

# O Inimigo Silencioso dos Seus Programas: O Timeout

## Como operações aparentemente inofensivas em desenvolvimento se transformam em falhas catastróficas em produção.

- O erro de performance mais comum e devastador no desenvolvimento ABAP é a leitura ineficiente de tabelas internas.
- Em ambientes de desenvolvimento, com poucos dados, o problema é invisível. Em produção, com milhões de registros, o mesmo código causa 'Timeouts' (ST22), paralisando processos críticos de negócio.
- Este guia revela as causas e apresenta as soluções definitivas para blindar seu código.

# O Custo Oculto da Busca Linear: Por Dentro do ‘Table Scan’

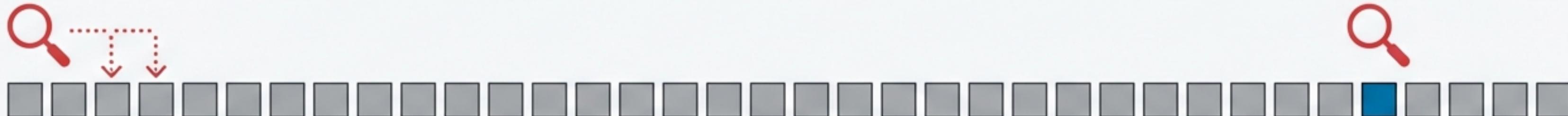
## O Problema

- Uma ‘Tabela Standard’ é como uma lista de compras: organizada apenas pela ordem de inserção. Não há um índice para buscas por conteúdo.
- Quando executamos ‘READ TABLE’ ou ‘LOOP ... WHERE’ sem uma chave otimizada, o ABAP inicia uma **Busca Linear**.
- Ele verifica a linha 1, depois a linha 2, a linha 3, e assim por diante, até encontrar o registro ou chegar ao fim da tabela.

## A Consequência

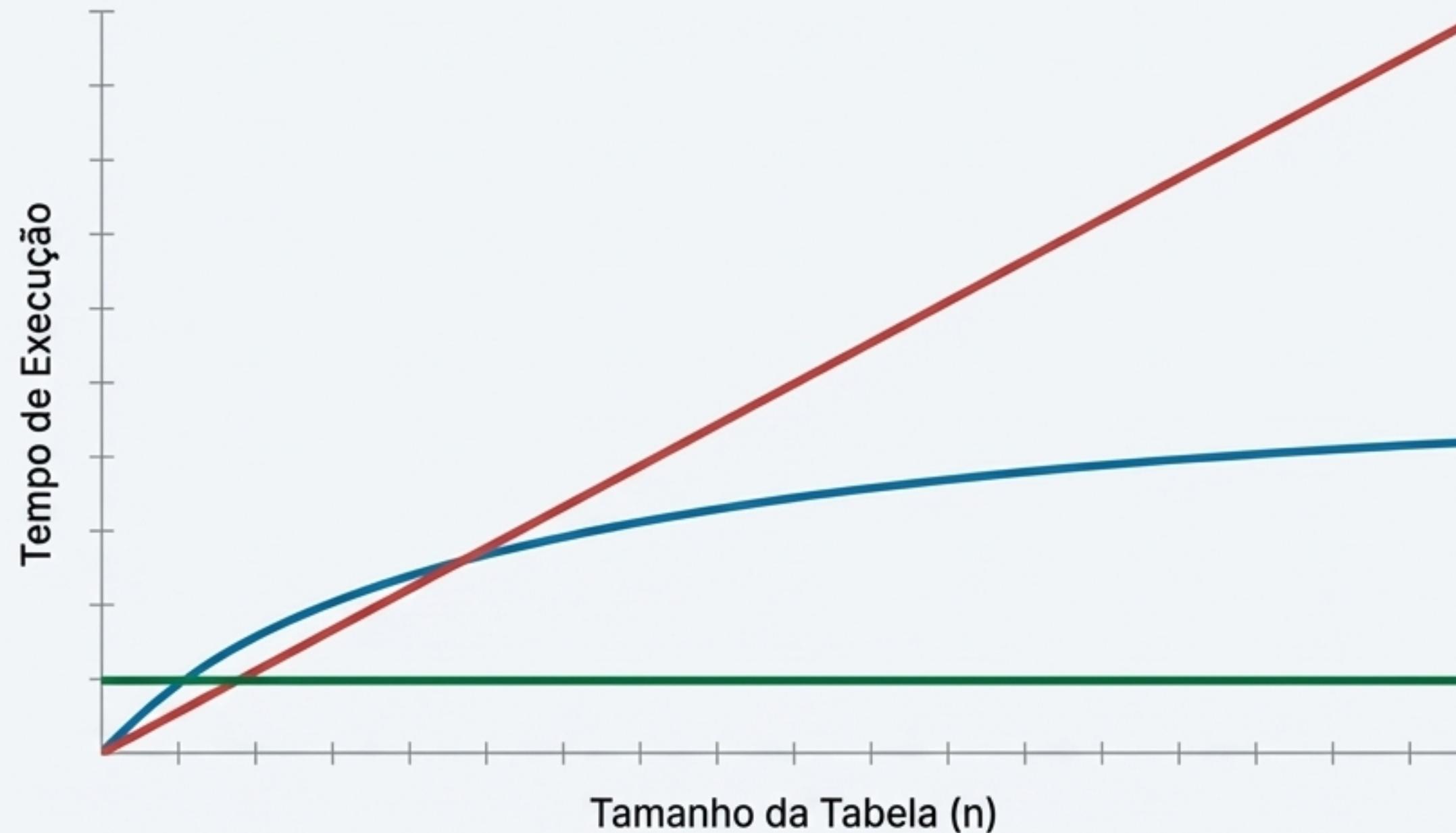
- **Cenário de Desenvolvimento:** Em 100 linhas, a busca é instantânea.
- **Cenário de Produção:** Em 1.000.000 de linhas, o sistema pode levar segundos para encontrar *um* único registro. Se essa busca estiver dentro de um loop, o programa irá travar.

