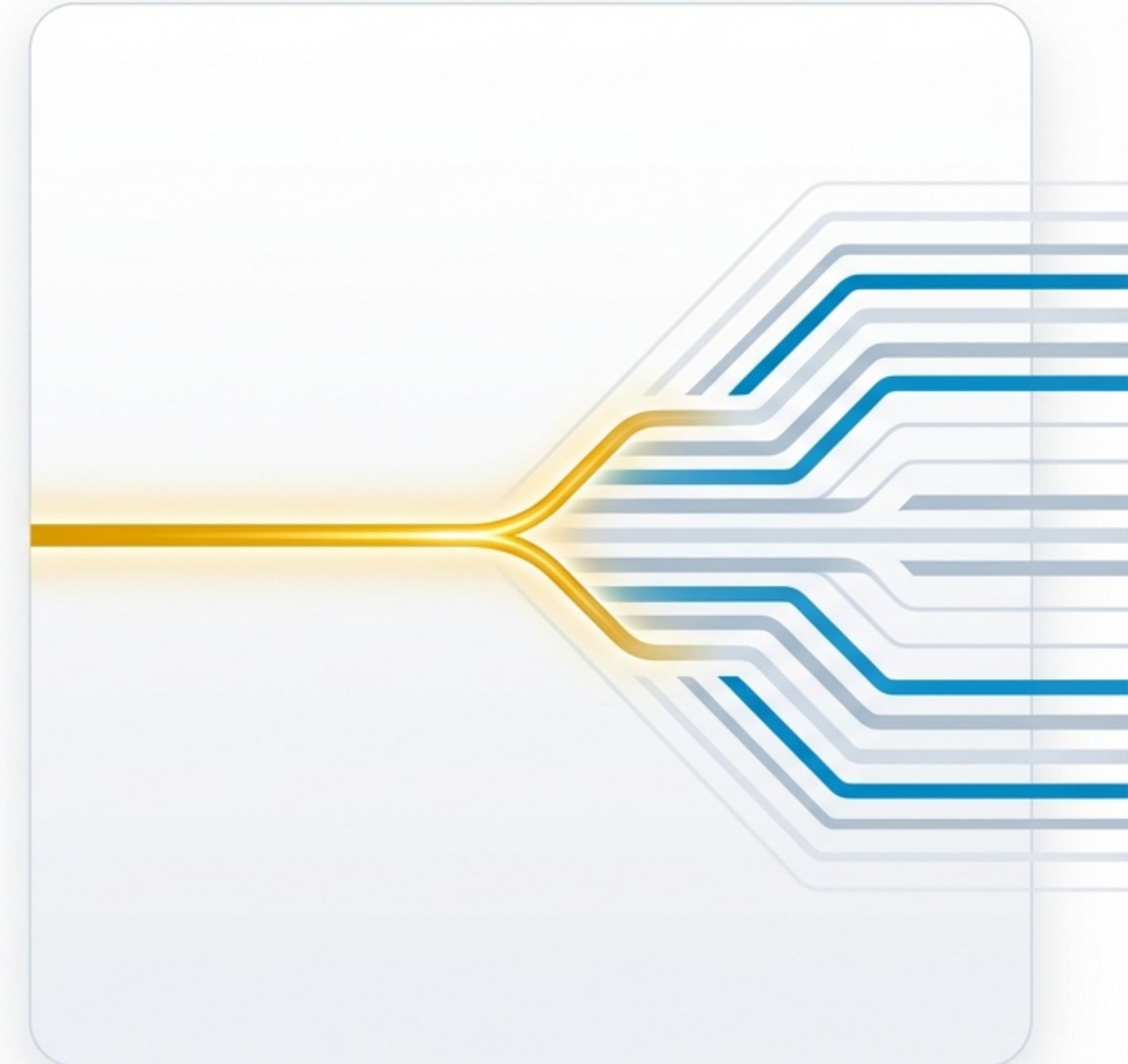


# **ABAP Moderno: Performance de Elite em Tabelas Internas**

De  $O(n)$  a  $O(1)$ : Dominando  
Chaves Secundárias e  
Gerenciamento de Memória



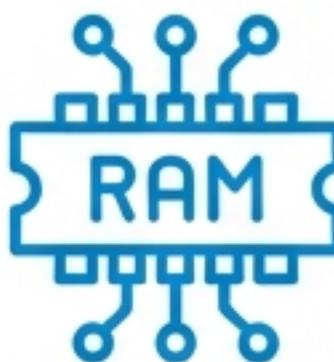
# Objetivos de Aprendizagem



Compreender a **Complexidade Algorítmica** (Notação Big O) aplicada a buscas em tabelas internas ABAP.



