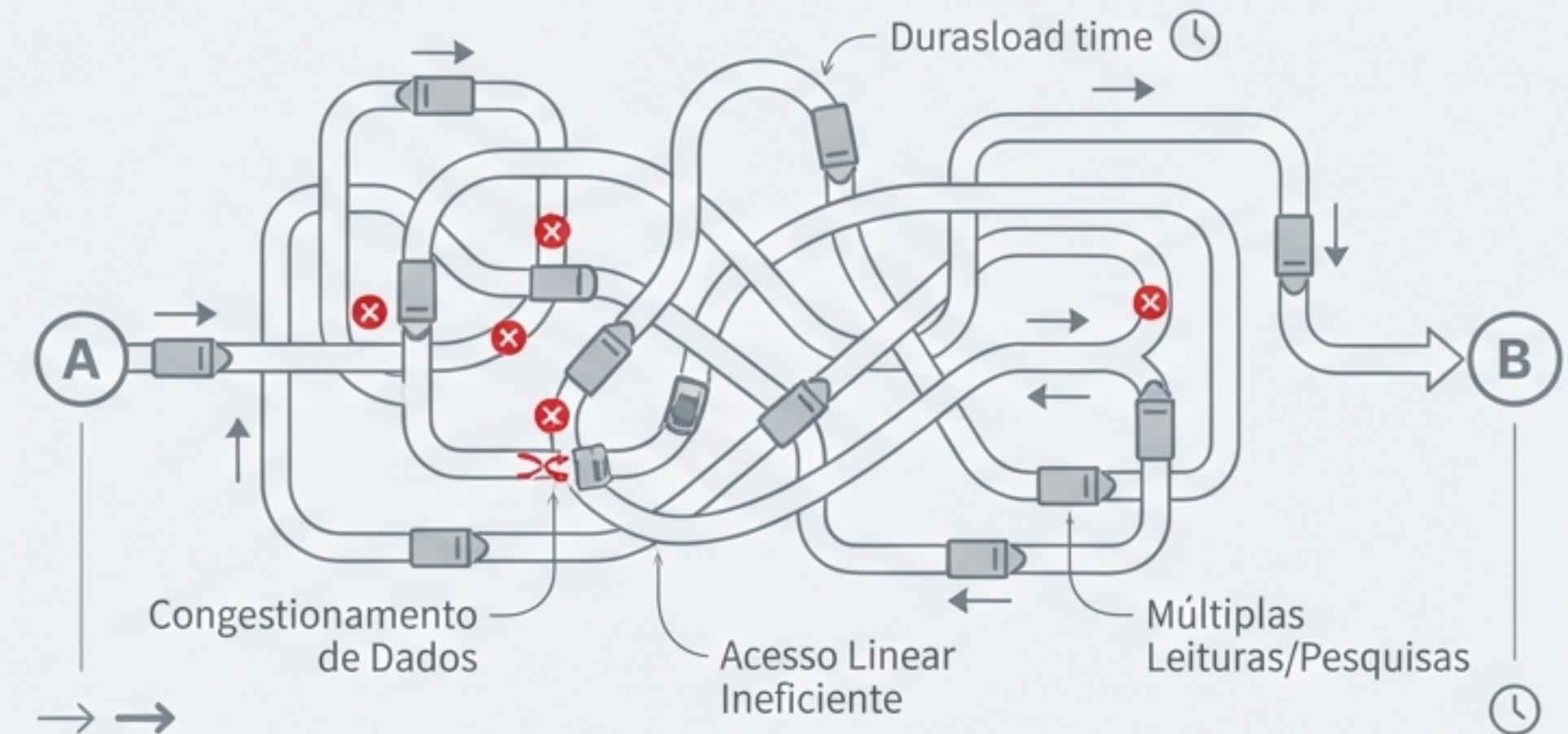
No S/4HANA, o volume de dados não apenas cresceu, ele explodiu.

Abordagens legadas que funcionavam para 10.000 registros resultam em **TIME_OUT** com 10 milhões.

