# Otimização Em Banco De Dados Oracle Através do Uso de Views Materializadas

**Frederico Dutra Vieira, Marco Antônio Pereira Araújo**

Centro de Ensino Superior de Juiz de Fora – Juiz de Fora – MG – Brazil

fredim.vieira@gmail.com, marco.araujo@pucminas.cesjf.br

***Abstract.*** *There is a great concern about system performance, regardless programming language or database of choice. Several factors may prevent good system performance such as bad coding, high network or hardware usage, poorly written queries and bad choices of configuration in a specific scenario. Any performance increase originated from a single process may be vital in order to make a feature work, specially when those are related to issuing reports. These are usually systems that rely on many SQL queries, becoming source of several database access. Whenever possible, optimization techniques should be employed in queries in order to gain better performance rates. This article is an introduction to Materialized View, a largely employed technique for databases that use tables or views fetching amidst high data volume.*

***Resumo.*** *Independente do banco de dados ou linguagem definida para o desenvolvimento de uma aplicação há uma preocupação muito grande com o desempenho do sistema. Muitos fatores podem interferir no desempenho da aplicação, como código mal escrito, alto consumo de rede ou hardware, consultas mal elaboradas ou configuração não adequada para cenário necessário para aquela aplicação. Qualquer ganho gerado em algum processo do sistema pode ser vital para viabilizar alguma funcionalidade da aplicação, principalmente quando associadas a emissão de relatórios. Normalmente são sistemas que trabalham muito com consultas SQL, realizando muito acesso ao banco de dados. Sempre que possível, técnicas de otimização devem ser aplicadas nas consultas para que o desempenho do SQL seja melhorado. Este artigo busca detalhar uma técnica muito utilizada em banco de dados para obter um melhor desempenho em sistemas que utilizam tabelas ou view de banco de dados com alto volume de dados, que é a View Materializada.*

**Palavras-chave:** Banco de Dados, SQL, Otimização, *View* Materializada.

## 1. Introdução

Cada vez mais a preocupação pela otimização das aplicações vem sendo explorada pelos profissionais da Tecnologia da Informação (TI). Com o aumento da utilização de sistemas de informação nas empresas, há também o crescimento dos sistemas

corporativos e do número de usuários, crescendo o número de dados e transações concorrentes no banco de dados (SARIN, 2000). Com isso, os profissionais de TI devem implementar técnicas para garantir o desempenho na recuperação ou atualização de dados. Podendo exemplificar com os Sistemas de Apoio a Decisão (SAD), se os usuários não obtiverem a informação rápida para auxiliar na decisão para o problema em questão, o sistema não está dando o apoio necessário para seu cliente.

Para melhorar o desempenho do funcionamento do sistema, é necessário considerar muitos fatores, como a arquitetura do sistema, o *browser* (caso seja um sistema Web), a rede, o *hardware* do cliente, o servidor e o banco de dados.

Como o objetivo deste trabalho é justamente explanar sobre técnicas para obter um melhor desempenho através da utilização do banco de dados, serão abordadas algumas das principais técnicas para um melhor rendimento na execução do *Structured Query Language* (SQL) no banco de dados Oracle.

Este trabalho encontra-se estruturado em mais quatro seções, além desta introdução. A seção 2 descreve as principais técnicas de otimização em banco de dados Oracle. A seção 3 aborda a *materialized view*, mostrando como criá-la, e sua utilidade quando utilizados bancos de dados. Já na seção 4 é feita uma análise de desempenho das técnicas listadas. Na seção 5 são apresentadas as considerações finais.

## 2. Otimização em banco de dados

A otimização consiste em aplicar técnicas a fim de obter as mesmas informações com o menor tempo possível.

Para melhorar o desempenho no banco de dados, técnicas de otimização podem ser aplicadas em todo o ambiente do banco, desde a correção de um *join* até o aumento de memória do servidor onde o banco de dados está sendo executado. Mesmo que se possam obter ganhos na execução das consultas ou de rendimento geral dos sistemas que utilizam o banco de dados, não serão abordadas as técnicas que se aplicam diretamente na execução do SQL neste trabalho. O gráfico da **Figura 1** pode ajudar a entender um pouco melhor esse motivo, pois detalha a média dos problemas de desempenho em banco de dados, por área de atuação (PRADO, 2012).

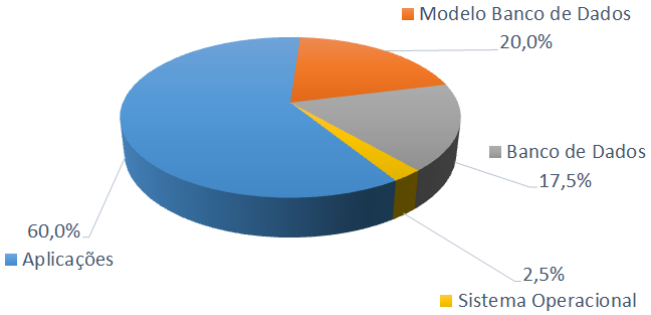



**Figura 1. Média percentual dos problemas de desempenho, por área de atuação (PRADO, 2012)**

Considerando que o item “Aplicações” é o que possui o maior percentual, entre os quatro itens analisados, é justamente nessa área que serão aplicadas as técnicas de

otimização, pois o ganho no desempenho dos sistemas pode ser mais considerável, que é na diminuição do tempo de execução do SQL.

As instruções SQL representam o maior volume de requisição ao Sistema de Gerenciamento de Banco de Dados (SGBD) e se essas instruções não tiverem bem estruturas, pode acarretar numa redução geral de desempenho do banco de dados. Muitas vezes o SQL é mal desenvolvido e não alcançam o resultado desejado e são vários os fatores que contribuem nesse quesito, bem como a falta de experiência no desenvolvimento de SQL, o curto prazo para o desenvolvimento da instrução ou até mesmo a busca somente pelo resultado do SQL. Muitos desenvolvedores não se preocupam com o desempenho da instrução gerada, não estudando ou aplicando técnicas para que o SQL obtenha os caminhos mais eficientes para sua execução (CARNEIRO et al., 2009) (SANTOS e SILVA, 2013).

O tempo de execução do SQL não deve ser o único motivador para um trabalho de otimização em banco de dados. Pode-se citar também a redução na concorrência de acesso aos dados, a otimização na taxa de transferência dos dados e a redução no tempo de criação e alteração dos dados.

Grande parte das técnicas apresentadas neste trabalho também podem ser aplicadas em outros SGBDs, entretanto, é necessário considerar que a sintaxe utilizada nas instruções SQL ou os comandos do banco de dados podem ser diferentes (PRADO, 2012).

## 2.1. Análise do plano de execução

Quando se realiza um trabalho de otimização em instruções SQL, a primeira etapa é a análise do plano de execução. O plano de execução fornece informações que, se bem analisadas, permite avaliar o desempenho do SQL e verificar se realmente é necessário realizar alguma otimização.