Implementar e consumir **Chaves Secundárias** (SECONDARY KEYS) para acelerar buscas em tabelas Standard.



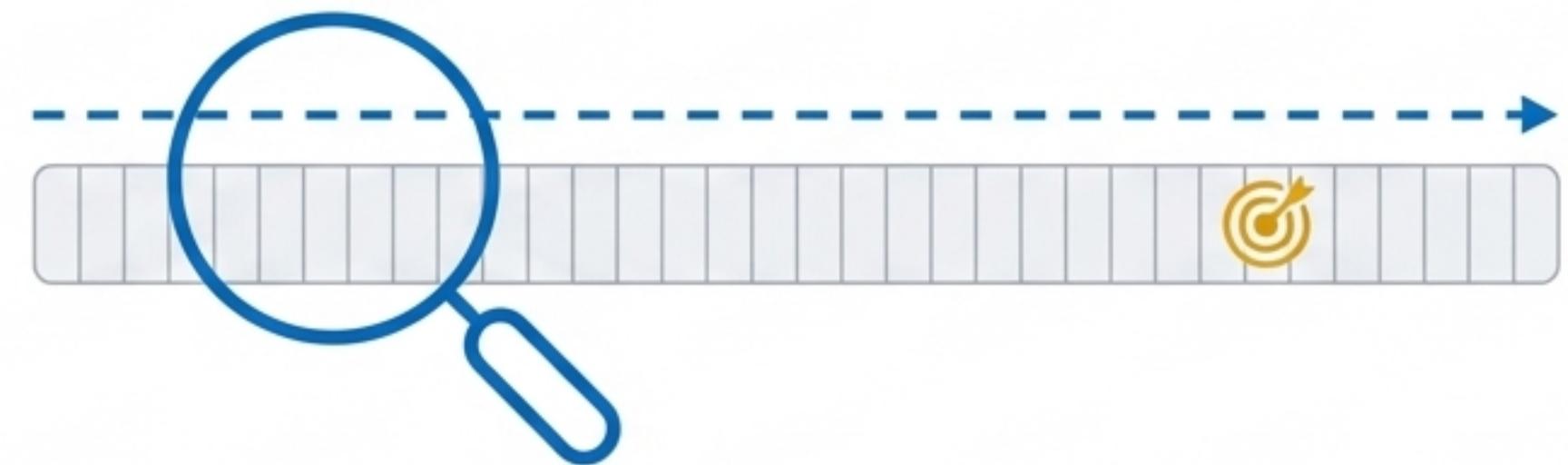
Gerenciar a memória eficientemente com **Field-Symbols** (ASSIGNING) e **Referências de Dados** (REFERENCE INTO).



Refatorar o antipadrão de **Nested Loops** transformando-os em operações de alta performance.

# O Problema: O Custo Oculto do “Table Scan”

- Uma Tabela Standard é como uma lista de compras. Para encontrar um item, você precisa ler um por um, do início ao fim.
- **Complexidade:**  $O(n)$  - O tempo de busca cresce linearmente com o número de registros ( $n$ ).
- **Impacto:** Em desenvolvimento (100 linhas) é instantâneo. Em produção (1.000.000 de linhas) causa **Timeouts** catastróficos.



# O Antagonista Clássico: Nested Loops

## Pseudo-código & Análise

### Cenário Ingênuo (Quadrático):

```
LOOP AT Tabela A (10.000 linhas).  
    READ TABLE Tabela B (5.000 linhas) ...  
ENDLOOP.
```

### O Desastre Matemático:

$10.000 \times 5.000 = \textbf{50.000.000}$   
de comparações.