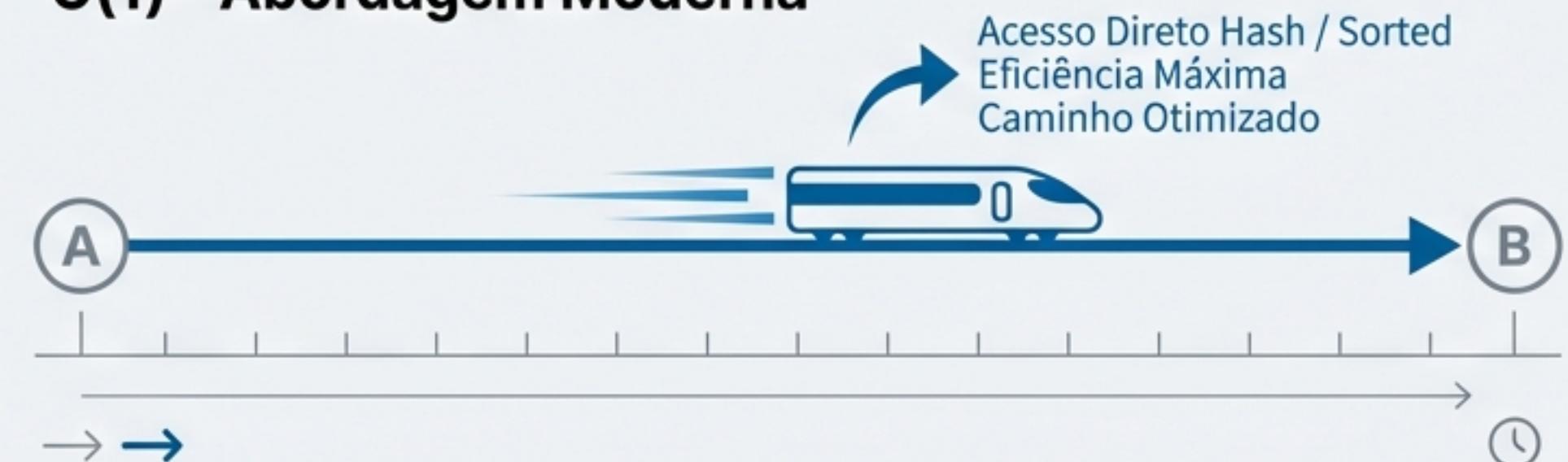
A sua maior alavanca de performance não está no hardware, mas na estrutura de dados que você escolhe.

A escolha do tipo de tabela interna pode significar a diferença entre um programa que executa em 5 segundos e um que falha após 10 minutos.

$O(n)$ - Abordagem Legada



$O(1)$ - Abordagem Moderna



A Escolha Fundamental: Suas Ferramentas de Acesso à Memória

Cada tipo de tabela interna oferece uma troca entre flexibilidade e performance. Conhecer a complexidade algorítmica (Big O Notation) de cada uma é a base para escrever código de alta performance.



Standard Table

O(n) - Tempo Linear

O tempo de busca cresce com o tamanho da tabela.



Sorted Table

O(log n) - Tempo Logarítmico

Incrivelmente rápido, mesmo com milhões de linhas.



Hashed Table

O(1) - Tempo Constante

Acesso instantâneo, independente do tamanho da tabela.

A Tabela Standard: Flexível, Comum, Perigosa

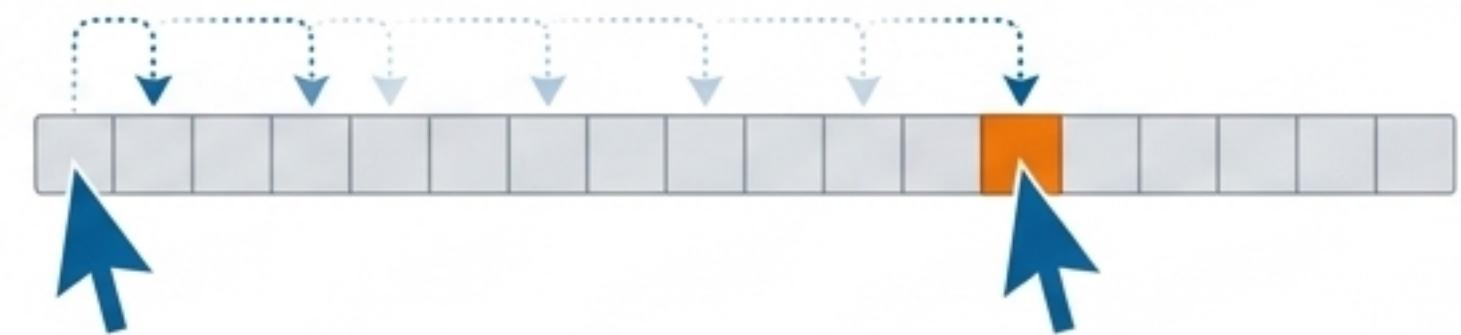
Como Funciona:

- **Estrutura:** Um array simples. A ordem é a da inserção (`APPEND`).
- **Acesso por Chave:** Busca Linear. O sistema varre a tabela do início ao fim.
- **Complexidade: $O(n)$.** Dobrar os dados significa dobrar o tempo de busca.