O plano de execução é uma sequência de passos e operações que o SGBD deverá realizar para executar um SQL. É exibido em linhas, detalhando todas as etapas que o banco de dados necessitou para executar a instrução passada, e que contém as seguintes informações (PRADO, 2012):

- ordenação das tabelas referenciadas pela instrução;
- método de acesso para cada tabela mencionada;
- método de ligação (*join*) para as tabelas afetadas nas operações;
- operações de dados tais como filtro, ordenação ou agregação;
- custo e cardinalidade da operação;
- conjunto de tabelas particionadas acessadas;
- se houve processamento paralelo.

No caso do Oracle, o plano de execução pode ser acessado através do comando *EXPLAIN PLAN*, junto da instrução SQL desejada para a análise. Esse comando será interpretado pelo SGBD e irá chamar o *query optimizer* (otimizador de *query*), que irá analisar e escolher o melhor plano para executar o SQL, e irá inserir, após a execução da

*query,* os dados descrevendo o plano escolhido na tabela *PLAN_TABLE*, sendo necessária a execução de uma *query* para buscar esses dados inseridos na tabela. Na **Figura 2** pode ser visualizado o exemplo de como gerar e consultar o plano de execução no banco de dados Oracle.

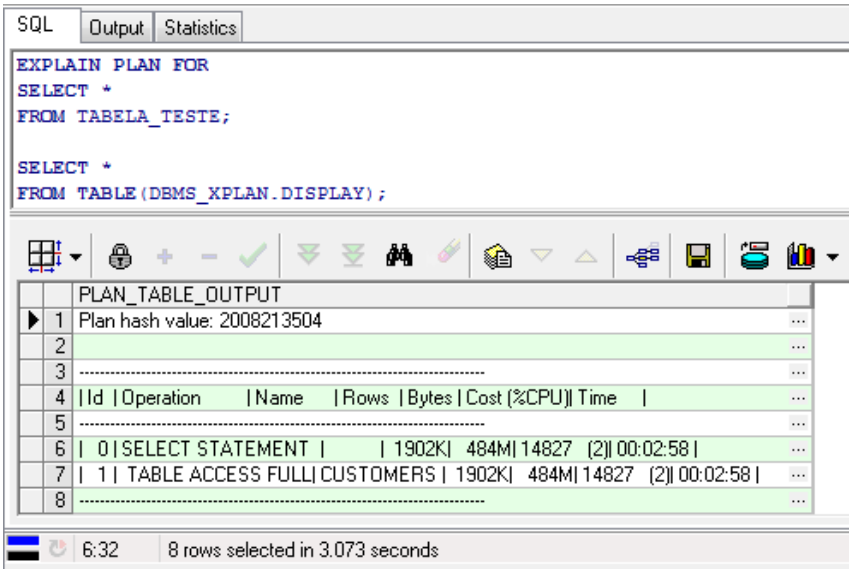



**Figura 2. Exemplo de criação do plano de execução no SGBD Oracle (do autor, 2015)**

O plano de execução pode ter sua sequência de passos alteradas de acordo com o ambiente em que a instrução SQL esteja sendo executada, e o *query optimizer* poderá escolher diferentes plano de execução de acordo com as configurações de ambiente do SGBD, como por exemplo, as configurações de *hardware*.

## 2.2. Utilização de índices

Índice é uma estrutura, opcional, muito utilizada em todos os SGBDs existentes no mercado, auxilia na melhoria de desempenho de consultas SQL. Gera uma lista ordenada com a posição de cada registro na tabela ou coleção de dados, fazendo com que a ligação entre essas tabelas seja realizada mais rapidamente.

No SGBD Oracle podem ser criados quatro grupos de índices diferentes para obter ganho de desempenho nas consultas. A seguir será apresentado um pouco das características de cada uma.

### 2.2.1. *B-Tree*

Este índice vem sendo utilizado no SGBD Oracle desde suas as primeiras versões, e é considerado o índice padrão. Através dele, o Oracle gerencia corretamente os blocos de dados, controlando a alocação dos ponteiros dentro de cada bloco. A criação desse índice é recomendada quando se torna necessária a execução de consultas de colunas com alta cardinalidade. Como por exemplo, colunas em que há unicidade de valores, como as chaves primárias (PK – *Primary Key*) de uma tabela (VALIATI, 2007). A **Figura 3** apresenta um exemplo desse índice.

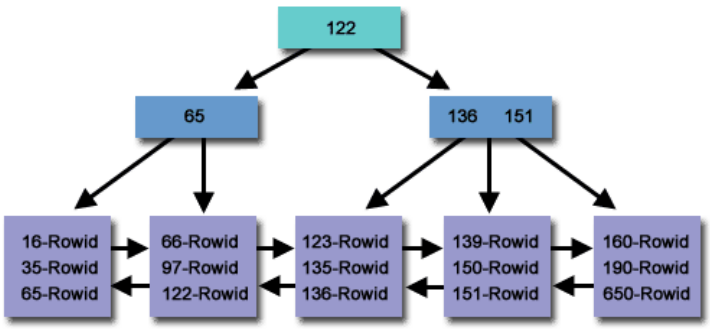



**Figura 3. Índice *B-Tree* (VALIATI, 2007)**

### 2.2.2. *Bitmap*

A criação do índice *Bitmap* é recomendada em colunas com baixa cardinalidade. Não sendo recomendada sua utilização em colunas com alterações de dados frequentes, inclusive o Oracle não permite a sua criação em chaves primárias (PRADO, 2012). Na **Figura 4** pode ser vista a representação gráfica do índice.

| | LINHA | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VALOR | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| MASCULINO | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| FEMININO | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

**Figura 4. Índice *Bitmap* (PRADO, 2012)**

### 2.2.3. *Function based*

Esse tipo de índice é recomendado quando há necessidade de executar consultas aplicando funções nas colunas em condições seletivas. Esses índices podem ter em sua estrutura os índices *B-Tree* ou *Bitmap* (PRADO, 2012). Um exemplo é apresentado na **Figura 5**.

SQL | Output | Statistics

```
SELECT MIN(DATA_NASCIMENTO) AS DT_NASC
FROM ALUNO
WHERE TO_CHAR(DATA_NASCIMENTO,'YYYY') > '1980';
```

**Figura 5. Índice *Function Based* (PRADO, 2012)**

### 2.2.4. *Domain indexes*

Esse índice foi projetado para trabalhar em casos especiais ou complexos, como por exemplo, em pesquisas de texto ou processamento de imagens. Para esse grupo, podem ser citados dois índices, o *Oracle Text* e o *Oracle Spatial*. O primeiro é muito recomentado na otimização de consultas de texto, onde as colunas possuem o tipo *VARCHAR2* ou *CLOB*.

