



CWI SOFTWARE

módulo 2 | banco de dados

Crescer 2015-1

Feevale



Oracle - Otimização

André Luís Nunes

abril/2015

otimização

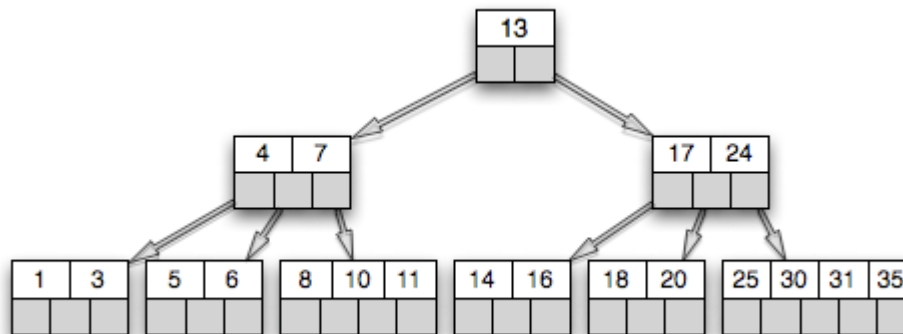
- Otimizador Oracle
- Explain
- Planejamento
- Dicas de boas práticas

Otimizador Oracle

• 1. ÍNDICES

São estruturas opcionais auxiliares que permitem otimizar o acesso aos dados de uma tabela.

- Exemplo de uma estrutura de um índice, em estrutura de árvore B-Tree:



• 1. ÍNDICES

Índices (Index) são importantes pois diminuem processamento e I/O em disco.

- Quando usamos um comando SQL para retirar informações de uma tabela, na qual, a coluna da mesma não possui um índice, o Oracle faz um Acesso Total a Tabela para procurar o dado, ou seja, realiza-se um FULL TABLE SCAN degradando a performance do Banco de Dados Oracle.
- Com o índice isso não ocorre, pois com o índice isso apontará para a linha exata da tabela daquela coluna retirando o dado muito mais rápido

• 1. ÍNDICES

Quando criá-los?

- Toda coluna que pertence a uma chave estrangeira precisa de índice!
- Toda coluna utilizada como filtro em determinada consulta precisa de índice.
- É possível criar índices com estrutura de colunas compostas (mais de uma coluna)
 - Exemplo: se determinada consulta sempre utiliza 2 campos, então deve-se criar um índice com ambas:

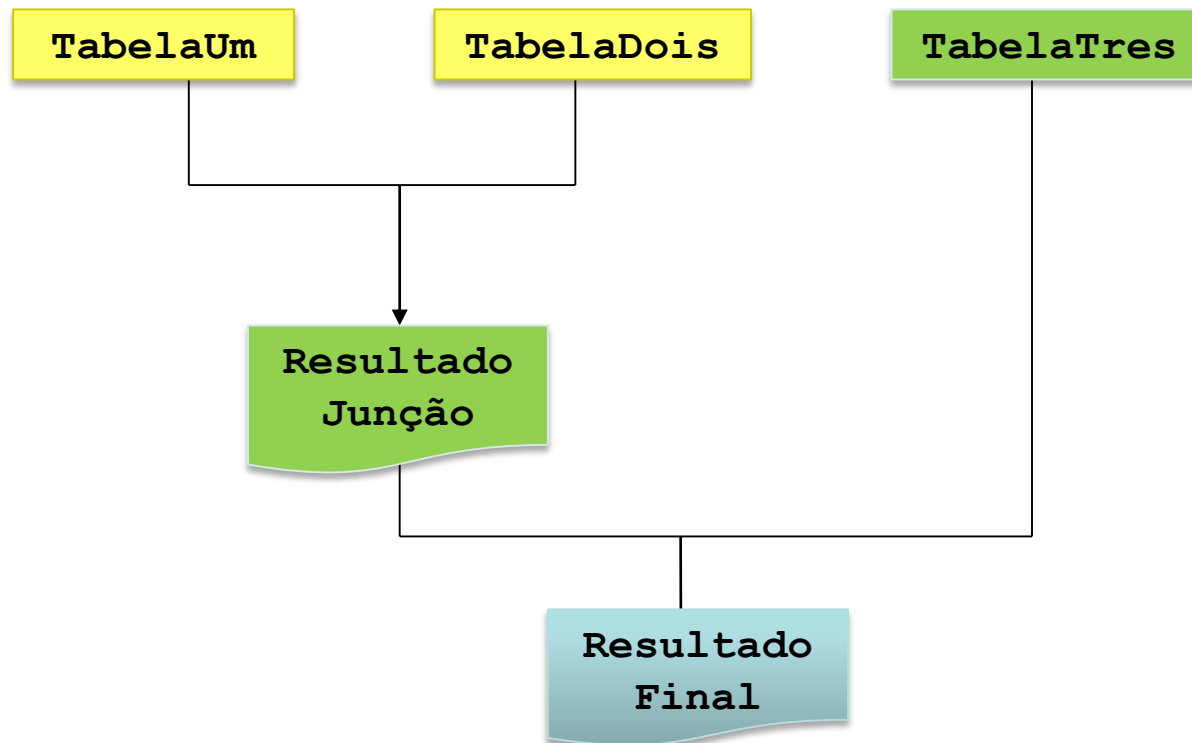
```
Select * From Cliente Where UF = :pUF and idCidade =:Pcidade
```
 - Criando o índice:

```
Create index IX_CLIENTE_UFCIDADE On Cidade (UF, IDCidade);
```

- **2. Métodos de acesso 1/8**
 - **Como são feitos os acessos no Oracle**
 - **Tipos de junções**
 - Nested Loop
 - Sort Merge
 - Hash
 - **Estatísticas**

• 2. Métodos de acesso 2/8

– Como são feitos os acessos no Oracle



- **2. Métodos de acesso 3/8**

- **Tipos de junções**

- **Nested Loop:** método mais vantajoso para junções de fontes de dados que contém poucos registros e existe uma relação clara de dependência entre as fontes. (NOT IN, EXISTS e IN forçam a utilizar este método).

• 2. Métodos de acesso 4/8

– Tipos de junções

- **Sort Merge:** se não existir relação de dependência entre as fontes de dados, o otimizador não poderá executar *Nested Loop*. Se for utilizado operadores de igualdade o método hash costuma ter melhor desempenho.