### Resultado:

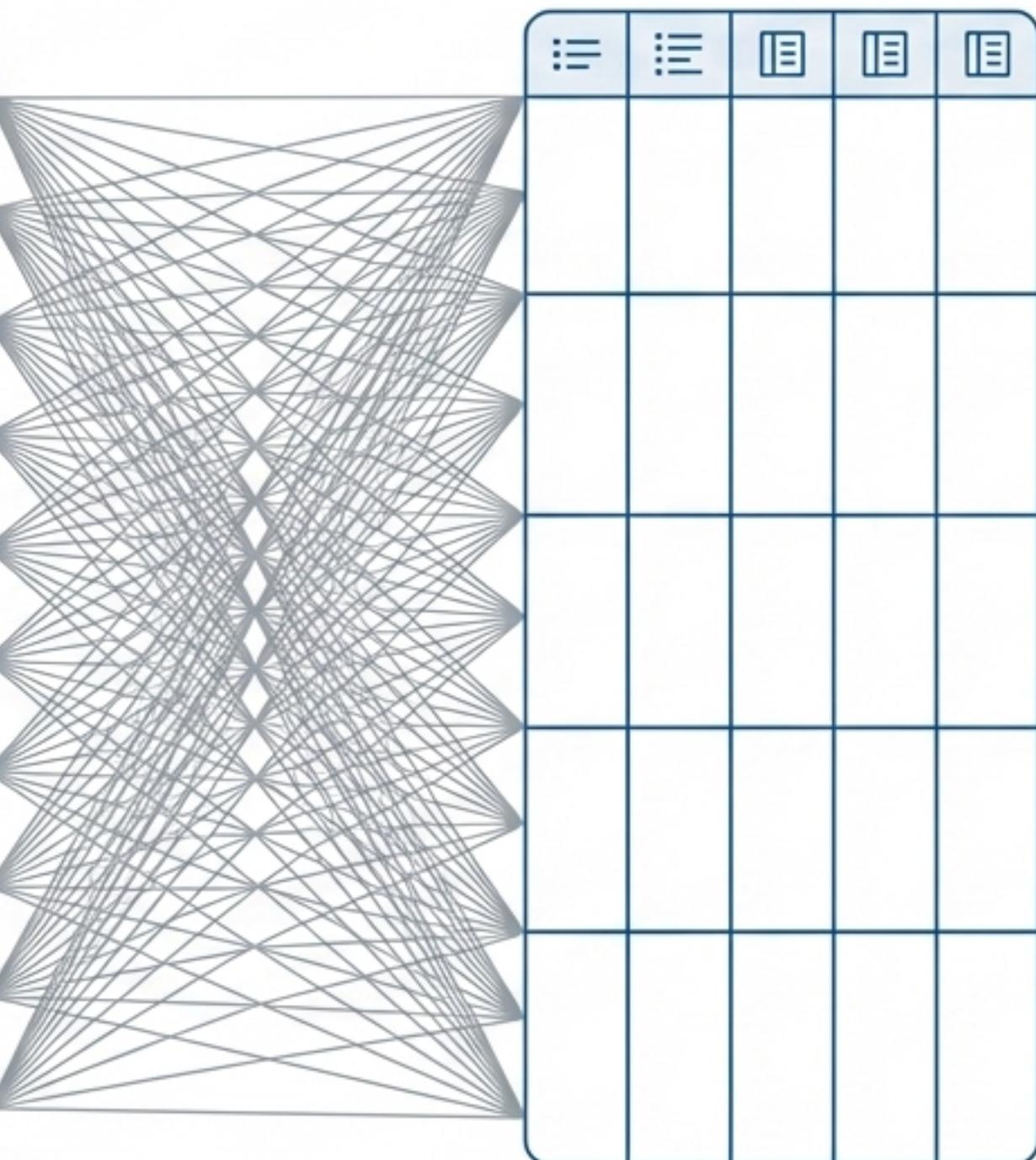
Processo travado, `TIMEOUT`.