## Table Scan



# Medindo a Eficiência: Entendendo a Complexidade Algorítmica (Notação Big O)

A Notação Big O descreve como o tempo de execução de um algoritmo cresce conforme o volume de dados ( $n$ ) aumenta.



- O(n) - Linear**  
O tempo cresce na mesma proporção que os dados. **Perigoso para grandes volumes.**  
(Ex: Busca em Tabela Standard)
- O(log n) - Logarítmica**  
O tempo cresce muito lentamente. Dobrar os dados não dobra o tempo.  
**Altamente escalável.**  
(Ex: Busca Binária em Tabela Sorted)
- O(1) - Constante**  
O tempo de acesso é o mesmo, não importa o tamanho da tabela.  
**O "Santo Graal" da performance.**  
(Ex: Busca em Tabela Hashed)

# A Solução Estratégica: Adicionando Inteligência com Chaves Secundárias

## O Cenário

Frequentemente, recebemos uma Tabela Standard de uma BAPI ou classe global e não podemos alterar seu tipo. Como otimizar as buscas?

## A Resposta

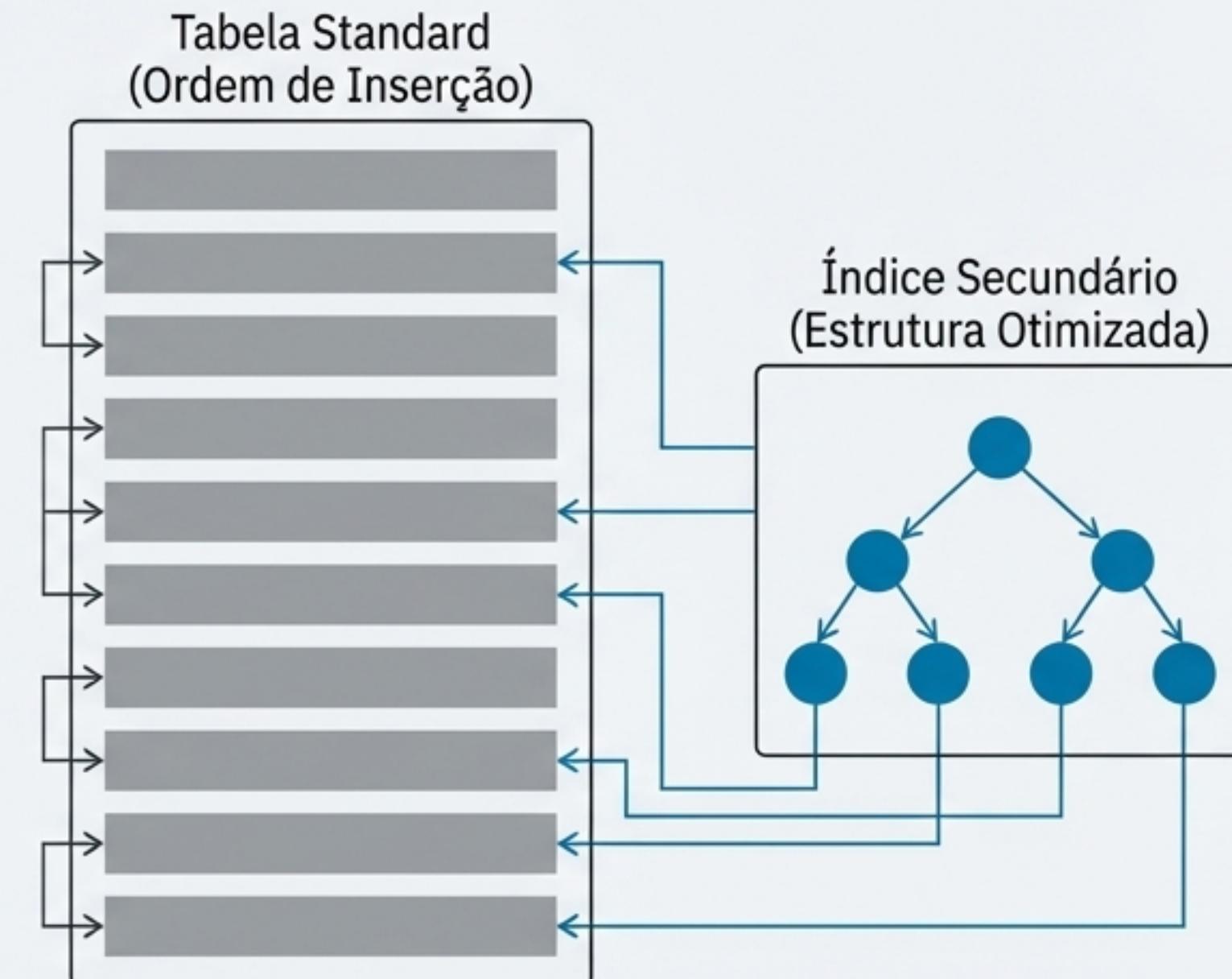
‘SECONDARY KEYS’.

Uma Chave Secundária cria um “índice auxiliar” em memória, paralelo à tabela.

O sistema mantém a ordem original, mas constrói uma estrutura de dados otimizada (árvore de busca ou mapa de hash) que aponta para as linhas da tabela principal.

## Custo-Benefício

Adiciona um pequeno overhead de memória (10-15%) e de CPU na inserção, mas o ganho na leitura é exponencial.



# Anatomia de uma Chave Secundária: `SORTED` vs. `HASHED`

## `SORTED KEY`

- Cria um índice ordenado (árvore binária).
- **Uso Ideal:** Buscas parciais (pelo primeiro campo da chave) e buscas por intervalo (ranges).