Quando Usar (e Quando Evitar):

- **✓ Ideal para:** Listas pequenas (<100 linhas), buffers temporários, ou quando o acesso é sempre feito pelo índice da linha (`READ TABLE ... INDEX ...`).
- **✗ Evitar para:** Buscas por chave em grandes volumes de dados. É a principal causa de gargalos de performance.

```
TYPES ... TYPE STANDARD TABLE OF ...
```



A Tabela Sorted: Ordem e Velocidade para Leituras



Como Funciona

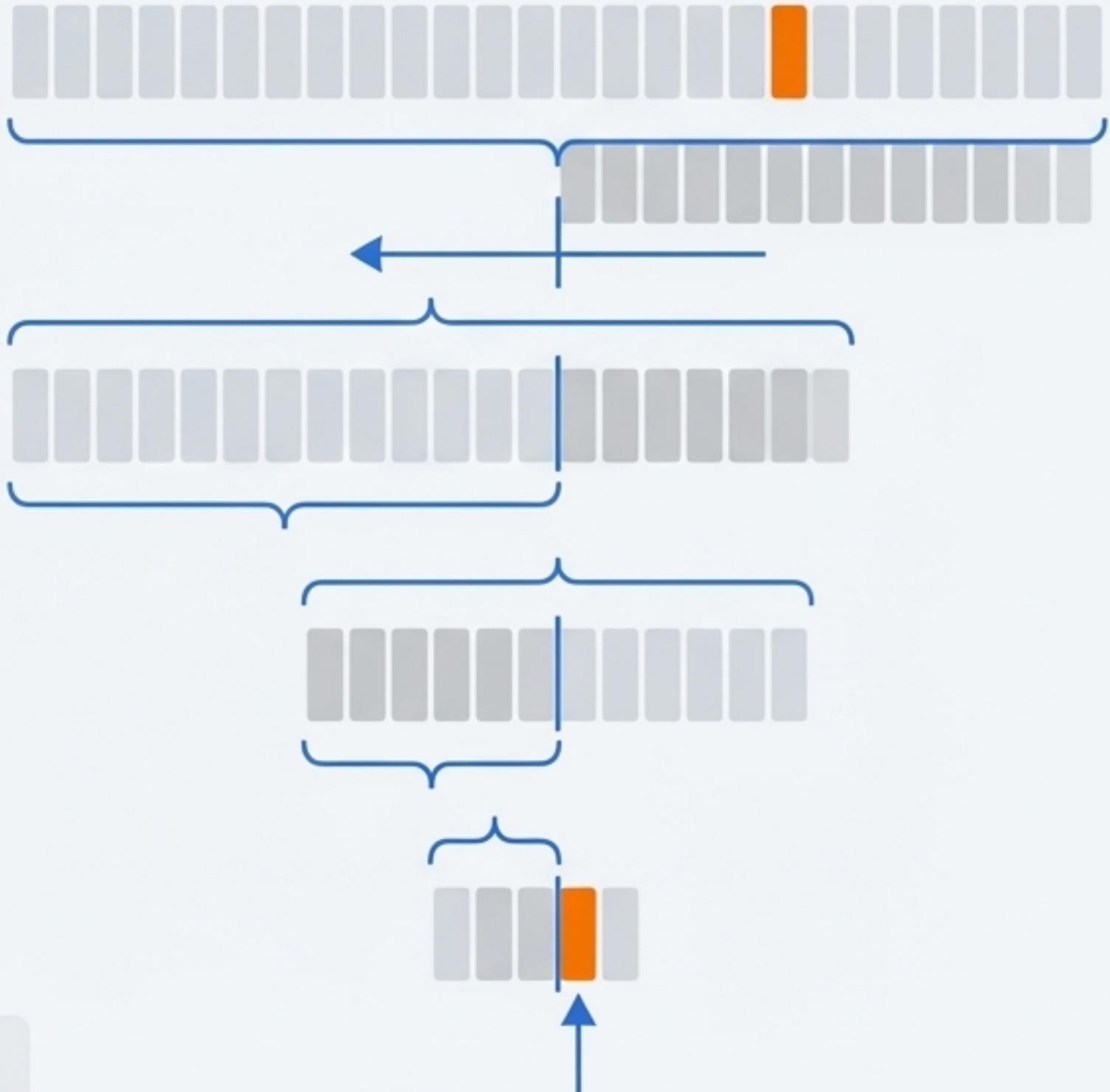
- **Estrutura:** Mantém um índice interno ordenado. Os dados são organizados automaticamente na inserção.
 - **Acesso por Chave:** Busca Binária. Divide a busca pela metade a cada passo.
 - **Complexidade: O(log n).** Extremamente performática para leituras.

O "Custo" da Ordem

A inserção (INSERT) é mais lenta que na tabela Standard, pois o sistema precisa recalcular e manter a ordenação.

Quando Usar

-  **Ideal para:** Tabelas com muitas leituras e poucas escritas, e quando a ordem dos dados é crucial (ex: FOR ALL ENTRIES).



TYPES ... TYPE SORTED TABLE OF ... WITH UNIQUE/NON-UNIQUE KEY ...

A Tabela Hashed: Acesso Imediato em Escala Massiva



Como Funciona:

- **Estrutura:** Usa um algoritmo de Hash para calcular o endereço de memória exato do registro a partir de sua chave.
- **Acesso por Chave:** Acesso Direto.
- **Complexidade: O(1)** - Tempo Constante. O tempo de acesso é o mesmo para 10 ou 10 milhões de linhas.