Tabela A

≡	—	—
≡	—	—
≡	—	—
≡	—	—
≡	—	—
≡	—	—
≡	—	—
≡	—	—
≡	—	—
≡	—	—

Tabela B

≡	≡	≡	≡	≡



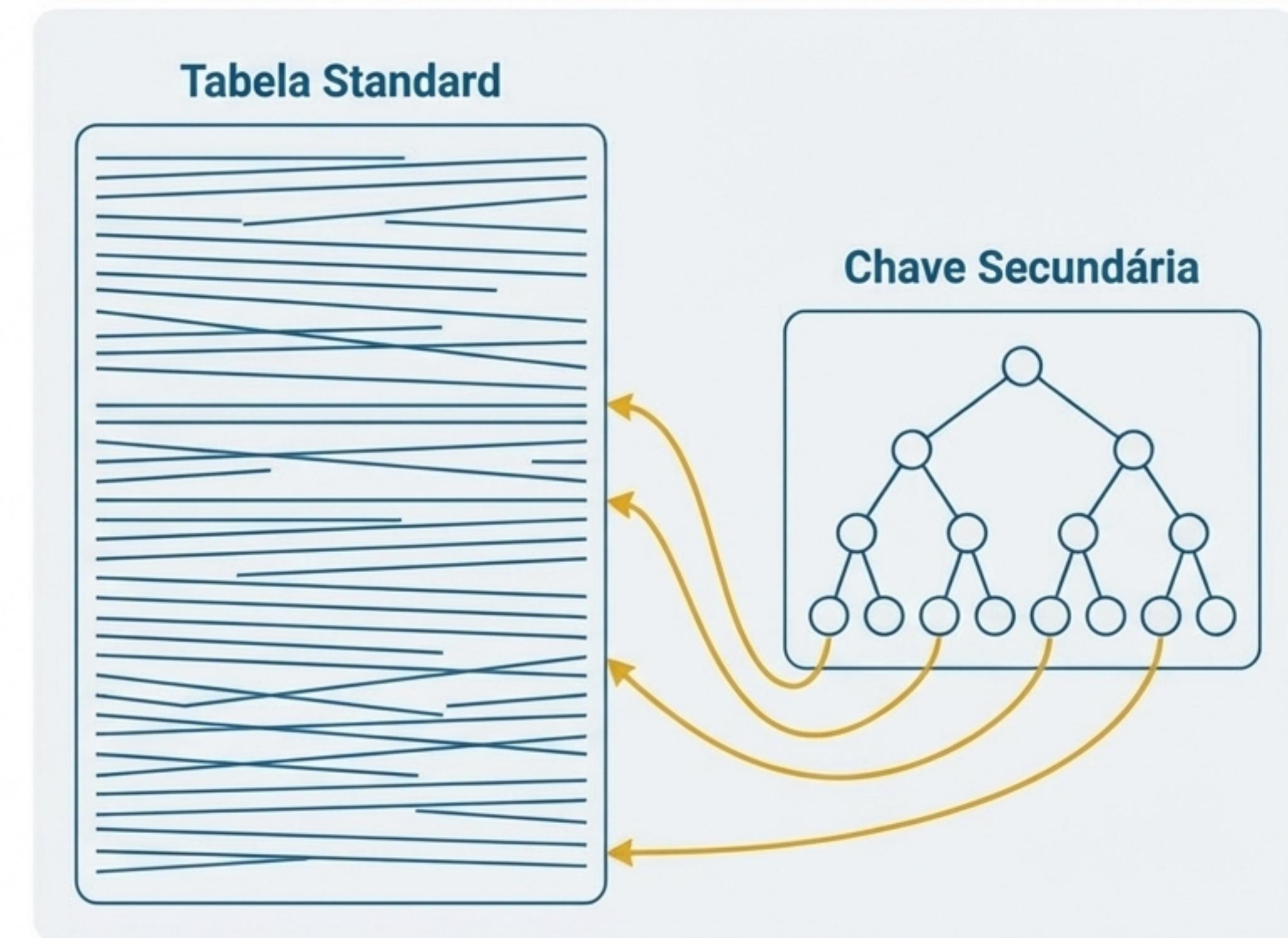
# A Solução: A Mágica das Chaves Secundárias

## O que são?

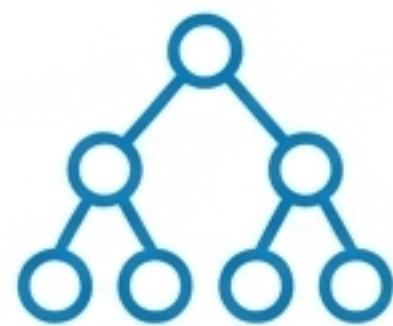
Um "índice auxiliar" em memória que o ABAP cria e mantém para você. A tabela original (Standard) (Standard) permanece intacta.

## Como funciona?

Em vez de varrer a tabela, o sistema consulta este índice otimizado (árvore ou mapa de hash) que aponta diretamente para a linha correta.