Antes de realizar a criação dos índices, é importante considerar alguns pontos (PRADO, 2012):

- é recomendado criar índices apenas para otimizar as consultas mais utilizadas, pois os índices não utilizados, ou pouco utilizados, atrapalham no desempenho das atualizações de registro e ocupam espaço em disco que poderia ser usado para outro fim;
- não criar índices únicos para cada coluna. Se a consulta realizada contempla cinco colunas, por exemplo, cria-se um índice que abrange essas cinco colunas;
- não criar índices em colunas de tabelas pequenas, pois o custo para processar um *Full Table Scan* é baixo. Por isso, não há necessidade da criação de índices nesse caso. *Full Table Scan* pode ser entendia como uma consulta com uma leitura sequencial, varrendo toda a tabela especificada. Nesse caso, são verificados todos os registros com os critérios de seleção;
- se sua consulta trabalha com *joins* entre tabelas, deve-se criar o índice entre as colunas de ligação das tabelas. O índice irá otimizar o método *join* e, consequentemente, o desempenho geral da consulta;
- analise as condições de filtro ou cardinalidade dos dados. Após ter sua necessidade entendida, cria-se o tipo de índice que melhor irá atender à situação.

## 2.3. Criar ligações entre as tabelas

Quando se cria uma instrução que trabalha com duas ou mais tabelas, é necessário que sejam feitas ligações entre elas. Essas ligações em banco de dados são chamadas de *join*, e basicamente são de dois tipos: *inner join* e *outer join*.

As utilizações desses *joins* devem ser feitas com muito cuidado, pois tanto o desempenho da sua instrução SQL, quanto o seu resultado, podem ser alterados.

Buscando uma melhoria no desempenho da instrução SQL, será detalhado o funcionamento dos *joins* no banco de dados Oracle e realizada uma análise de desempenho dos diferentes tipos (SANTOS, 2014):

- *Inner join*:

  O *inner join*, ou apenas *join*, realiza a junção entre as tabelas através da verificação entre os registros das tabelas envolvidas. Se os registros forem iguais nessas tabelas e tiverem os mesmos valores, a verificação realizada será verdadeira e as tabelas terão seus valores combinados, carregando então apenas os registros das tabelas cujos seus valores pertencem simultaneamente às tabelas envolvidas. A teoria de conjuntos pode auxiliar no entendimento. Os dados carregados no resultado do SQL serão a interseção entre as duas tabelas, conforme demonstrado n **Figura 6**.

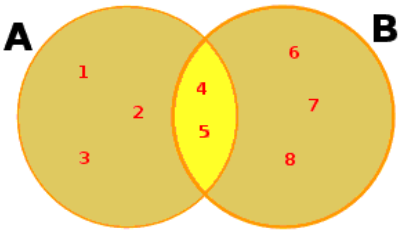



**Figura 6. *Inner join* (SANTOS, 2014)**

O *inner join* pode ser implementado de duas formas, conforme exemplo na **Figura 7**.

```
1   -- Forma 1:
2   SELECT A.ID,
3          B.DESCRICAO
4   FROM TABELA_A A,
5        TABELA_B B,
6   WHERE A.ID = B.ID;
7
8   -- Forma 2:
9   SELECT A.ID,
10         B.DESCRICAO
11  FROM TABELA_A A INNER JOIN TABELA_B B
12  ON A.ID = B.ID;
```

**Figura 7. Exemplo de implementação do *inner join* (SANTOS, 2014)**

- *Outer join:*

  O *outer join* funciona de forma similar ao *inner join*, porém há a opção de estender o resultado obtido pelo *inner join*, possibilitando que sejam carregados todos os valores de uma determinada tabela, mesmo que a combinação entre os valores das tabelas envolvidas seja falsa. Há três subdivisões para o uso do *outer join*: *left outer join*, *right outer join* e *full outer join*.

  - *Left outer join*

    O *left outer join* irá carregar todos os dados da tabela A, independente se há igualdade na comparação entre os valores da tabela B. Os valores carregados da coluna B serão informados quando os dados tiverem registros equivalentes. Quando os dados da tabela B não forem relacionados com a da tabela A, os dados referentes da tabela B ficarão sem dados a serem informados, conforme pode ser visto no exemplo da **Figura 8**.

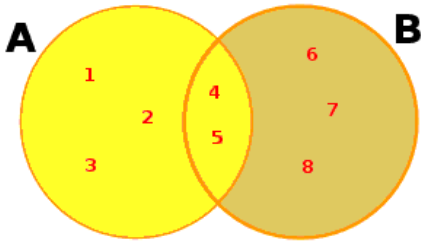



**Figura 8. *Left outer join* (SANTOS, 2014)**

Assim como o *inner join*, o *left outer join* possui duas formas de implementação, conforme exemplo na **Figura 9**.

```
1   -- Forma 1:
2   SELECT A.ID,
3          B.DESCRICAO
4   FROM TABELA_A A,
5        TABELA_B B,
6   WHERE A.ID(+) = B.ID;
7
8   -- Forma 2:
9   SELECT A.ID,
10         B.DESCRICAO
11  FROM TABELA_A A LEFT OUTER JOIN TABELA_B B
12  ON A.ID = B.ID;
```

**Figura 9. Exemplo de implementação do *left outer join* (SANTOS, 2014)**

- *Right outer join*

  O *right outer join* é exatamente igual ao *left outer join*, porém com inversão da tabela onde os dados serão carregados, independentemente se existe relação entre seus dados ou não. Na **Figura 10** pode ser visto sua representação.

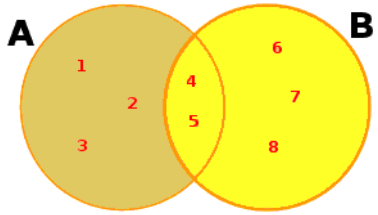



**Figura 10. *Right outer join* (SANTOS, 2014)**

Como o *right outer join* é bem similar ao *left outer join*, também possui duas formas para ser implementado, conforme apresentado na **Figura 11**.

```
1   -- Forma 1:
2   SELECT A.ID,
3          B.DESCRICAO
4   FROM TABELA_A A,
5        TABELA_B B,
6   WHERE A.ID = B.ID(+);
7
8   -- Forma 2:
9   SELECT A.ID,
10         B.DESCRICAO
11  FROM TABELA_A A RIGHT OUTER JOIN TABELA_B B
12  ON A.ID = B.ID;
```

**Figura 11. Exemplo de implementação do *right outer join* (SANTOS, 2014)**

- *Full outer join*

  Já o *full outer join* foi definido para que a junção entre as tabelas envolvidas retorne todos os registros existentes nessas tabelas. Assim como no *left* ou *right outer join*, os dados da tabela onde não há relação com a outra será carregado como nulo. A **Figura 12** exemplifica essa situação na forma de conjuntos.