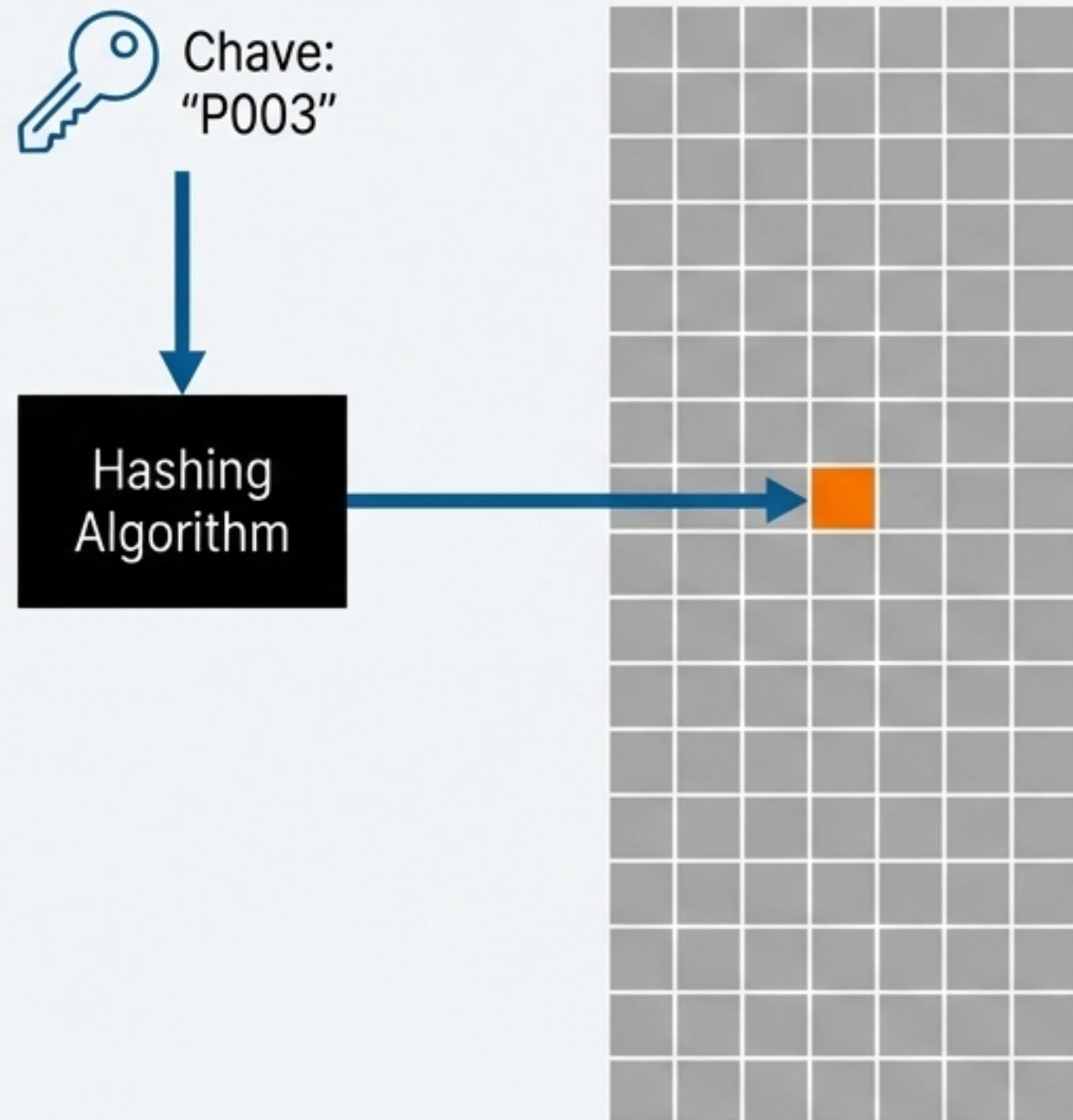
Restrições Essenciais:

- A chave **DEVE** ser **UNIQUE**.
- Não permite acesso por índice (ex: READ ... INDEX 1), pois não há uma ordem sequencial garantida.

Quando Usar:

- **Ideal para:** Caches de dados, armazenamento de dados mestre (clientes, materiais) onde o acesso é sempre feito por um ID único.

```
TYPES ... TYPE HASHED TABLE OF ... WITH UNIQUE KEY ...
```



Da Estrutura à Sintaxe: Escrevendo Código Moderno, Limpo e Seguro

Escolher a tabela certa é metade da batalha. A outra metade é usar a sintaxe que extrai o máximo de performance e legibilidade do seu código.

Vamos abandonar:

- Verificações manuais de `sy-subrc`.
- Variáveis de trabalho (`work areas`) desnecessárias.
- Loops ineficientes.

Vamos adotar:

- Expressões de Tabela (`**itab**[...]`).
- Tratamento de exceções robusto.
- Construção de tabelas em linha com `**VALUE**`.