# Escolhendo a Ferramenta Certa: Sorted vs. Hashed

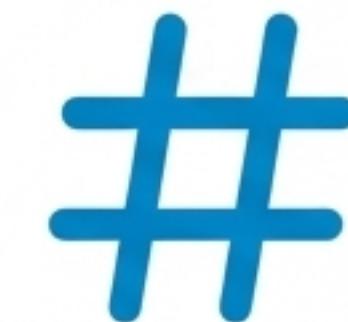


## SORTED KEY

**Mecanismo:** Índice ordenado (Árvore Binária).

**Complexidade:**  $O(\log n)$  - Extremamente rápido e escalável.

**Ideal para:** Buscas parciais e por intervalos (ranges).



## HASHED KEY

**Mecanismo:** Mapa de Hash.

**Complexidade:**  $O(1)$  - Acesso instantâneo (o 'Santo Graal').

**Ideal para:** Buscas exatas pela chave completa.

# Mão na Massa: Sintaxe e Uso

## 1. Definição da Chave:

```
TYPES ty_t_data TYPE STANDARD TABLE OF ty_data
  WITH DEFAULT KEY
    WITH UNIQUE HASHED KEY key_id COMPONENTS id
    WITH NON-UNIQUE SORTED KEY key_cat COMPONENTS category.
```

## 2. Uso Obrigatório na Leitura:

```
LOOP AT lt_data ASSIGNING <fs_row>
  USING KEY key_cat
  WHERE category = 'Hardware'.
```



**Atenção:** Sem `USING KEY`, a busca volta a ser linear!

# Além da Busca: Otimizando o Acesso à Memória

Ao iterar em um loop, como os dados da linha são acessados?



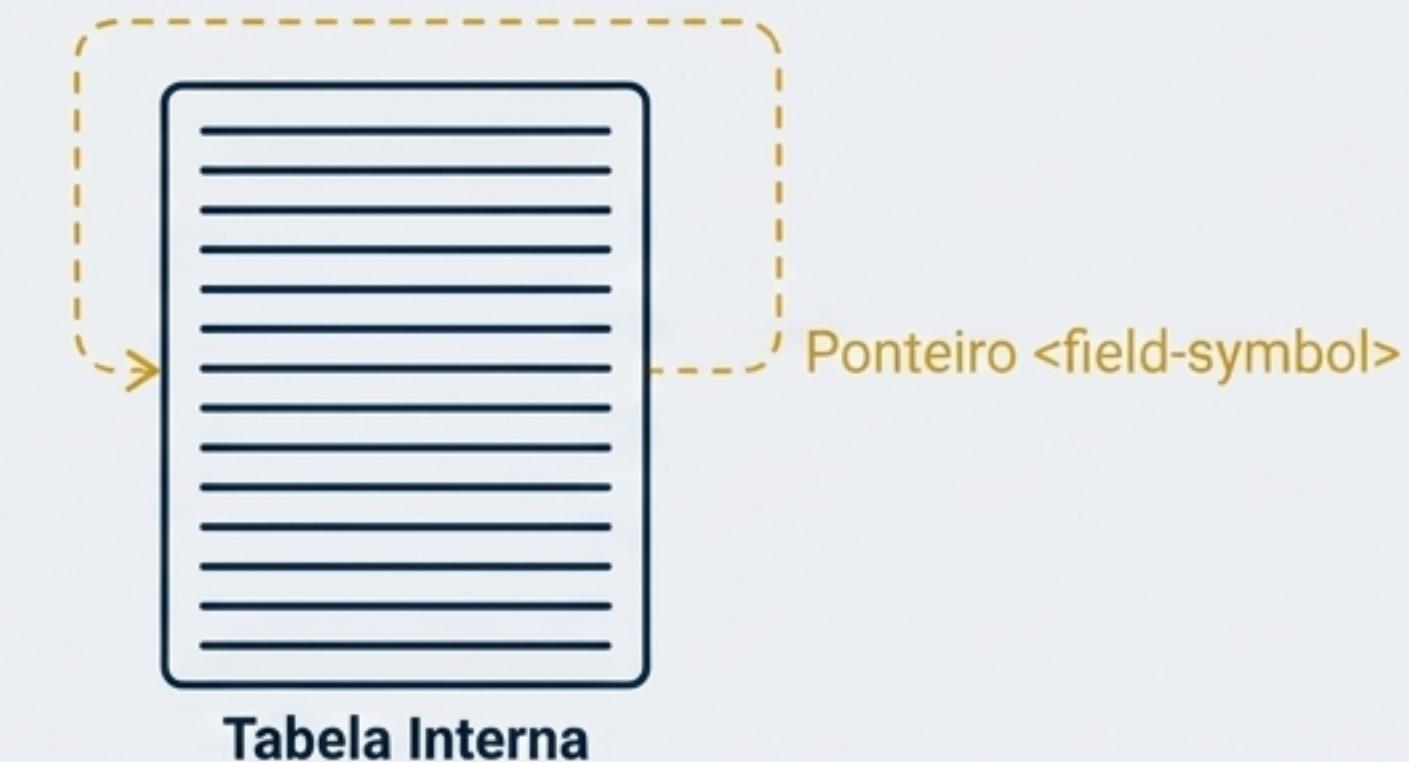
## Cópia (INTO)