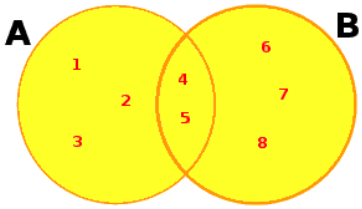



**Figura 12. *Full outer join* (SANTOS, 2014)**

Já o *full outer join* possui apenas uma forma para ser implementado, como ser visto no exemplo da **Figura 13**.

```
1  SELECT A.ID,
2         B.DESCRICAO
3  FROM TABELA_A A FULL OUTER JOIN TABELA_B B
4  ON A.ID = B.ID;
```

**Figura 13. Exemplo de implementação do *full outer join* (SANTOS, 2014)**

## 2.4. Avaliação do uso da cláusula *WITH* para minimizar a execução de um mesmo SQL

Outra possibilidade em otimização de consultas SQL em banco de dados Oracle é a cláusula *WITH*. Essa cláusula faz parte do padrão *ANSI SQL 99* e foi adicionado na sintaxe Oracle em sua versão 9i release 2. A cláusula *WITH* possibilita a reutilização de blocos de subconsultas ou tabelas que são referenciadas inúmeras vezes, a partir da criação de uma tabela temporária, que existe apenas no escopo da instrução SQL em que a cláusula está contida. É armazenada em memória ou em *tablespace* temporária, que possui um acesso mais rápido que as *tablespace* comuns.

A cláusula *WITH* pode ser usada para reduzir a repetição e simplificar instruções SQL complexas. Quando há uma consulta onde uma tabela é referenciada várias vezes, pode ser evitado que essa tabela seja executada em cada referência no SQL. A tabela temporária criada pela cláusula armazena o resultado da consulta e permite acesso aos dados somente desse novo objeto. Como a execução é feita em uma tabela apenas com os dados necessários, e já filtrada, se torna mais rápido recuperar os dados dessa forma. A **Figura 14** apresenta um exemplo de utilização dessa cláusula.

```sql
-- SQL sem a cláusula WITH
SELECT A.DESCRICAO
FROM TABELA_A A
WHERE A.CODIGO = 'C'
UNION
SELECT A.DESCRICAO
FROM TABELA_A A
WHERE A.CODIGO <> 'A';

-- SQL com a cláusula WITH
WITH CES AS (SELECT A.CODIGO,
                    A.DESCRICAO
             FROM TABELA_A A
            )
SELECT C.DESCRICAO
FROM CES C
WHERE C.CODIGO = 'C'
UNION
SELECT C.DESCRICAO
FROM CES C
WHERE C.CODIGO <> 'A';
```

**Figura 14. Exemplo de utilização da cláusula *WITH* (do autor, 2015)**

Segundo (PRADO, 2010), com a implementação dessa técnica, é possível otimizar, em média, 30% na execução de uma consulta. Na seção 4 desse trabalho será feita a análise de desempenho da cláusula.

## 2.5. Utilização de *functions*, *stored procedures* e *packages*

Segundo (PRADO, 2011) a criação de *stored procedures* gera muita polêmica. Quando se utiliza esse tipo de objeto, a maioria das regras de negócio da aplicação está sendo criada dentro do banco de dados. É uma boa prática de desenvolvimento separar a aplicação em camadas, onde cada uma tem uma função específica dentro do sistema, como, por exemplo, a camada de interface com o banco de dados e camada das regras de negócio. Mas não é o foco deste trabalho discutir ou entrar nesse assunto, mas sim em otimização de consultas SQL.

Também, segundo (PRADO, 2011), é possível reduzir 55,26% no tempo de execução de um bloco com mil transações via *stored procedure*, comparando esse mesmo bloco sendo executado via SQL através de uma aplicação.

# 3. *View* Materializada (*Materialized View* – MV)

## 3.1. Definição

MV's são objetos de banco de dados utilizados para armazenar o retorno de uma instrução SQL. É um recurso muito utilizado quando há uma necessidade de melhorar o desempenho de instruções SQL muito complexas, que trabalham com muitas tabelas e um volume de dados muito alto (OLIVEIRA, 2015).

A MV tem um funcionamento bem parecido com as visões (*views*) de banco de dados, mas existe uma diferença muito importante entre elas, e que faz toda a diferença no desempenho da execução. A *view* é um objeto criado através de uma consulta estabelecida como parâmetro e essa consulta é sempre executada quando essa *view* é chamada. Já a MV executa a consulta estabelecida como parâmetro apenas na sua criação ou atualização. Os dados contidos na MV são salvos novamente no banco, como se fosse uma tabela. Mesmo que esses dados possam ser recuperados no banco de dados através de um SQL, e não utiliza espaço do banco de dados para o mesmo registro, o uso

da MV é muito importante para ganho de desempenho em aplicações. Principalmente nos *Data Warehouses*, onde a maior preocupação está no desempenho dos relatórios e não no espaço consumido no banco de dados (MIRANDA, 2014).

## 3.2. Criação de *Views* Materializadas

Na **Figura 15** pode ser visto um exemplo da sintaxe para criação de uma MV no Oracle.

```
1  CREATE MATERIALIZED VIEW MATERIALIZED_VIEW
2  REFRESH COMPLETE ON DEMAND
3  START WITH TO_DATE('01-01-2016 00:00:00',
4                     'DD-MM-YYYY HH24:MI:SS')
5      NEXT SYSDATE + (1/24)
6  AS
7  SELECT A.ID,
8         B.DESCRICAO
9  FROM TABELA_A A,
10      TABELA_B B
11 WHERE A.ID = B.ID;
```

**Figura 15. *Script* de criação de uma *materialized view* (do autor, 2015)**

Os comandos utilizados no script da **Figura 15** funcionam basicamente da seguinte forma (ALMEIDA, 2004):

- ***CREATE MATERIALIZED VIEW***: comando inicial para a criação da MV no banco de dados. Logo após esse comando, vem o nome que esse objeto terá dentro do banco de dados. No *script* da figura, o nome adotado foi *MATERIALIZED_VIEW*;
- ***REFRESH COMPLETE***: esse comando serve para que a MV tenha seus dados recriados quando for atualizada;
- ***ON DEMAND***: serve para indicar que o banco de dados não irá atualizar a MV automaticamente. Os dados serão atualizados apenas pela execução programado (ou manualmente) através do comando *DBMS_MVIEW.REFRESH* (‘nome da MV’);
- ***AS SELECT***: nessa etapa é passada a instrução SQL que irá popular os dados existentes na MV.

## 4. Análise de Desempenho

Para a realização da análise de desempenho deste trabalho, foi criado um cenário com dados aleatórios, sendo que foi dada uma carga em cada tabela com cerca de 3 milhões de registros. Na **Figura 16** pode ser analisado o modelo relacional do banco de dados criado, detalhando seus campos, chaves primárias (PK) e estrangeiras (FK).