Lendo Linhas: De `READ TABLE` para Expressões Diretas

A Abordagem Clássica

```
READ TABLE lt_flights INTO ls_flight  
  WITH KEY carrier_id = 'LH'.  
IF sy-subrc = 0.  
  " Processar ls_flight...  
ENDIF.
```

Requer uma work area ('ls_flight').

Verificação manual do 'sy-subrc'.

Código mais verboso.

A Expressão de Tabela

```
TRY.  
  DATA(ls_flight) = lt_flights[ carrier_id = 'LH' ].  
  " Processar ls_flight...  
CATCH cx_sy_itab_line_not_found.  
  " Tratar erro de forma explícita.  
ENDTRY.
```

Declaração 'INLINE' da variável.

Resultado direto ou exceção.

Código limpo e seguro ('fail-fast').

O Poder do Encadeamento (Chaining): Acesse Dados Profundos Diretamente

O Problema Legado: Para ler um único campo, você precisava ler a linha inteira para uma work area e só então acessar o campo.

A Solução Moderna: Expressões de tabela permitem encadear o acesso, indo direto ao ponto sem variáveis intermediárias.

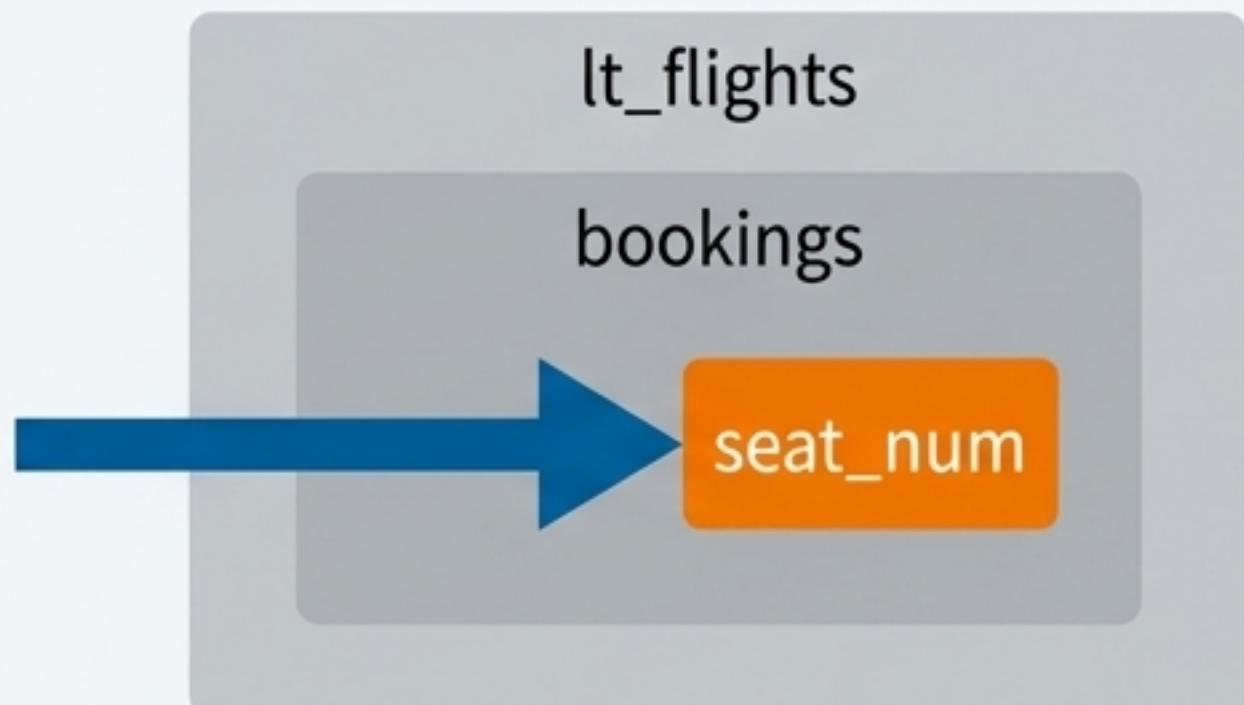
Exemplos Práticos:

1. Acessar um campo específico:

```
" Obter o preço do produto 'P001' em uma única linha.  
DATA(lv_price) = lt_products[ id = 'P001' ]-price.
```

2. Navegar em estruturas aninhadas (deep structures):

```
" Obter o número do assento de um cliente específico em um voo.  
DATA(lv_seat) = lt_flights[ 1 ]-bookings[ customer = 'SAP' ]-seat_num.
```



Verificações Inteligentes: `line_exists` e `line_index`

Substitua `READ TABLE`'s verbosos por funções predicativas claras e com propósito definido.

Para Verificar se uma Linha Existe:

Legado:

```
READ TABLE lt_tab WITH KEY ... TRANSPORTING NO FIELDS.
```

Moderno:

```
IF line_exists( lt_tab[ key = val ] ). ✓
```

→ **Benefício:** A intenção do código é explícita. Você está perguntando "existe?", não "leia nada".