Basicamente classifica todas as linhas relevantes na primeira tabela pela chave JOIN, bem como as linhas relevantes da segunda tabela pela chave JOIN, e junta estas linhas classificadas.

Geralmente a última opção do otimizador.

• 2. Métodos de acesso 5/8

– Tipos de junções

- **Hash:** ausência de estatísticas leva a utilizar este operador, funções no *where* também forçam a utilizar este operador. O otimizador escolhe a menor fonte de dados para armazenar em memória. Requer mais memória que os outros métodos.

• 2. Métodos de acesso 6/8

- **Estatísticas:** são armazenadas no dicionário de dados. Mais detalhes sobre os objetos do banco de dados:

- **Tabela**

- » Número de registros
- » Número de blocos
- » Tamanho médio do registro

- **Coluna**

- » Número de valores distintos (NDV) na coluna
- » Número de nulos na coluna
- » Distribuição dos dados (histograma)

• 2. Métodos de acesso 7/8

– **Estatísticas (continuação):** armazenam mais detalhes sobre os objetos do banco de dados:

- **Sistema**

- » Desempenho e utilização de I/O e CPU

- **Índice**

- » Número de blocos folha

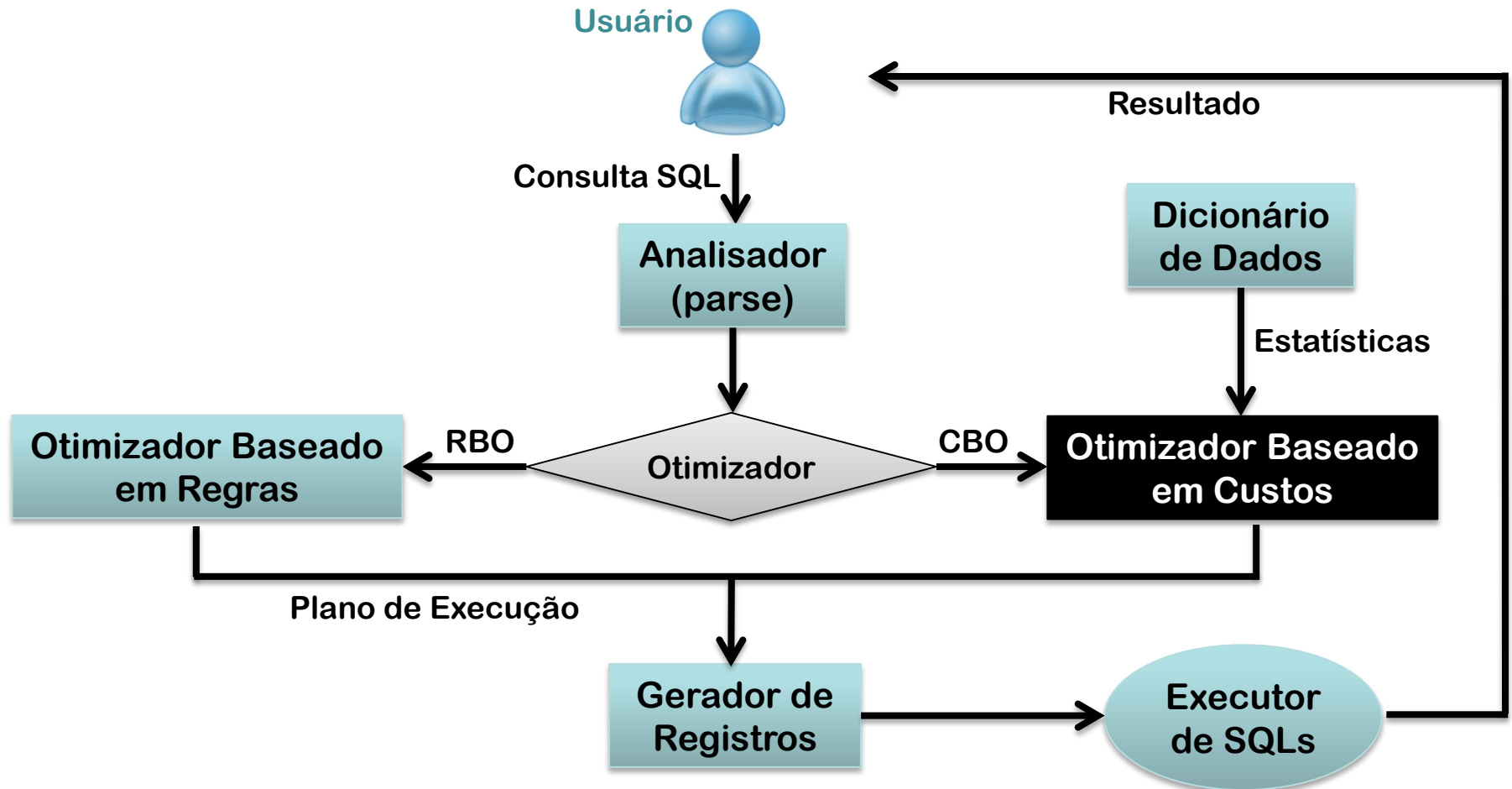
- » Níveis

As informações sobre as estatísticas podem ser consultadas nas visões do sistema, como:

- USER_TABLES
- USER_INDEXES
- USER_TAB_COLUMNS

Coluna **LAST_ANALYZED** indica última análise realizada.

• 2. Métodos de acesso 8/8



• 2. Modos de otimização

- **RULE:** baseado em regras (15 regras), não utiliza informações das estatísticas.
- **CHOOSE:** avalia a estatística e verifica qual o menor custo de I/O e CPU.
- **ALL_ROWS:** Objetivo de retornar todas as linhas da consulta com o menor tempo possível. Recomendado para ambientes de processamento em lote, DW.
- **FIRST_ROWS:** o recomendado para ambientes com alto nível transacional, OLTP. Otimiza para trazer a primeira linhas, forçando o uso de índices.

• 3. Plano de execução

– Métodos de acesso

- **Full Table Scan (FTS):** leitura sequencial de todos os registros da tabela.
- **Rowid:** é o mais rápido, através da identificação do registro retorna a informação do bloco de dados.
- **Index Lookup:** os dados são acessados através da busca pela chave no índice, que contém a informação do bloco de dados referente.
 - Quatro métodos: **unique** , **index range**, **full** e **fast full**.