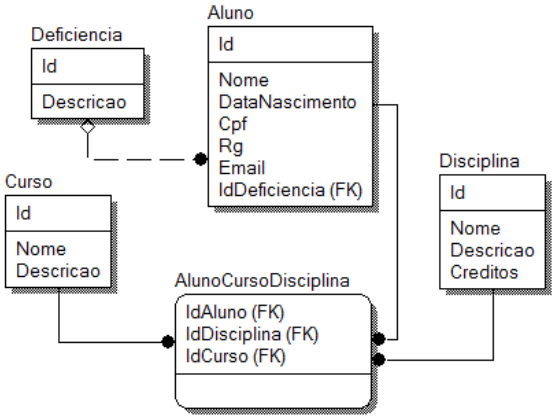



**Figura 16. Modelo relacional utilizado no estudo (do autor, 2015)**

Neste trabalho foram citadas algumas técnicas para auxiliar na melhoria do desempenho da execução de instruções SQL em banco de dados Oracle, mas até agora não foi mostrado na prática os ganhos que cada técnica pode proporcionar.

Será exemplificado, através de alguns exemplos, os custos de processamento e tempo de execução de um mesmo SQL, para cada técnica aprestada no trabalho, exceto para a técnica de *stored procedure*, pois não é possível obter o plano de execução da mesma.

Nas **Figuras 17** e **18,** o SQL está sendo executado sem nenhum índice, e a instrução é executada em 25,026 segundos, com um custo de processamento de 47355.

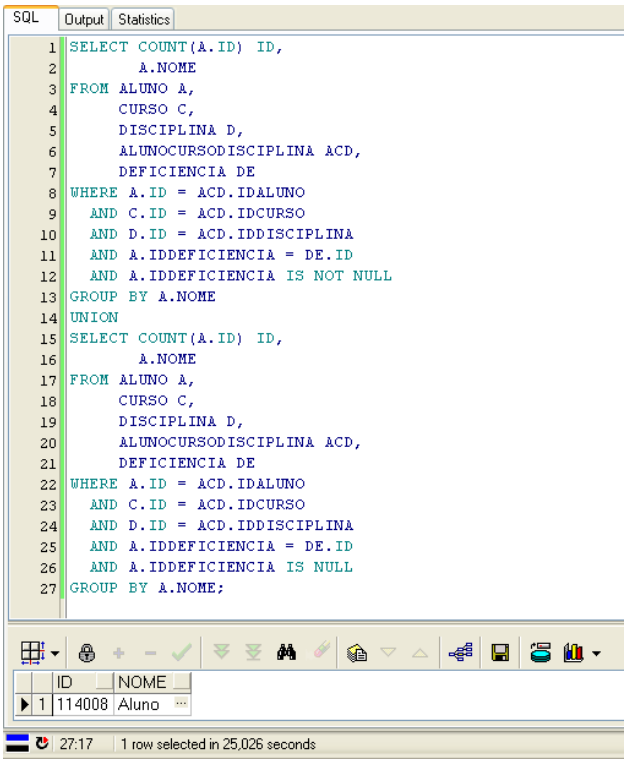



**Figura 17. Tempo de execução do SQL sem índice (do autor, 2015)**

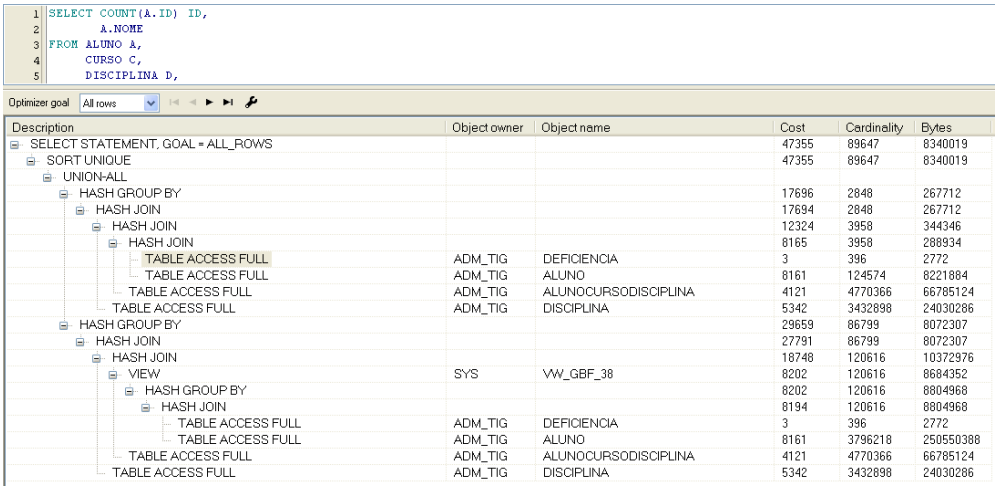



**Figura 18. Custo de processamento do SQL sem índice (do autor, 2015)**

Com a criação do índice foi possível reduzir o tempo de execução em 9,72%. O SQL foi processado com 22,593 segundos, conforme a **Figura 19**, e o custo de processamento de 30107, como pode ser visto no plano de execução na **Figura 20**, já com os índices criados.

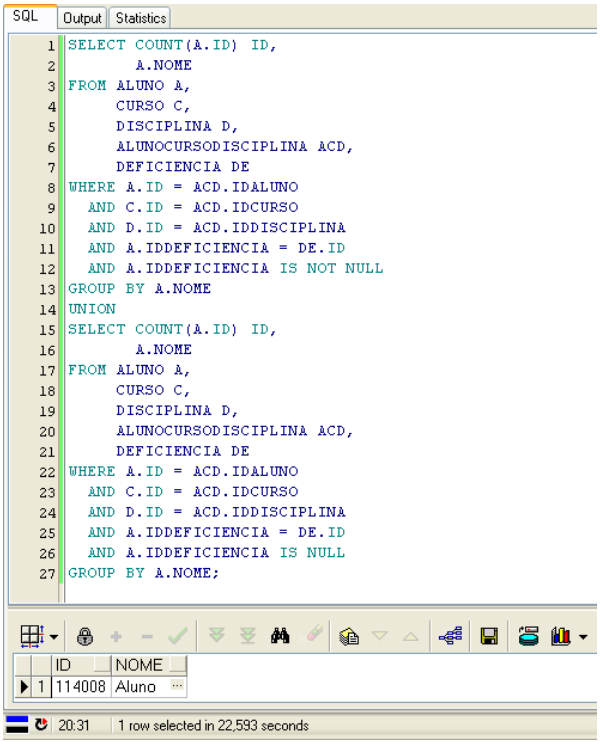
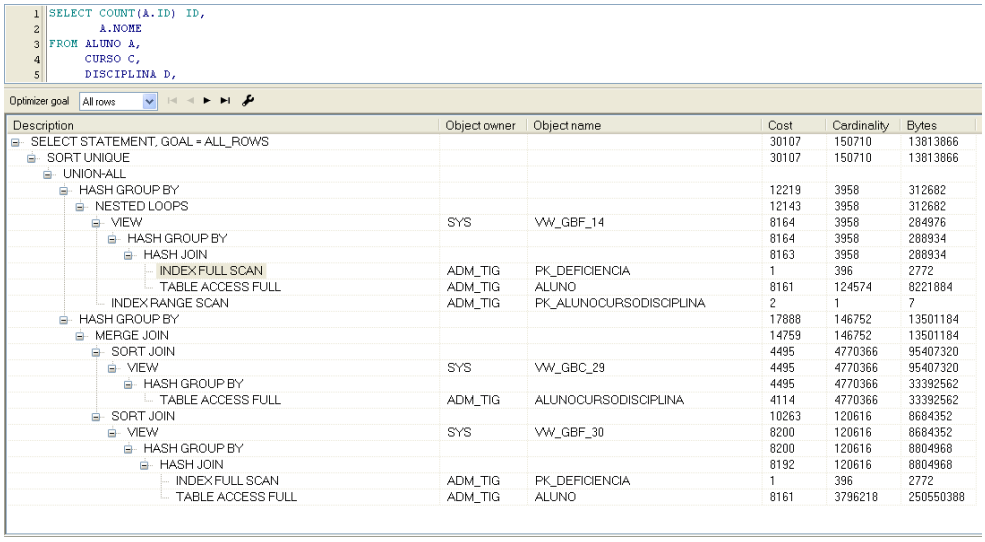