- **Complexidade:**  $O(\log n)$

## `HASHED KEY`

- Cria um mapa de hash.
- **Uso Ideal:** Buscas exatas pela chave completa. Imbatível em velocidade.
- **Complexidade:**  $O(1)$

```
TYPES: BEGIN OF ty_data,
        id      TYPE i,
        category TYPE string,
        name    TYPE string,
      END OF ty_data.
```

```
TYPES ty_t_data TYPE STANDARD TABLE OF ty_data
```

```
  WITH DEFAULT KEY
```

```
    " Chave HASHED única para buscas por ID exato
```

```
  WITH UNIQUE HASHED KEY key_id COMPONENTS id
```

```
    " Chave SORTED não-única para buscas por categoria
```

```
  WITH NON-UNIQUE SORTED KEY key_cat COMPONENTS category.
```

‘UNIQUE’ garante a unicidade da chave (validado na inserção), enquanto ‘NON-UNIQUE’ permite valores duplicados.

# Ativando a Alta Performance: O Uso Obrigatório de 'USING KEY'

## A Regra Crítica

Para que o ABAP utilize a chave secundária, você **deve** ser explícito na sua leitura. Se a cláusula `USING KEY` for omitida, o sistema reverterá para a busca linear padrão, ignorando seu índice otimizado.

```
" Acesso Ultra-Rápido via Chave Hashed (0(1))
" Sintaxe moderna para leitura de linha única
DATA(ls_row) = lt_data[ KEY key_id COMPONENTS id = 500 ].  
  
" Loop Otimizado via Chave Sorted (0(log n))
" Sem 'USING KEY', este loop faria um scan completo.
" Com 'USING KEY', ele salta direto para o bloco da categoria 'Hardware'.
LOOP AT lt_data ASSIGNING <fs_row>
    USING KEY key_cat
    WHERE category = 'Hardware'.
    " ... processamento ...
ENDLOOP.
```

Aqui está o segredo!