• 3. Plano de execução

– Métodos de acesso → Index Lookup:

- **Index Unique:** sempre retorna um registro único;
- **Index Range Scan:** retorna um intervalo de registros, geralmente através dos operadores >, <, >=, <=, between;
- **Index Full Scan:** leitura sequencial no índice. Quando possui estatísticas, provavelmente é mais eficiente quando o otimizador escolhe este método;

» Exemplo: `Select EmpNo, EName From Emp Order by 1,2;`

- **Index Fast Full Scan:** leitura sequencial no índice, porém sem precisar ordená-lo.

» Exemplo: `Select EmpNo, EName From Emp;`

• 3. Plano de execução: HINT de otimização

Os 8 mais utilizados:

- **INDEX:** força o uso de determinado índice;
- **PARALLEL:** permite quantidade de servidores;
- **FIRST_ROWS:** força o uso de índices e retorna mais rapidamente as primeiras linhas;
- **RULE:** força a otimização baseada em regras;
- **FULL:** força a varredura completa da tabela (full scan);
- **LEADING:** semelhante ao ORDERED, mas é usado para informar apenas a primeira tabela;
- **USE_NL:** força a utilização do Nested Loops (loops aninhados);
- **USE_HASH:** força a utilização do método HASH.

• 3. Plano de execução: HINT de otimização

Para utilizar este recurso é necessário colocar logo após o SELECT o início de comentário seguido do operador de soma + e em seguida a opção e os parâmetros de cada um.

Exemplos:

- Forçando a ordem das tabelas (ORDERED)

```
SELECT /*+ORDERED*/  
e.Employee_Id, e.Last_Name, d.Name  
FROM Employee e,  
Department d  
WHERE e.Department_Id = d.Department_Id;
```

- Forçando determinado índice (INDEX)

```
SELECT /*+INDEX(EMPLOYEE,IX_EMPLOYEE_LASTNAME)*/  
Employee_Id, Last_Name  
FROM Employee;
```

• 3. Plano de execução

Veja os quadros abaixo, com informações sobre as tabelas e a estrutura da tabela Employee:

Informações sobre as tabelas			
Tabela	Registros	Blocos	Méd. Tam. Linha
Employee	32	1	42
Department	11	1	15
Location	4	1	11

EMPLOYEE			
Column	Null	Datatype	Index
EMPLOYEE_ID	Not Null	NUMBER(4)	EMP_PRIMARY_KEY
LAST_NAME		VARCHAR2(15)	IX_EMPLOYEE_LAST_NAME
FIRST_NAME		VARCHAR2(15)	IX_EMPLOYEE_MIDDLE
MIDDLE_INITIAL		VARCHAR2(1)	
JOB_ID		NUMBER(3)	
MANAGER_ID		NUMBER(4)	
HIRE_DATE		DATE	
SALARY		NUMBER(7,2)	
COMMISSION		NUMBER(7,2)	
DEPARTMENT_ID		NUMBER(2)	IX_EMPLOYEE_DEPARTMENT

→ Selecionado

• 3. Plano de execução

Estrutura das tabelas Department e Location:

DEPARTMENT			
Column	Null	Datatype	Index
DEPARTMENT_ID	Not Null	NUMBER(2)	DEPT_PRIMARY_KEY
NAME		VARCHAR2(14)	
LOCATION_ID		NUMBER(3)	IX_DEPARTMENT_LOCATION

Selecionado

LOCATION			
Column	Null	Datatype	Index
LOCATION_ID	Not Null	NUMBER(3)	LOC_PRIMARY_KEY
REGIONAL_GROUP		VARCHAR2(20)	IX_LOC_REGIONAL_GROUP

Filtro aplicado, operador de igualdade (o índice existente não é *unique*).

• 3. Plano de execução

COMO GERAR O EXPLAIN :

```
Explain plan for  
  Select *  
  From Tabela;
```

COMO VISUALIZAR:

```
Select *  
from table(dbms_xplan.display);
```

• 3. Plano de execução: exemplo

- Entendendo o plano de execução (PL/SQL Developer):

```
Select e.last_name "LName", d.name "Dept"
From   Employee   E,
       Department D,
       Location    L
where  l.regional_group = 'BOSTON'
and    d.Location_ID    = l.Location_ID
and    e.Department_ID  = d.Department_ID
```

Optimizer goal: All rows

Description	Object name	Cost	IO cost	Cardinality	Bytes	CPU cost
SELECT STATEMENT, GOAL = ALL_ROWS		4	4	8	288	38919
TABLE ACCESS BY INDEX ROWID	EMPLOYEE	1	1	3	30	8781
NESTED LOOPS		4	4	8	288	38919
NESTED LOOPS		3	3	3	78	14521
TABLE ACCESS FULL	LOCATION	2	2	1	11	6691
TABLE ACCESS BY INDEX ROWID	DEPARTMENT	1	1	3	45	7831
INDEX RANGE SCAN	IX_DEPARTMENT_LOCATION	0	0	3		1450
INDEX RANGE SCAN	IX_EMPLOYEE_DEPARTMENT	0	0	3		1650

• 3. Plano de execução: exemplo

- Gerando o plano de execução (sqlplus):

```
C:\Windows\system32\CMD.exe - sqlplus /nolog

D:\CWI\SQL>sqlplus /nolog

SQL*Plus: Release 10.2.0.1.0 - Production on Tue Sep 15 10:10:10 2009
Copyright (c) 1982, 2005, Oracle. All rights reserved.

SQL> set lines 300
SQL> set pages 150
SQL> conn cwi/cwi
Connected.
SQL> explain plan for
  2  Select e.last_name "LName", d.name "Dept"
  3  From    Employee   E,
  4          Department D,
  5          Location   L
  6  where   l.regional_group = 'BOSTON'
  7          and d.Location_ID = l.Location_ID
  8          and e.Department_ID = d.Department_ID
  9  ;

Explained.

SQL>
```

```
EXPLAIN PLAN FOR
Select e.last_name "LName",
       d.name "Dept"
From   Employee   E,
       Department D,
       Location   L
where  l.regional_group = 'BOSTON'
and    d.Location_ID   = l.Location_ID
and    e.Department_ID = d.Department_ID
```

• 3. Plano de execução: exemplo

- Entendendo o plano de execução (sqlplus):

```
C:\Windows\system32\CMD.exe - sqlplus /nolog
SQL> select * from table(dbms_xplan.display);
```