**Figura 19. Tempo de execução do SQL com índice (do autor, 2015)**



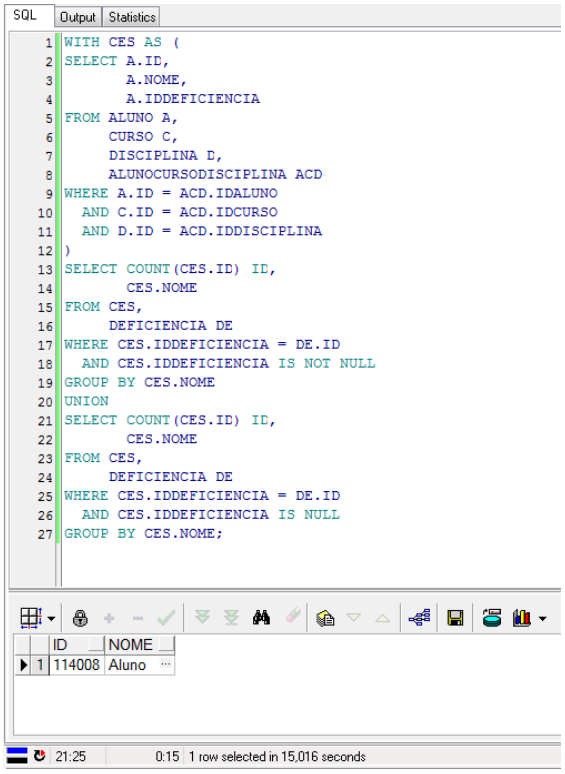
```
SELECT COUNT(A.ID) ID,
       A.NOME
FROM ALUNO A,
     CURSO C,
     DISCIPLINA D,
```

Optimizer goal: All rows

| Description | Object owner | Object name | Cost | Cardinality | Bytes |
|---|---|---|---|---|---|
| SELECT STATEMENT, GOAL = ALL_ROWS | | | 30107 | 150710 | 13813866 |
| SORT UNIQUE | | | 30107 | 150710 | 13813866 |
| UNION-ALL | | | | | |
| HASH GROUP BY | | | 12219 | 3958 | 312682 |
| NESTED LOOPS | | | 12143 | 3958 | 312682 |
| VIEW | SYS | VW_GBF_14 | 8164 | 3958 | 284976 |
| HASH GROUP BY | | | 8164 | 3958 | 288934 |
| HASH JOIN | | | 8163 | 3958 | 288934 |
| INDEX FULL SCAN | ADM_TIG | PK_DEFICIENCIA | 1 | 396 | 2772 |
| TABLE ACCESS FULL | ADM_TIG | ALUNO | 8161 | 124574 | 8221884 |
| INDEX RANGE SCAN | ADM_TIG | PK_ALUNOCURSODISCIPLINA | 2 | 1 | 7 |
| HASH GROUP BY | | | 17888 | 146752 | 13501184 |
| MERGE JOIN | | | 14759 | 146752 | 13501184 |
| SORT JOIN | | | 4495 | 4770366 | 95407320 |
| VIEW | SYS | VW_GBC_29 | 4495 | 4770366 | 95407320 |
| HASH GROUP BY | | | 4495 | 4770366 | 33392562 |
| TABLE ACCESS FULL | ADM_TIG | ALUNOCURSODISCIPLINA | 4114 | 4770366 | 33392562 |
| SORT JOIN | | | 10263 | 120616 | 8684352 |
| VIEW | SYS | VW_GBF_30 | 8200 | 120616 | 8684352 |
| HASH GROUP BY | | | 8200 | 120616 | 8804968 |
| HASH JOIN | | | 8192 | 120616 | 8804968 |
| INDEX FULL SCAN | ADM_TIG | PK_DEFICIENCIA | 1 | 396 | 2772 |
| TABLE ACCESS FULL | ADM_TIG | ALUNO | 8161 | 3796218 | 250550388 |

**Figura 20. Custo de processamento do SQL com índice (do autor, 2015)**

Nas **Figura 21** e **22**, pode ser analisado que, após a inclusão da cláusula *WITH*, o custo de processamento e o tempo de execução foram reduzidos, melhorando a execução do SQL, já com os índices criados. O tempo de execução nesse caso foi reduzido em 39,99%, passando para 15,016 segundos.



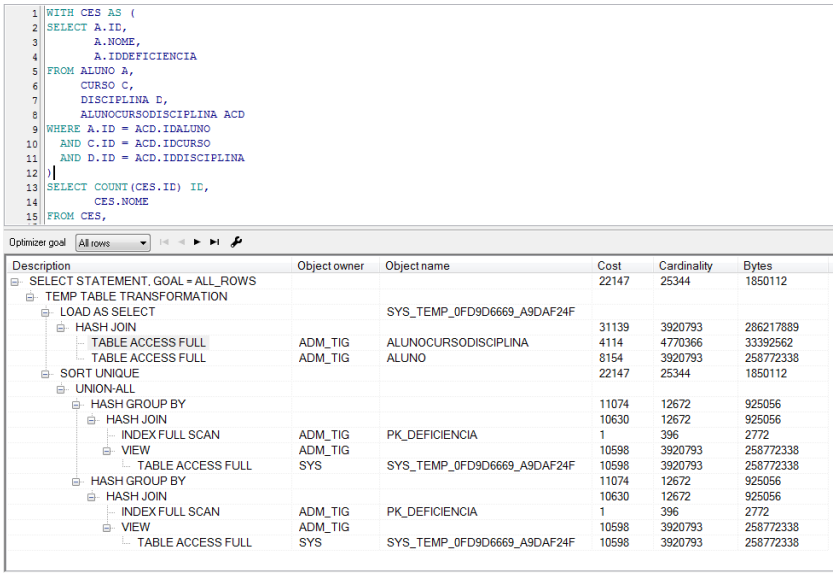
SQL | Output | Statistics

```
WITH CES AS (
SELECT A.ID,
       A.NOME,
       A.IDDEFICIENCIA
FROM ALUNO A,
     CURSO C,
     DISCIPLINA D,
     ALUNOCURSODISCIPLINA ACD
WHERE A.ID = ACD.IDALUNO
  AND C.ID = ACD.IDCURSO
  AND D.ID = ACD.IDDISCIPLINA
)
SELECT COUNT(CES.ID) ID,
       CES.NOME
FROM CES,
     DEFICIENCIA DE
WHERE CES.IDDEFICIENCIA = DE.ID
  AND CES.IDDEFICIENCIA IS NOT NULL
GROUP BY CES.NOME
UNION
SELECT COUNT(CES.ID) ID,
       CES.NOME
FROM CES,
     DEFICIENCIA DE
WHERE CES.IDDEFICIENCIA = DE.ID
  AND CES.IDDEFICIENCIA IS NULL
GROUP BY CES.NOME;
```