Para Obter o Índice de uma Linha:

Legado:

```
READ TABLE lt_tab WITH KEY ... e depois usar sy-tabix.
```

Moderno:

```
DATA(lv_idx) = line_index( lt_tab[ key = val ] ). ✓
```

→ **Benefício:** Retorna o índice diretamente ou 0 se não encontrar. Sem dependência de variáveis de sistema ('sy-').

Construindo Tabelas Instantaneamente com o Operador `VALUE`

Chega de múltiplos `APPEND`s para inicializar uma tabela. O construtor `VALUE` permite criar e popular tabelas de forma declarativa e em um único passo.

Exemplo: Criando uma Tabela do Zero:

```
" Inicializa a tabela de moedas com dois registros.  
DATA(lt_currencies) = VALUE ty_currencies_tab(  
    ( code = 'EUR' name = 'Euro' )  
    ( code = 'USD' name = 'Dólar' )  
).
```

Exemplo: Adicionando a uma Tabela Existente com `BASE`:

Importante: Sem `BASE`, `VALUE` sobrescreve o conteúdo. Com `BASE`, ele anexa.

```
" Adiciona o Real brasileiro à tabela existente.  
lt_currencies = VALUE #(  
    BASE lt_currencies  
    ( code = 'BRL' name = 'Real' )  
).
```

Initial State:

code	name



After VALUE:

code	name
EUR	Euro
USD	Dólar

After BASE:

code	name
EUR	Euro
USD	Dólar
BRL	Real

O Masterclass: Combinando Hashed Tables e Chaves Secundárias

O Cenário:

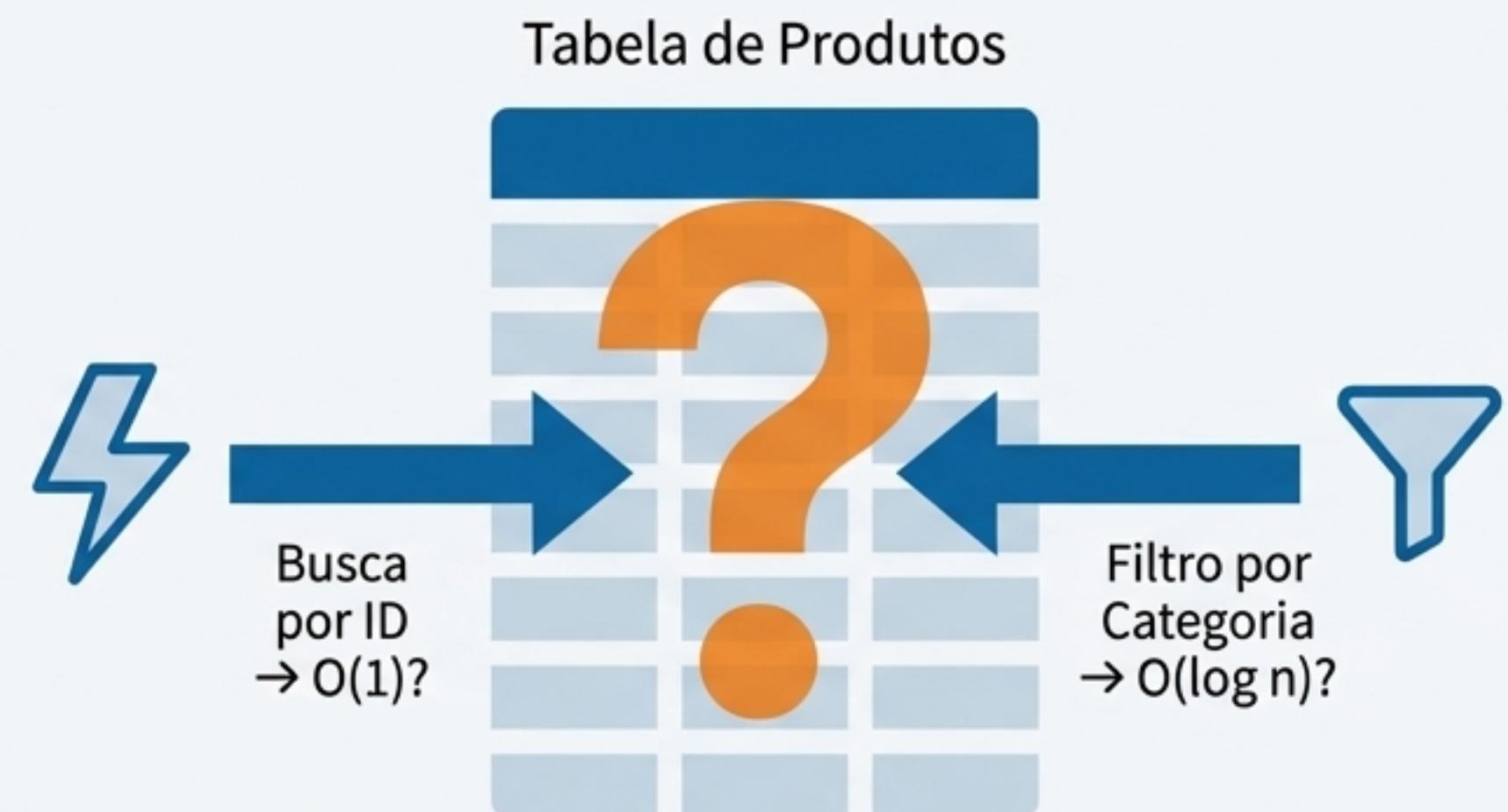
Temos uma tabela gigante de produtos. Nossas duas operações mais críticas são:

1. **Busca por ID de Produto:** Precisa ser instantânea. Milhares de vezes por segundo. (Candidato perfeito para uma Tabela Hashed).
2. **Filtrar por Categoria:** Precisamos de todos os produtos de uma categoria ('Hardware', 'Software'). Uma busca linear ($O(n)$) em uma tabela Hashed seria um desastre de performance. LOOP AT ... WHERE

A Pergunta:

Como podemos ter acesso $O(1)$ pelo `id` E, ao mesmo tempo, um acesso otimizado $O(\log n)$ pela `category`?

Uma Tabela Hashed com uma Chave Secundária Ordenada.

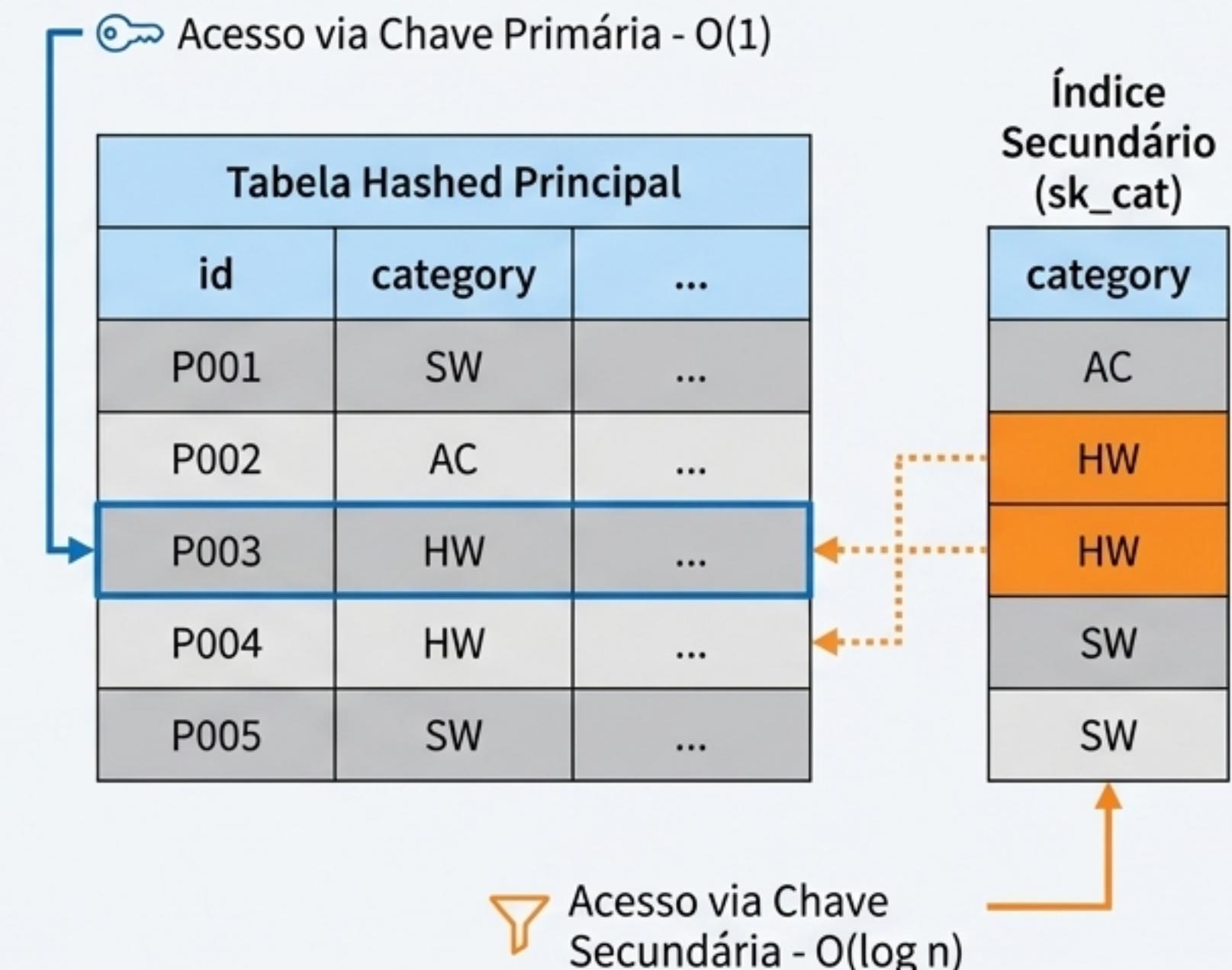


A Anatomia da Solução: Duas 'Vias Expressas' para Seus Dados

A Chave Secundária cria um segundo índice otimizado dentro da sua tabela, permitindo alta performance em mais de um critério de busca.