Select * from table(dbms_xplan.display);

```

PLAN_TABLE_OUTPUT
-----
Plan hash value: 896372569

-----
| Id | Operation                      | Name                      | Rows | Bytes | Cost (%CPU)| Time |
-----
|  0 | SELECT STATEMENT                |                          |      8 | 320 | 4 (0)| 00:00:01 |
|  1 |   TABLE ACCESS BY INDEX ROWID | EMPLOYEE                  |      3 | 33 | 1 (0)| 00:00:01 |
|  2 |     NESTED LOOPS                 |                          |      8 | 320 | 4 (0)| 00:00:01 |
|  3 |       NESTED LOOPS               |                          |      3 | 87 | 3 (0)| 00:00:01 |
| * 4 |         TABLE ACCESS FULL      | LOCATION                  |      1 | 11 | 2 (0)| 00:00:01 |
|  5 |           TABLE ACCESS BY INDEX ROWID | DEPARTMENT              |      3 | 54 | 1 (0)| 00:00:01 |
| * 6 |             INDEX RANGE SCAN    | IX_DEPARTMENT_LOCATION   |      3 |    | 0 (0)| 00:00:01 |
| * 7 |             INDEX RANGE SCAN    | IX_EMPLOYEE_DEPARTMENT   |      3 |    | 0 (0)| 00:00:01 |
-----

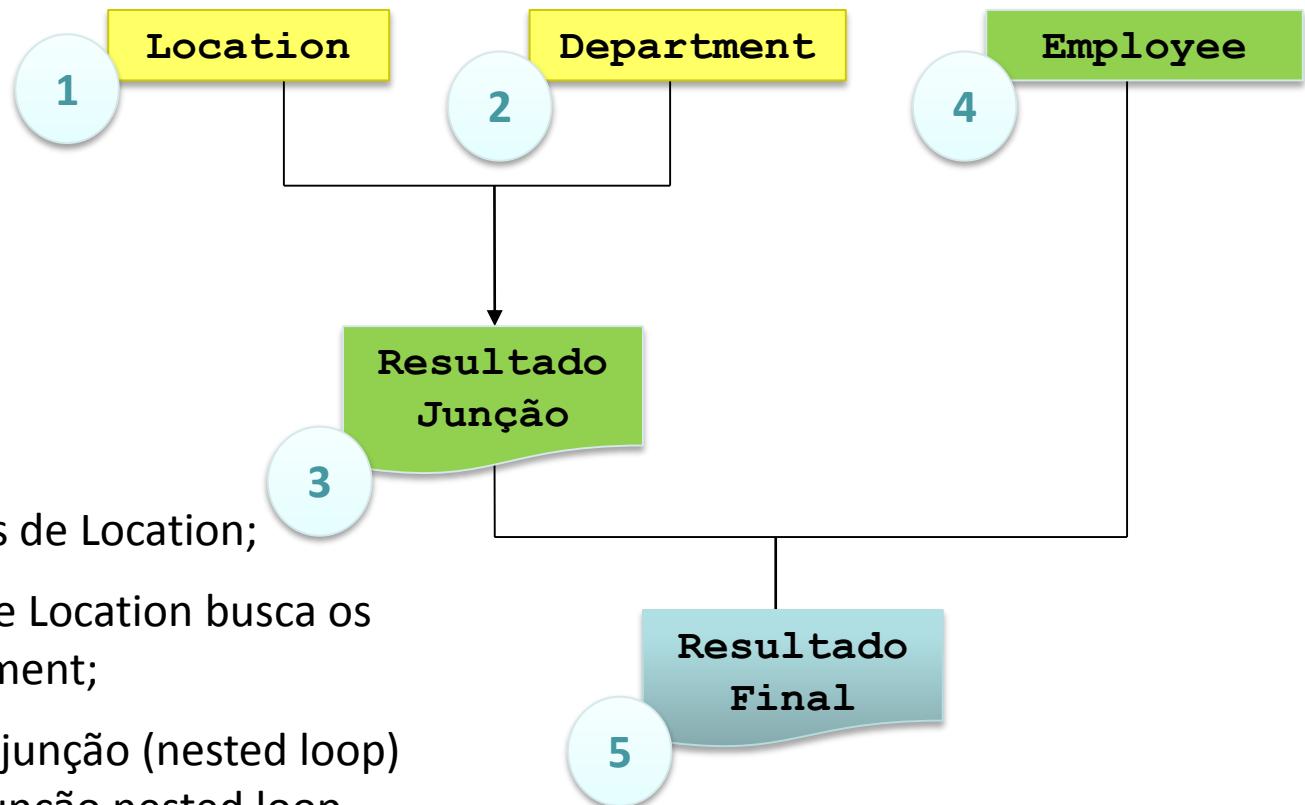
Predicate Information (identified by operation id):
-----
  4 - filter("L"."REGIONAL_GROUP"='BOSTON')
  6 - access("D"."LOCATION_ID"="L"."LOCATION_ID")
  7 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")

21 rows selected.

```

• 3. Plano de execução

- Como são feitos os acessos no Oracle



- 1) Lê todos os registros de Location;
- 2) Para cada registro de Location busca os registros de Department;
- 3) Com o resultado da junção (nested loop) anterior faz outra junção nested loop com Employee (4).

• 3. Plano de execução

– O quê/como avaliá-lo?

- 1º **FULL SCAN (table ou INDEX)**: verifique o número de registros da tabela, avalie o percentual da tabela utilizado no processo.
- 2º **HASH JOIN**: muitas vezes são gerados pela ausência de índice apropriado.
- 3º **COST**: essa coluna indica o valor do custo da operação dentro do processo, identifique onde está o maior e ataque-o.
- 4º **FILTERS/ACCESS**: avalie se os filtros (colunas) utilizados estão de acordo com os índices.
- 5º **AVALIAÇÃO DA QUERY**: faça uma leitura no comando SQL identificando possíveis “más-práticas”.