| | ID | NOME |
|---|---|---|
| 1 | 114008 | Aluno |

21:25 | 0:15 | 1 row selected in 15,016 seconds

**Figura 21. Tempo de execução do SQL com *WITH* (do autor, 2015)**



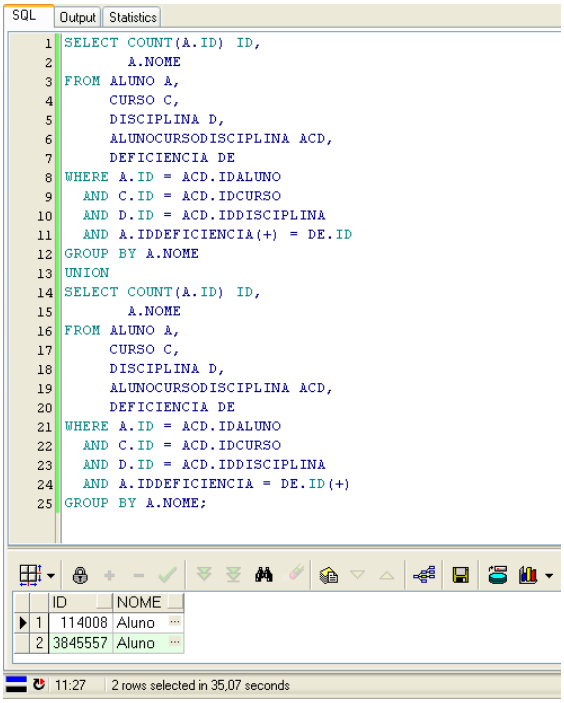
```
WITH CES AS (
SELECT A.ID,
       A.NOME,
       A.IDDEFICIENCIA
FROM ALUNO A,
     CURSO C,
     DISCIPLINA D,
     ALUNOCURSODISCIPLINA ACD
WHERE A.ID = ACD.IDALUNO
  AND C.ID = ACD.IDCURSO
  AND D.ID = ACD.IDDISCIPLINA
)
SELECT COUNT(CES.ID) ID,
       CES.NOME
FROM CES,
```

Optimizer goal: All rows

| Description | Object owner | Object name | Cost | Cardinality | Bytes |
|---|---|---|---|---|---|
| SELECT STATEMENT, GOAL = ALL_ROWS | | | 22147 | 25344 | 1850112 |
| TEMP TABLE TRANSFORMATION | | | | | |
| LOAD AS SELECT | | SYS_TEMP_0FD9D6669_A9DAF24F | | | |
| HASH JOIN | | | 31139 | 3920793 | 286217889 |
| TABLE ACCESS FULL | ADM_TIG | ALUNOCURSODISCIPLINA | 4114 | 4770366 | 33392562 |
| TABLE ACCESS FULL | ADM_TIG | ALUNO | 8154 | 3920793 | 258772338 |
| SORT UNIQUE | | | 22147 | 25344 | 1850112 |
| UNION-ALL | | | | | |
| HASH GROUP BY | | | 11074 | 12672 | 925056 |
| HASH JOIN | | | 10630 | 12672 | 925056 |
| INDEX FULL SCAN | ADM_TIG | PK_DEFICIENCIA | 1 | 396 | 2772 |
| VIEW | ADM_TIG | | 10598 | 3920793 | 258772338 |
| TABLE ACCESS FULL | SYS | SYS_TEMP_0FD9D6669_A9DAF24F | 10598 | 3920793 | 258772338 |
| HASH GROUP BY | | | 11074 | 12672 | 925056 |
| HASH JOIN | | | 10630 | 12672 | 925056 |
| INDEX FULL SCAN | ADM_TIG | PK_DEFICIENCIA | 1 | 396 | 2772 |
| VIEW | ADM_TIG | | 10598 | 3920793 | 258772338 |
| TABLE ACCESS FULL | SYS | SYS_TEMP_0FD9D6669_A9DAF24F | 10598 | 3920793 | 258772338 |

**Figura 22. Custo de processamento do SQL com *WITH* (do autor, 2015)**

Nas **Figura 23** e **24** é possível analisar que a ligação entre as tabelas pode alterar completamente o tempo de execução e o custo de processamento do SQL no banco de dados. Alterando o método de ligação entre as tabelas de *inner join* para o *left outer join* e *right outer join,* aumentou em 40,13% o tempo de execução da instrução, passando o tempo de execução para 35,07 segundos, e o custo de processamento para 109985.



SQL | Output | Statistics

```
SELECT COUNT(A.ID) ID,
       A.NOME
FROM ALUNO A,
     CURSO C,
     DISCIPLINA D,
     ALUNOCURSODISCIPLINA ACD,
     DEFICIENCIA DE
WHERE A.ID = ACD.IDALUNO
  AND C.ID = ACD.IDCURSO
  AND D.ID = ACD.IDDISCIPLINA
  AND A.IDDEFICIENCIA(+) = DE.ID
GROUP BY A.NOME
UNION
SELECT COUNT(A.ID) ID,
       A.NOME
FROM ALUNO A,
     CURSO C,
     DISCIPLINA D,
     ALUNOCURSODISCIPLINA ACD,
     DEFICIENCIA DE
WHERE A.ID = ACD.IDALUNO
  AND C.ID = ACD.IDCURSO
  AND D.ID = ACD.IDDISCIPLINA
  AND A.IDDEFICIENCIA = DE.ID(+)
GROUP BY A.NOME;
```