A Declaração

```
TYPES ty_prod_hashed_table TYPE HASHED TABLE OF ty_product
WITH UNIQUE KEY id "A 'via expressa' principal (O(1))"
WITH NON-UNIQUE SORTED KEY sk_cat "A 'via expressa' secundária (O(log n))"
COMPONENTS category.
```



O Código em Ação: Performance e Legibilidade

1. Populando com VALUE:

```
DATA(lt_products) = VALUE ty_prod_hashed_table(  
  ( id = 'P001' category = 'HW' ... )  
  ( id = 'P002' category = 'AC' ... )  
  ...  
).
```



2. Acesso Primário (Hash - O(1)):

Busca instantânea pelo ID único.

```
IF line_exists( lt_products[ id = 'P002' ] ). O(1)
```

```
IF line_exists( lt_products[ id = 'P002' ] ) ENDIF.
```

```
lt_products[ id = 'P003' ]-price = 1100.
```

3. Acesso Secundário (Sorted - O(log n)):

Loop otimizado pela chave secundária. Evita o scan completo da tabela.

ABAP Source Sans Pro

```
LOOP AT lt_products INTO DATA(ls_prod) O(log n)  
  USING KEY sk_cat WHERE category = 'HW'.  
  ...  
ENDLOOP.
```

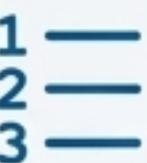
4. Modificação Direta:

Atualiza o preço usando a chave primária de forma limpa.

```
lt_products[ id = 'P003' ]-price = 1100.
```



Seu Playbook de Performance: Legado vs. Moderno

Conceito	Abordagem Legada	ABAP Moderno e Performático	Impacto
	READ TABLE tab WITH KEY... INTO wa.	wa = tab[k = v].	Sintaxe limpa, direta, à prova de falhas.
	READ ... TRANSPORTING NO FIELDS.	IF line_exists(tab[...]).	Código mais legível e com intenção clara.
	READ TABLE ...` então sy-tabix	idx = line_index(tab[...]).	Acesso direto sem dependência de `sy-fields.
	APPEND wa TO tab.` / INSERT ...	tab = VALUE #(BASE tab(...)).	Construção em massa eficiente e declarativa.
	LOOP WHERE` em Tabela Standard	LOOP ... USING KEY` Secundária	Transforma O(n) em O(log n). A maior otimização.

Glossário Essencial para o Desenvolvedor Moderno



Standard Table

Acesso por chave não otimizado (Busca Linear - $O(n)$). Use para acesso por índice ou listas pequenas.



Sorted Table

Sempre ordenada pela chave (Busca Binária - $O(\log n)$). Ideal para muitas leituras e poucas escritas.



Hashed Table

Acesso por chave única otimizado (Hash - $O(1)$). A escolha para grandes volumes de dados únicos.



Secondary Key

Um índice adicional que permite buscas rápidas por campos que não estão na chave primária.



Table Expression

Sintaxe moderna (`itab[...]`) para leitura de registros que lança exceção (`cx_sy_itab_line_not_found`) em caso de falha.



Big O Notation

Notação que descreve a eficiência de um algoritmo. $O(1)$ (constante) é o ideal; $O(n)$ (linear) é um alerta de performance.

Teste Sua Maestria: Quiz Rápido



Qual tipo de tabela interna garante tempo de acesso constante ($O(1)$), independentemente do número de registros, mas exige uma chave única?



Hashed Table. Ela utiliza um algoritmo de hash para calcular a posição exata do registro na memória.



O que acontece se eu tentar executar `DATA(ls_line) = lt_tabela[id = '999']` e o registro não existir?



O sistema lançará uma exceção da classe `cx_sy_itab_line_not_found`. Se não for tratada com `TRY...CATCH`, causará um Short Dump.

Teste Sua Maestria: Quiz Rápido (Parte 2)

-  Tenho uma tabela Standard gigante de 'Vendas' e preciso filtrar frequentemente por 'Data', que não é a chave primária. Como otimizar essa busca sem mudar o tipo da tabela para Sorted ou Hashed?
-  Definindo uma **Chave Secundária Ordenada** (`WITH NON-UNIQUE SORTED KEY sk_data COMPONENTS data`) e utilizando `USING KEY sk_data` nas leituras e loops.
-  Qual é o 'trade-off' (o custo) de usar uma Tabela Sorted em comparação com uma Standard?
-  A operação de **inserção** (`INSERT`) é **mais lenta** na Tabela Sorted, pois o sistema precisa encontrar a posição correta para manter a ordenação a cada nova entrada.