• 4. Profiler 1/4

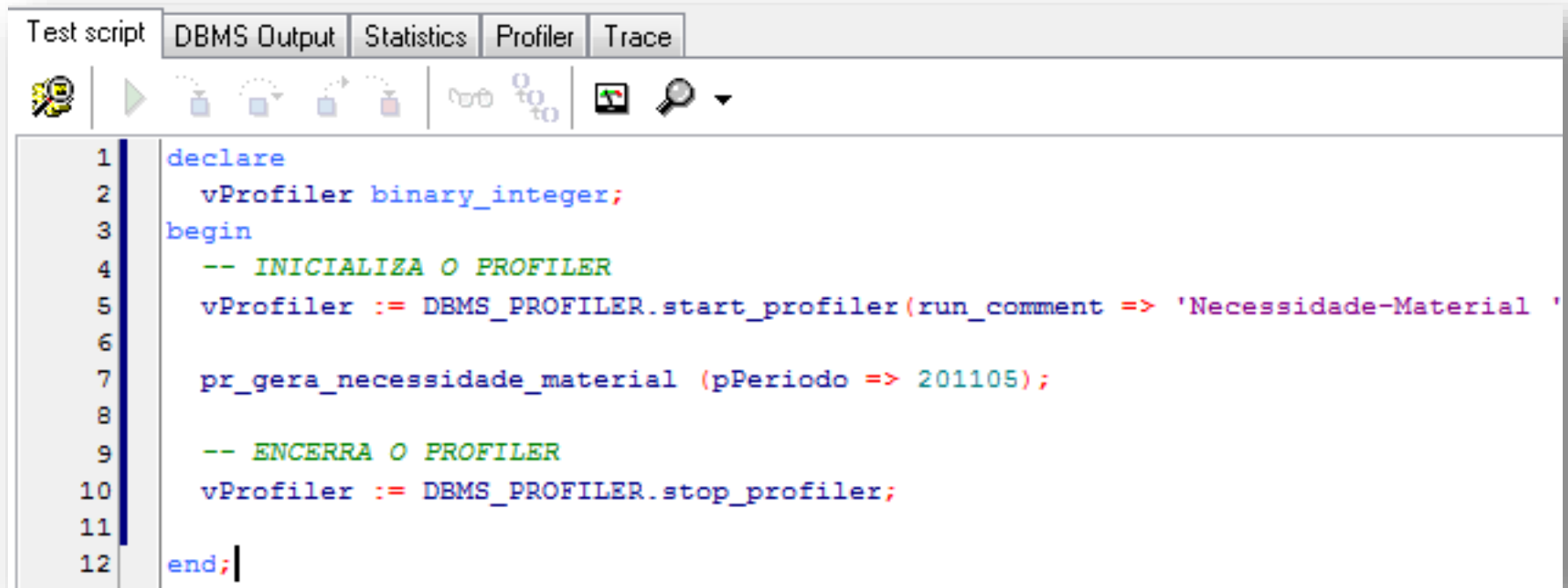
- Permite identificar qual a linha de determinado procedimento teve maior tempo durante a execução.

- Para instalação siga o guia do Oracle-Base:

http://www.oracle-base.com/articles/9i/DBMS_PROFILER.php

• 4. Profiler 2/4

- Exemplo de execução criando um profiler:



The screenshot shows the Oracle SQL Developer interface with the 'Test script' tab selected. The script contains the following SQL code:

```
1 declare
2     vProfiler binary_integer;
3 begin
4     -- INICIALIZA O PROFILER
5     vProfiler := DBMS_PROFILER.start_profiler(run_comment => 'Necessidade-Material ');
6
7     pr_gera_necessidade_material (pPeriodo => 201105);
8
9     -- ENCERRA O PROFILER
10    vProfiler := DBMS_PROFILER.stop_profiler;
11
12 end;
```

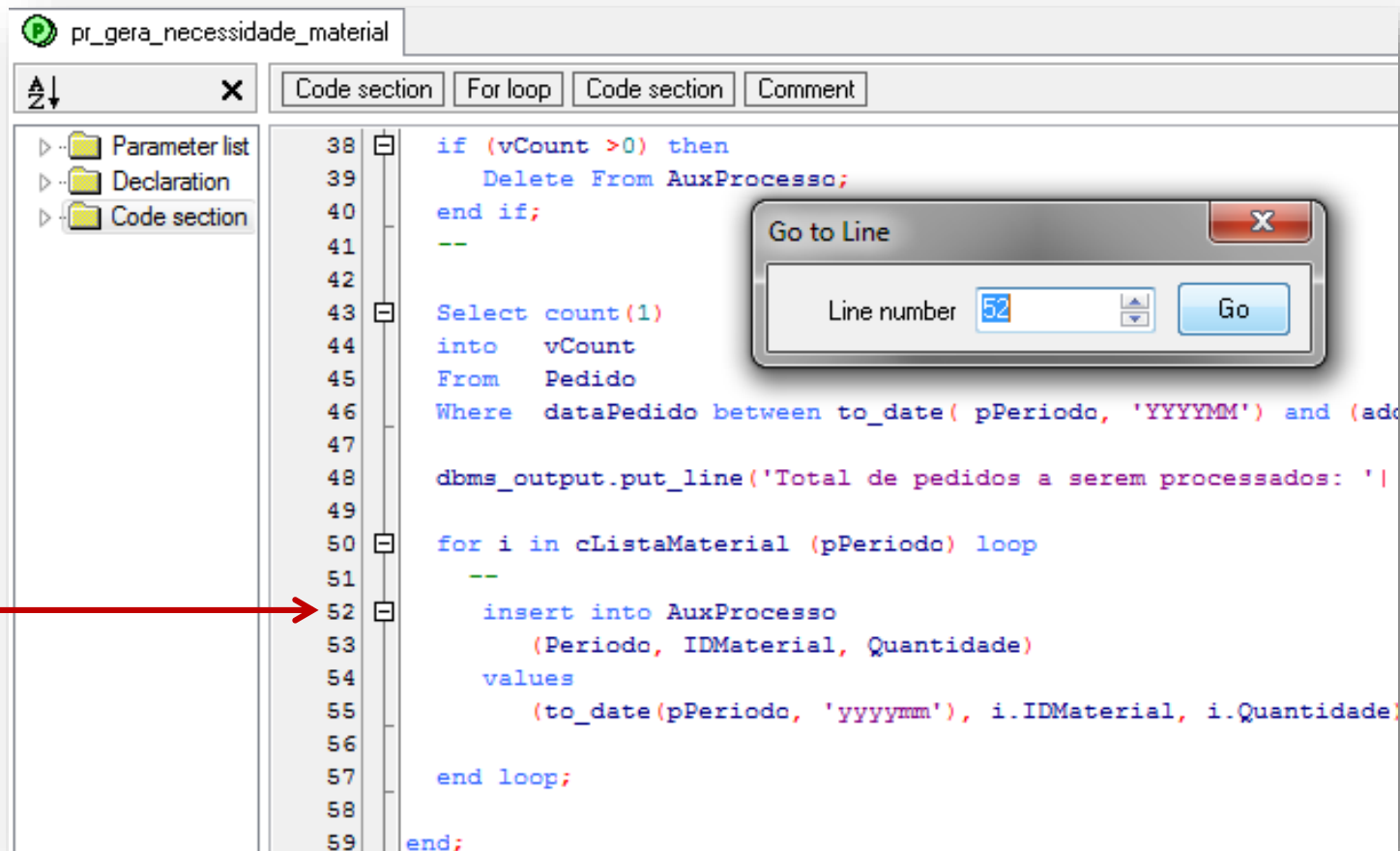
• 4. Profiler 3/4

- Consultando o resultado do profiler:

Test script DBMS Output Statistics Profiler Trace					
Run		Necessidade-Material 2011-09-19 13:40:52		Unit	
Unit	Line	Total time	Occurrences	Text	
ANONYMOUS BLOCK	7	10	1		
ANONYMOUS BLOCK	10	3	1		
PR_GERA_NECESSIDADE_MATERIAL	1	11	1	procedure pr_gera_necessidade_material (pPeriodo	
PR_GERA_NECESSIDADE_MATERIAL	3	0	1	cursor cListaProduto is	
PR_GERA_NECESSIDADE_MATERIAL	4	31	1	Select pro.IDProduto, pro.Nome	
PR_GERA_NECESSIDADE_MATERIAL	9	0	1	cursor cListaMaterial (pPeriodo in integer) is	
PR_GERA_NECESSIDADE_MATERIAL	10	101	1	select pm.idmaterial,	
PR_GERA_NECESSIDADE_MATERIAL	26	0	1	begin	
PR_GERA_NECESSIDADE_MATERIAL	28	64,636	8000	for i in cListaProduto loop	
PR_GERA_NECESSIDADE_MATERIAL	29	1,732	7998	vIDProduto := i.idproduto;	
PR_GERA_NECESSIDADE_MATERIAL	30	1,165	7998	vNome := i.nome;	
PR_GERA_NECESSIDADE_MATERIAL	31	0	1	end loop;	
PR_GERA_NECESSIDADE_MATERIAL	34	735	1	Select count(1)	
PR_GERA_NECESSIDADE_MATERIAL	38	1	1	if (vCount >0) then	
PR_GERA_NECESSIDADE_MATERIAL	39	114,080	1	Delete From AuxProcesso;	
PR_GERA_NECESSIDADE_MATERIAL	40	0	1	end if;	
PR_GERA_NECESSIDADE_MATERIAL	43	29,430	1	Select count(1)	
PR_GERA_NECESSIDADE_MATERIAL	48	30	1	dbms_output.put_line('Total de pedidos a serem pro	
PR_GERA_NECESSIDADE_MATERIAL	50	278,904	18504	for i in cListaMaterial (pPeriodo) loop	
PR_GERA_NECESSIDADE_MATERIAL	52	507,686	18502	insert into AuxProcesso	
PR_GERA_NECESSIDADE_MATERIAL	57	0	1	end loop;	
PR_GERA_NECESSIDADE_MATERIAL	59	5	1	end;	

• 4. Profiler 4/4

- Identificando linha do procedimento:



Planejamento

• 5. Desempenho: dez pontos

1- Projete e desenvolva pensando em desempenho

- Estabeleça metas de desempenho;
- Meça o desempenho o mais cedo possível (evite surpresas);
- Priorize processos críticos;
- Considere desnormalizar alguma pesquisa desde cedo e outras opções de desempenho.
 - Para relatórios armazene os dados conforme serão consultados. Faça um esforço maior para armazenar os dados de modo que o Oracle tenha o mínimo trabalho possível para buscar e apresentar os dados.

• 5. Desempenho: dez pontos

2- Otimize o ambiente de desenvolvimento

- Grande parte do SQL que executa mal no ambiente de produção tem origem em tabelas com poucos registros no ambiente de desenvolvimento;
- Tente criar um ambiente próximo do que será utilizado em produção, nas mesmas proporções de volumes para cada tabela;
- Avalie o plano de execução das consultas, se necessário utilize o TKPROF.

• 5. Desempenho: dez pontos

3- Índices inteligentes

- Indexe para suportar a seletividade utilizada nos filtros (*where*) e nos relacionamentos (*join*);
- Considere o uso de índices compostos para atacar *lookups-tables*;
- Nem sempre criar um índice resolve o problema, muitas vezes pode gerar outro problema (lentidão na atualização dos dados).

• 5. Desempenho: dez pontos

4- Reduza o *parse*

➤ Utilize Bind Variable;

- Uso de parâmetros em cursores, procedures, functions;
- SQL dinâmico também permite o uso de bind variables (DBMS_SQL e Execute Immediate).

Bind Variable - Execute immediate.tst

Bind Variable - DBMS_SQL.tst

➤ Reutilize cursores na aplicação;

- Compartilhe consultas utilizadas em mais de um processo.

• 5. Desempenho: dez pontos

5- Aproveite as vantagens do CBO

- Prefira utilizar os modos de otimização baseados em custo;
 - Utilize o pacote DBMS_STATS.
- Atualize as estatísticas regularmente
 - Tabelas pequenas: diaramente;
 - Tabelas grandes: semanalmente (a noite ou finais de semana).

• 5. Desempenho: dez pontos

6- Ataque *table scans* acidentais

- Localize acidentais leituras sequencias em tabelas, muitas vezes podem ser evitadas.
- Principais causas:
 - Falta de índices;
 - Uso de operadores de negação e diferente;
 - Comparações com valores nulos;
 - Uso de funções sobre colunas.
 - Colunas selecionadas desnecessárias.

• 5. Desempenho: dez pontos

7- Otimize *table scans* necessários

- Caso seja necessário executar uma leitura em toda a tabela procure reduzir seu tamanho;
- Realoque campos dos tipos LONG e BLOB, caso seu uso não seja frequente crie uma tabela específica;
- Evite criar muitas colunas opcionais em uma tabela;
- Considere utilizar o *fast full index scan*.