Esta cláusula ativa o índice e evita o 'Table Scan'.

# Gerenciando Memória: O Combate entre Cópia (`INTO`) e Referência (`ASSIGNING`)

Em cada iteração de um loop, o ABAP precisa disponibilizar a linha atual. A forma como isso é feito tem um impacto drástico na performance.

## Abordagem 1: Cópia (`INTO`)

- Mecanismo:** Cria uma duplicata física (cópia byte a byte) da linha em uma "Work Area".
- Prós:** Seguro. Modificações na Work Area não afetam a tabela original.
- Contras:** Custo de CPU e memória. Em "Wide Tables", alocar e copiar centenas de colunas milhões de vezes degrada a performance.

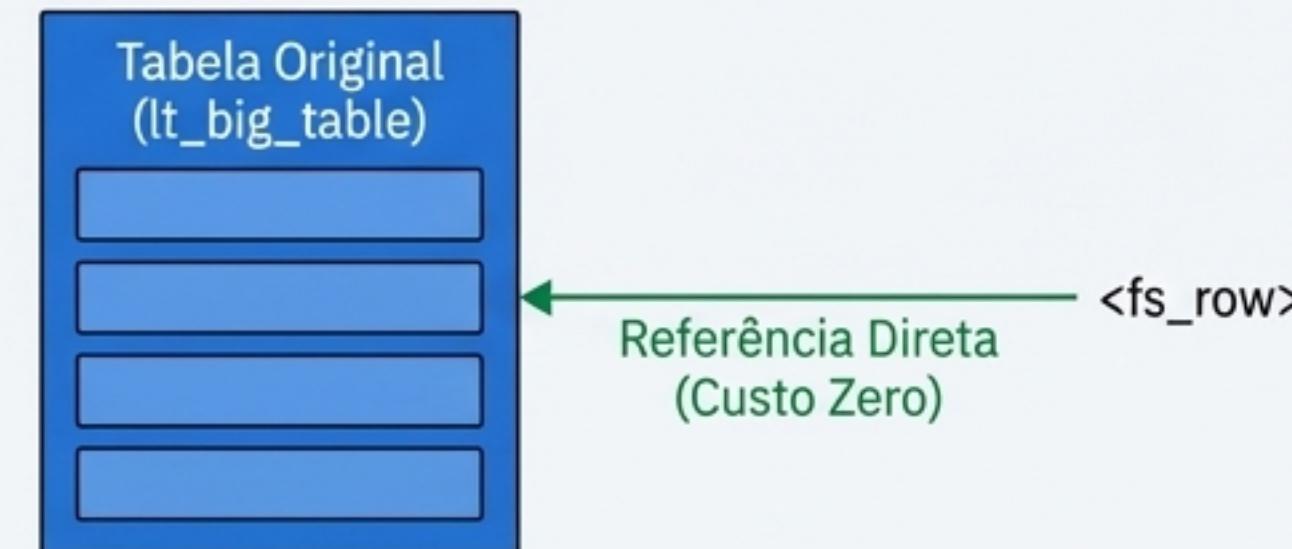
```
LOOP AT lt_big_table INTO DATA(ls_copy).
```



## Abordagem 2: Referência (`ASSIGNING`)

- Mecanismo:** Cria um ponteiro (`Field-Symbol`) que aponta diretamente para o endereço de memória da linha na tabela. Nenhuma cópia é feita.
- Prós:** Velocidade máxima, consumo mínimo de memória. Permite modificar a tabela diretamente.
- Contras:** Exige cuidado. Modificar um campo da chave de uma tabela Sorted/Hashed pode causar um dump.

```
LOOP AT lt_big_table ASSIGNING FIELD-SYMBOL(<fs_row>).
```



# O Anti-Padrão Definitivo: Desarmando a Bomba-Relógio do Loop Aninhado

**Cenário Clássico:** Cruzar dados de duas tabelas (ex: Pedidos e Clientes).



```
LOOP AT Pedidos (10.000 linhas).
  READ TABLE Clientes (5.000 linhas) WHERE ...
ENDLOOP.
```

## Cálculo do Desastre

10.000 laços x 5.000 leituras =

**50.000.000**

Cinquenta Milhões de operações. Timeout garantido.



```
LOOP AT Pedidos (10.000 linhas).
  READ TABLE Clientes USING KEY ... WHERE ...
ENDLOOP.
```

## Cálculo da Eficiência

10.000 laços x 1 leitura =

**10.000**

Dez Mil operações.