Cria uma duplicata física da linha na memória (Work Area).  
Lento e custoso para tabelas largas ('Wide Tables').



## Referência (ASSIGNING)

Cria um ponteiro direto para a linha na memória da tabela.  
Performance máxima, sem custo de cópia.



# Guia de Decisão Rápido: Estratégias de Loop

Cenário	Estratégia Recomendada	Por quê?
Modificar a tabela	ASSIGNING FIELD-SYMBOL(<fs>)	Permite alteração direta, performance máxima.
Tabela Larga (muitas colunas)	ASSIGNING FIELD-SYMBOL(<fs>)	Evita o custo da cópia de memória a cada iteração.
Ler um campo pequeno (Inteiro/Char)	INTO DATA(wa)	Mais legível, custo de cópia desprezível.
Contexto OO moderno	REFERENCE INTO DATA(lr)	Alternativa segura e orientada a objetos ao `ASSIGNING`.

# Prova de Conceito: Colocando a Teoria à Prova

## O Experimento

1. **Setup:** Criar uma tabela interna com **500.000** registros de materiais.
2. **Cenário:** Buscar pelo último material inserido (M499999) - o pior caso para a busca linear.
3. **Medição:** Cronometrar o tempo de execução (em microsegundos) das duas abordagens.



## As Abordagens em Teste



### Teste Lento

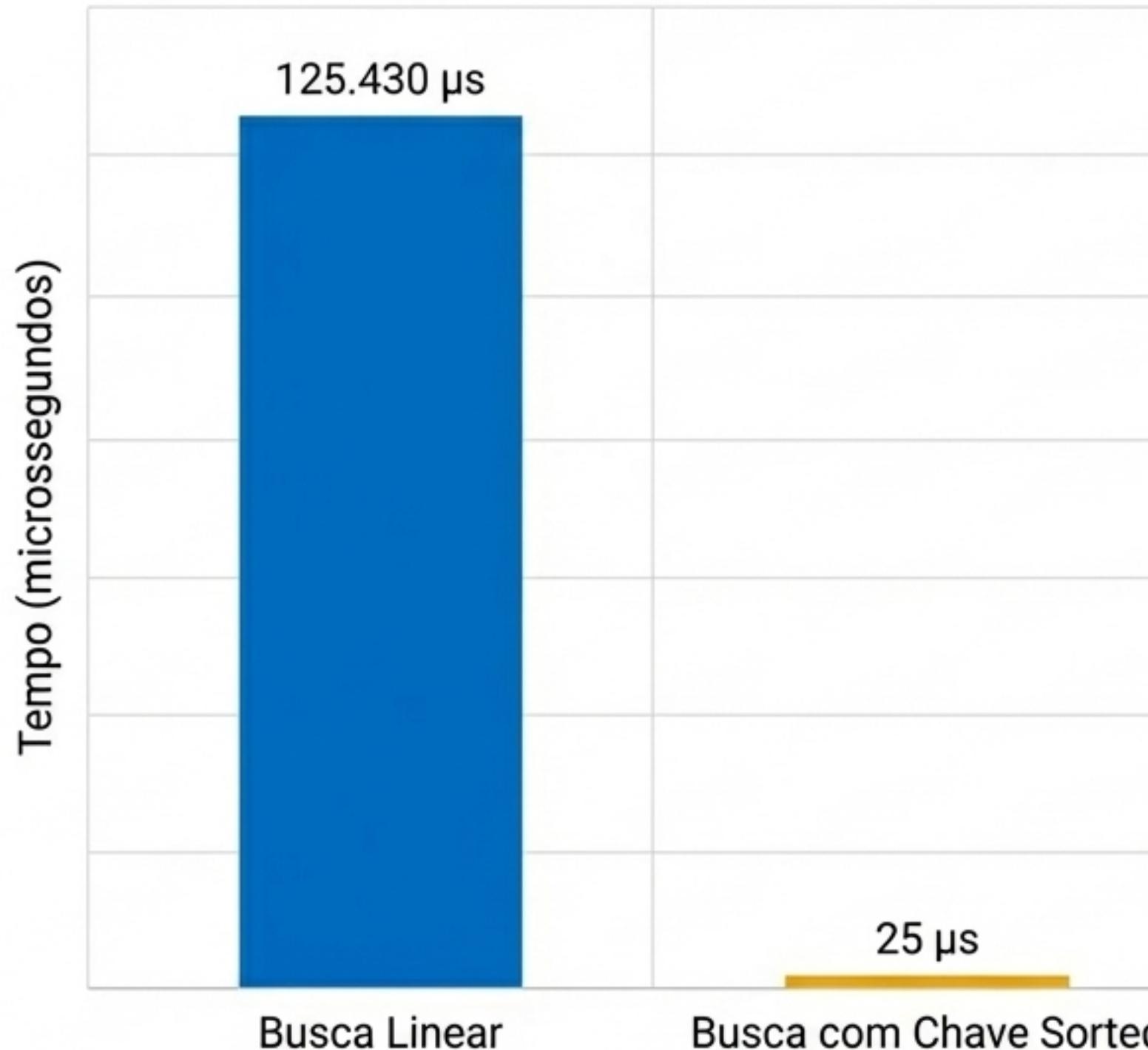
LOOP AT ... WHERE matnr = ...  
(Busca Linear Padrão)



### Teste Rápido

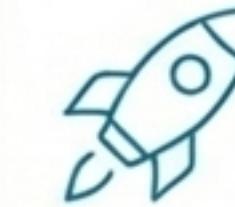
LOOP AT ... USING KEY sk\_matnr  
WHERE matnr = ... (Busca com  
Chave Secundária Sorted)

# Resultados do Benchmark: A Diferença é Brutal



**Tempo Busca Linear**

125.430 microssegundos



**Tempo Busca com Chave**

25 microssegundos