• 5. Desempenho: dez pontos

8- Otimize *joins*

- Selecione o melhor método de join
 - Nested loops são os melhores para junções indexadas de subconsultas (com relação mestre-detalle clara);
 - Hash joins são geralmente melhores para junções com grandes tabelas.
- Faça o máximo de restrições nos níveis mais internos;
- Quando possível utilize EXISTS.

• 5. Desempenho: dez pontos

9- Utilize array/vetores

- Caso o retorno de determinada consulta é utilizada mais de uma vez, prefira carregar para memória a estrutura e reutilize-a;
- Dependendo da linguagem podem ser adotados métodos diferentes;
- Verifique a possibilidade de utilizar tabelas temporárias
 - Global temporary table, possui dois tipos de persistência de dados:
 - Por sessão (preserva os dados até o fim da conexão);
 - Por transação (tem o conteúdo excluído a cada *commit* ou *rollback*).
 - Tabelas temporárias não geram *LOG* (utilizam a área temporária)

• 5. Desempenho: dez pontos

10- Considere utilizar PL/SQL para SQL complexo

- Em consultas muito complexas considere a possibilidade de utilizar blocos PL/SQL (procedure, function, etc).
- Consultas com muitas tabelas (mais de 8 acessos);
- Updates complexos, com muitas subqueries;
- Verifique a possibilidade de utilizar funções com retorno do tipo PIPE ROW.

Boas Práticas

• 6. Boas práticas – leituras lógicas

Esforce-se para eliminar as leituras lógicas

- o foco sempre esteve em eliminar/diminuir as leituras físicas (isto deve ser mantido).
- O alto número de leituras lógicas pode implicar num alto consumo de CPU também.

```
1 declare
2     vdata      date;
3     dt_inicio  date := sysdate + 1/24/60/2; -- (+30 segundos)
4 begin
5     -- VERIFICA O TEMPO DECORRIDO --
6     loop
7         select sysdate into vdata from dual;
8         if (vdata >= dt_inicio) then
9             exit;
10        end if;
11    end loop;
12 end;
```

Executará mais de
1 milhão de vezes

DBMS_LOCK.SLEEP(30);

• 6. Boas práticas – *overhead*

Hint NOCOPY em parâmetros IN OUT

- Para que o parâmetro seja por referência e não por valor.
- Em casos onde é passado um array/record ou estrutura muito grande, para evitar o *overhead* pode ser utilizado este *hint*.

```
1 DECLARE
2
3     TYPE EmpTabTyp IS TABLE OF Employees%ROWTYPE;
4     emp_tab EmpTabTyp := EmpTabTyp(NULL); -- initialize
5
6     -- sem NOCOPY
7     PROCEDURE pr_upd_emp (tab IN OUT EmpTabTyp) IS
8     BEGIN
9         -- Verificações, calculos e atribuições...
10        null;
11    END;
12
13    -- com NOCOPY
14    PROCEDURE pr_upd_emp_nc (tab IN OUT NOCOPY EmpTabTyp) IS
15    BEGIN
16        -- Verificações, calculos e atribuições...
17        null;
18    END;
```

Verificar restrições do **NOCOPY** antes de implementar.

• 6. Boas práticas - simplifique

SIMPLIFIQUE AO MÁXIMO!

- **Seja atômico:** utilize stored procedure se necessário;
- **Elimine classificações desnecessárias:** caso o resultado final da consulta não necessite de ordenação não utilize, considere substituir UNION por UNION ALL.
- **Elimine a necessidade de consultar segmentos de UNDO:** avalie as consultas e modificações, após sofrer uma alteração um registro é escrito na área de UNDO, onde fica disponível para consultas até que seja feito o commit ou rollback (mais detalhes no item DW).

• 6. Boas práticas - dbms_application_info

Identifique o processo

- **DBMS_APPLICATION_INFO:** permite informar o módulo e a ação da sessão do banco de dados.
- View v_\$session: as colunas MODULE e ACTION.
- Isto facilita o monitoramento do processo por parte dos DBA's. A cada etapa do processo a ação deve ser atualizada dentro do procedimento armazenado.

• 6. Boas práticas - Exception's

Use corretamente

- **Funções:** SEMPRE utilize o RETURN dentro do EXCEPTION de uma *Function* de banco.
- **Stored Procedures:** previna a ocorrência de erros dentro de um procedimento armazenado (procedure/package).
- **Não oculte exceptions:** NÃO oculte a ocorrência de erros. Isso pode inibir alguma situação não prevista e gerar transtorno para identificar a causa do problema na aplicação.

Gere a informação em uma tabela se necessário.

Exception
When others Then NULL;

NÃO FAÇA!!!

- **6. Boas práticas - DBMS_OUTPUT**

- Não utilize dentro de stored procedure**

- **DBMS_OUTPUT.PUT_LINE:** não utilize dentro de procedimentos armazenados, utilize apenas em ambiente de testes/desenvolvimento.

• 6. Boas práticas - triggers

Dicas

- **Especifique as colunas:** informe quais colunas deverão ser utilizadas na trigger.
- **WHEN:** utilize a validação para evitar processamento desnecessário. Exemplo: `When (IDCliente is null)`
- **Constraints:** para validações onde uma CHECK CONSTRAINT ou FOREIGN KEY pode ser utilizada prefira-as ao invés da trigger.
- **Regras de negócio:** evite utilizar regras de negócio dentro da trigger.

• 6. Boas práticas - cursores

Prefira cursor IMPLÍCITO

- **SELECT-INTO:** quando um SELECT-INTO atender a solicitação prefira-o, ao invés de utilizar um cursor OPEN-FETCH-CLOSE.
- **FOR-LOOP:** prefira utilizar um cursor com FOR-LOOP ao invés de OPEN-FETCH-CLOSE. Cursores explícito consome mais recursos do banco de dados.
- **CLOSE CURSOR:** se for utilizado o OPEN-FETCH cursor, NUNCA esqueça de fechá-lo, para liberar memória.