O Ganho: A abordagem otimizada é cerca de **5.000 vezes mais rápida**.

# A Prova Incontestável: Benchmark de Performance em Ação

Vamos simular uma carga massiva de dados e medir o tempo de busca de um registro no final da tabela (o pior caso para a busca linear).

## 1. O Setup

- Criamos uma tabela standard `lt\_materials` com **500.000** registros.
- Adicionamos uma chave secundária sorted: `WITH NON-UNIQUE SORTED KEY sk\_matnr COMPONENTS matnr`.
- Alvo da busca: O material '**M499999**'.

## 2. O Código do Teste

```
" Teste 1: Busca Linear (lenta)
GET RUN TIME FIELD DATA(t1).
LOOP AT lt_materials INTO ls_wa WHERE matnr = lv_target.
ENDLOOP.
GET RUN TIME FIELD DATA(t2).
DATA(diff_linear) = t2 - t1.

" Teste 2: Busca com Chave Secundária (rápida)
GET RUN TIME FIELD t1.
LOOP AT lt_materials ASSIGNING <fs_fast>
      USING KEY sk_matnr
      WHERE matnr = lv_target.
ENDLOOP.
GET RUN TIME FIELD t2.
DATA(diff_sorted) = t2 - t1.
```

O código completo da classe `zcl\_perf\_demo` estará disponível como anexo.

# Os Resultados Falam por Si: Linear vs. Chave Secundária

Tempo da Busca Linear

**25.480**

microsssegundos

Tempo da Busca com Chave Sorted

**5**

microsssegundos

**Ganho de Performance: 5.096 vezes mais rápido!**

- A busca linear precisou varrer 499.999 registros para encontrar o alvo.
- A busca com chave sorted usou a busca binária ( $O(\log n)$ ), que em 500.000 registros requer aproximadamente 19 comparações. O resultado é praticamente instantâneo.

# Guia de Decisão Rápido: Qual Estratégia de Loop Usar?

Cenário	Estratégia Recomendada	Por quê?
<b>Iterar para ler um campo pequeno (Inteiro, Char)</b>	INTO DATA(wa)	Mais legível e o custo de cópia da work area é desprezível.
<b>Precisar modificar o conteúdo da tabela</b>	ASSIGNING FIELD-SYMBOL(<fs>)	Permite alterar '<fs>-campo' diretamente, modificando a tabela sem 'MODIFY'.
<b>Iterar sobre uma tabela larga (muitas colunas)</b>	ASSIGNING FIELD-SYMBOL(<fs>)	Evita a alocação de memória e o custo da cópia de centenas de bytes a cada iteração.
<b>Filtrar a tabela por um campo não-chave</b>	USING KEY <chave_secundária>	Transforma uma busca lenta O(n) em uma busca ultra-rápida O(log n) ou O(1).

# Seu Novo Arsenal de Alta Performance: Resumo Técnico

## Glossário Essencial

- **Busca Linear ( $O(n)$ ):** O padrão para Tabelas Standard. Varre a tabela do início ao fim. Ineficiente para grandes volumes.
- **Busca Binária ( $O(\log n)$ ):** Algoritmo de "dividir para conquistar" usado em `Sorted Tables` e `Secondary Sorted Keys`. Altamente escalável.
- **Acesso Hashed ( $O(1)$ ):** Acesso direto via cálculo matemático. O mais rápido possível. Usado em `Hashed Tables` e `Secondary Hashed Keys`.
- **Chave Secundária:** Um índice adicional que permite buscas rápidas por campos que não fazem parte da chave primária, sem alterar o tipo da tabela.
- **Field-Symbol (`<fs>`):** Uma variável ponteiro que aponta para um endereço de memória. Essencial para evitar cópias de dados e modificar tabelas em loops eficientemente.

**Regra de Ouro:** Se você vai ler uma tabela interna muitas vezes por uma chave específica (ex: dentro de um loop), ela **jámais** deveria ser acessada de forma linear. Use Chaves Secundárias.