**Ganho de Performance**

**~5.000x mais rápido!**

# Glossário Técnico: Consolidando o Conhecimento



## Busca Linear ( $O(n)$ )

Varrer a tabela do início ao fim. Padrão para Tabelas Standard sem chave explícita.



## Busca Binária ( $O(\log n)$ )

Algoritmo "dividir para conquistar" para tabelas ordenadas. Usado em tabelas Sorted.



## Hash ( $O(1)$ )

Acesso matemático direto à memória. O mais rápido possível.



## Chave Secundária

Índice adicional em memória para buscas rápidas por campos não-chave.



## Field-Symbol (<fs>)

Um ponteiro que aponta para um endereço de memória, evitando cópias de dados.

# Resumo: Suas Novas Ferramentas de Performance

**Se você lê uma tabela muitas vezes por uma chave específica, ela NÃO DEVE ser Standard ou acessada linearmente.**

## Checklist de Otimização

-  Identifique **LOOPs** aninhados ou leituras repetitivas.
-  Adicione uma **CHAVE SECUNDÁRIA** (Hashed para buscas exatas, **Sorted** para ranges).
-  Use **USING KEY** em seus comandos **LOOP** e **READ**.
-  Prefira **ASSIGNING** em vez de **INTO** para tabelas largas ou quando precisar modificar dados.

# Quiz Rápido: Teste seu Conhecimento



Por que a busca em uma Tabela Hashed é considerada  $O(1)$  e a mais rápida para grandes volumes?



Tenho uma tabela Standard com 500 mil clientes. Como otimizo uma busca repetitiva pelo CPF (que não é a chave primária)?



Qual a vantagem de performance de usar `ASSIGNING FIELD-SYMBOL` em vez de `INTO` em um loop sobre uma tabela com 200 colunas?