• 7. Dica #1

– Algumas alternativas para determinadas operações:

NÃO FAÇA	FAÇA
<pre>SELECT account_name, trans_date, amount FROM transaction WHERE SUBSTR(account_name,1,7) = 'CAPITAL';</pre>	<pre>SELECT account_name, trans_date, amount FROM transaction WHERE account_name LIKE 'CAPITAL%';</pre>
<pre>SELECT account_name, trans_date, amount FROM transaction WHERE account_name = NVL (:acc_name, account_name);</pre>	<pre>SELECT account_name, trans_date, amount FROM transaction WHERE account_name LIKE NVL (:acc_name, '%');</pre>
<pre>SELECT account_name, trans_date, amount FROM transaction WHERE TRUNC (trans_date) = TRUNC (SYSDATE);</pre>	<pre>SELECT account_name, trans_date, amount FROM transaction WHERE trans_date BETWEEN TRUNC (SYSDATE) AND TRUNC (SYSDATE) + .99999;</pre>
<pre>SELECT account_name, trans_date, amount FROM transaction WHERE account_name account_type = 'AMEXA';</pre>	<pre>SELECT account_name, trans_date, amount FROM transaction WHERE account_name = 'AMEX' AND account_type = 'A';</pre>

• 7. Dica #2 - cálculos

- **Cálculos no where:** evite cálculos na cláusula *where* sobre colunas indexadas. Exemplo:

```
... WHERE IDCliente+1000 = :pi_Codigo
```

Transfira o cálculo para o parâmetro, mesmo que isso gere uma complexidade maior.

```
... WHERE IDCliente = :pi_Codigo-1000
```

• 7. Dica #3 - non-SARG's

- **Evite operadores non-SARG's (Search Arguments):** evite utilizar esses operadores podem suprimir a utilização de índices.
 - **Operadores de negação:** NOT, <>, !=, ...
 - **Comparação com NULL:** is not null, is null.
- Procure atacar com os comandos >= ou <=, dê preferência sempre por igualdade.
- Valores nulos não são armazenados nos índices.
- Esses tipos de operadores forçarão um SCAN sobre o índice ou tabela. Projete para evitar que isso seja utilizado no sistema. Se necessário utilize um valor “fake”.

• 7. Dica #4 - Acessos

- **Minimize o número de *lookups tables*:** tente minimizar os acessos (*update, select, insert* ou *delete*). Veja um exemplo:

Não use:

```
SELECT emp_name
FROM emp
WHERE emp_cat = (SELECT MAX (category)
                  FROM emp_categories)
AND emp_range = (SELECT MAX (sal_range)
                  FROM emp_categories)
AND emp_dept = 0020;
```

Ao invés de 2 acessos
substitua por 1
apenas, pois a regra de
seleção é a mesma.

Faça assim:

```
SELECT emp_name
FROM emp
WHERE (emp_cat, sal_range)
      = (SELECT MAX (category),
              MAX (sal_range)
          FROM emp_categories)
AND emp_dept = 0020;
```


• 7. Dica #4 - Acessos

(continuação) considere utilizar **DECODE** ou **CASE**:

Não use:

```
SELECT COUNT(*)  
FROM emp  
WHERE status = 'Y'  
AND emp_name LIKE 'SMITH%';  
-----  
  
SELECT COUNT(*)  
FROM emp  
WHERE status = 'N'  
AND emp_name LIKE 'SMITH%';
```

Faça assim:

```
SELECT COUNT(DECODE(status, 'Y', 'X', NULL)) Y_count,  
       COUNT(DECODE(status, 'N', 'X', NULL)) N_count  
FROM emp  
WHERE emp_name LIKE 'SMITH%';
```

• 7. Dica #4 - Acessos

(continuação) considere utilizar **Returning** (disponível desde a versão 9i) em Insert, Update ou Delete ao invés de fazer um Select após a operação:

Não use:

```
UPDATE Sales_Order
SET     Total      = Total * 1.5
WHERE   Order_ID = 555;
--
SELECT Total
INTO    vTotal
FROM    Sales_Order
WHERE   Order_ID = 555;
```

Faça assim:

```
UPDATE    Sales_Order
SET       Total      = Total * 1.5
WHERE     Order_ID = 555
RETURNING Total INTO vTotal;
```

• 7. Dica #5 - Acessos desnecessários

- **Views:** evite reutilizar *views*, crie consultas específicas para a necessidade.
- **Acessos desnecessários:** avalie SEMPRE se todas as tabelas e colunas são realmente necessárias em uma consulta, na mesma forma que as colunas no SELECT.
- **Arrays (PL/Tables) ou Temporary Table:** utilize *array* quando possível para evitar acessos desnecessários ao banco. Porém, sempre limpe a estrutura no final de execução, para liberar memória.

• 7. Dica #6 - Verificação de existência

- **Utilize EXISTS:** para verificações de existências prefira utilizar *EXISTS* ao invés de *INNER JOIN*. Implicitamente é verificado apenas a primeira linha (rownum=1) neste operador.
- Prefira utilizar EXISTS em relação ao IN onde a lista pode trazer valores repetidos.

• 7. Dica #7 - Datas

- **Colunas para armazenar valores de data:** evite utilizar *varchar2* para armazenar data, se não for possível utilizar date utilize Number.

Exemplo: 20090125 (YYYYMMDD) - 25 de janeiro de 2009.

Isto facilita a ordenação e permite consultar sem utilizar o TO_DATE na coluna.

• 7. Dica #8 - Parâmetros

- **Bind Variables:** procure utilizar parâmetros nas queries, para evitar que seja feito o *parse* a cada chamada.

Exemplo: `select colunaA From TabelaA where IDTabelaA = :pID.`

Com isso o plano de execução é reaproveitado.

Mesmo com o uso de SQL Dinâmico é possível utilizar de BIND VARIABLES.

• 7. Dica #9 - Tipos de dados

- **Tipo correto de dados:** utilize sempre o tipo correto de dados nos filtros, para evitar conversões automáticas.

Tipo da Coluna	Filtro	Filtro Aplicado (após conversão implícita)	Usará Índice
Number	Where Emp_ID = '123'	Where Emp_ID = 123	Sim
Varchar2	Where TipoEmp = 123	Where TO_NUMBER(TipoEmp) = 123	Não

OBS.: caso o conteúdo da coluna **TipoEmp** não tenha somente números ocorrerá erro.

Conversao Implicita.sql

• 7. Dica #10

- **Evite DISTINCT:** procure alternativas para este operador, como o EXISTS, por exemplo.

Não use:

```
SELECT DISTINCT dept_no, dept_name
FROM dept D,
     emp E
WHERE D.dept_no = E.dept_no;
```

Faça assim:

```
SELECT dept_no, dept_name
FROM dept D
WHERE EXISTS ( SELECT 1
                FROM emp E
                WHERE E.dept_no = D.dept_no );
```


Dúvidas



andrenunes@cwi.com.br