| | ID | NOME |
|---|---|---|
| 1 | 114008 | Aluno |
| 2 | 3845557 | Aluno |

11:27 2 rows selected in 35,07 seconds

**Figura 23. Tempo de execução do SQL com *left/righ outer join* (do autor, 2015)**

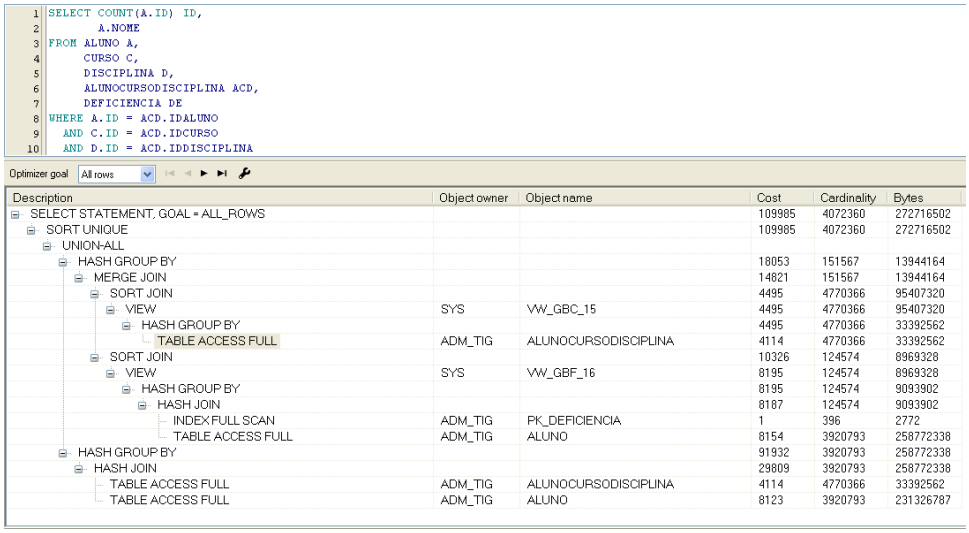



**Figura 24. Custo de processamento do SQL com *left/right outer* (do autor, 2015)**

Nas **Figuras 25** e **26** fica evidenciada a grande redução no tempo de execução da instrução SQL e do custo de processamento quando se trabalha com *materialized views*. O mesmo resultado que se obtinha com 25,026 segundos pôde ser obtido com apenas 0,01 segundo, uma redução de 99,96%.

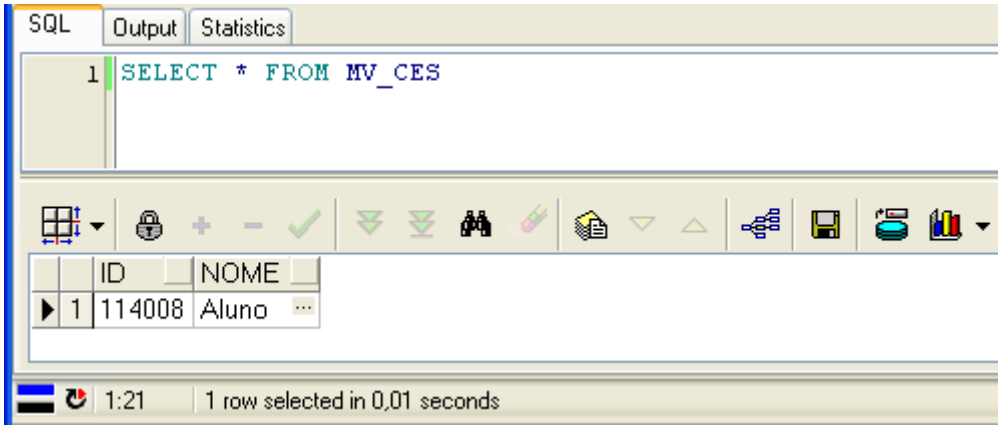



**Figura 25. Tempo de execução da *materialized view* (do autor, 2015)**

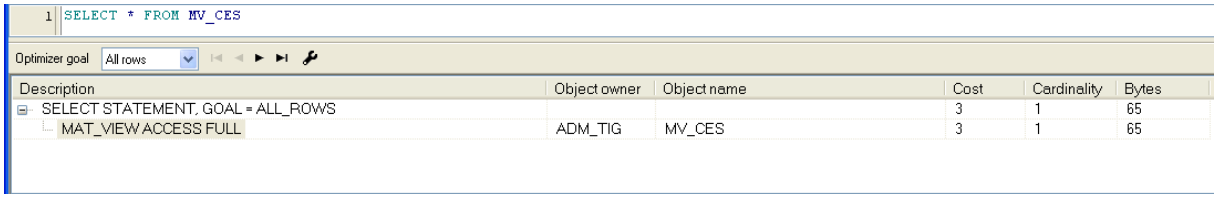



**Figura 26. Custo de processamento da *materialized view* (do autor, 2015)**

## 5. Conclusão

Com o crescimento do uso de sistemas nas organizações, o aumento dos dados e fluxo de informação crescem também dentro das empresas. A busca pela melhoria no desempenho das aplicações é constante, mas para otimizar o desempenho da aplicação há a necessidade de conhecimentos avançados em otimização em diversas áreas de

atuação, bem como em sistemas operacionais, redes de computadores e bancos de dados. Este trabalho teve o objetivo de esclarecer algumas técnicas utilizadas para o ganho de desempenho na execução de SQL em bancos de dados. Ao diminuir o tempo de processamento da instrução SQL no banco de dados, diminui a concorrência no SGBD e o tráfego de rede, podendo melhorar indiretamente no desempenho de outras aplicações.

Todas as técnicas explanadas neste trabalho possui pontos negativos e é muito importante utilizá-las após análise do ambiente e conhecimento da técnica. É aconselhado que primeiramente seja feita uma análise do plano de execução da instrução SQL que deseja realizar a otimização. Através do plano de execução pode ser visto que a lentidão não esteja na execução do SQL ou que aponte onde é necessário aplicar uma determinada técnica de otimização, como por exemplo a criação de um índice. Como pôde ser visto neste trabalho, a criação de índice pode reduzir o tempo e custo da execução do SQL, porém isso gera um custo para o SGBD. Os índices consomem espaço no banco de dados e devem ser usados apenas quando necessário. Além do que, os índices podem sofrer quebras em tabelas onde os registros são constantemente deletados e isto gera uma necessidade de monitoramento. As *stored procedures* também possuem pontos fracos, como por exemplo a criação das regras de negócio fora da estrutura da aplicação e também a dificuldade de migrar a banco de dados caso a empresa necessite. O uso da MV reduz muito o tempo de execução do SQL, porém ela tem um custo para ser criada e atualizada. Esse tempo pode ser recuperado nas futuras execuções do sistema, mas isto deve ser analisado e verificar se atende às necessidades da aplicação. A MV tem outro ponto negativo que é a redundância dos dados no banco, ocupando mais espaço e reduzindo a integridade do banco de dados.

Como já dito, a MV é um recurso muito utilizado no ambiente de *Data Warehouse,* mas possui uma importância muito grande também em outros ambientes de banco de dados. Os bancos de dados transacionais também lidam com alto volume de dados e possuem sistemas executando instruções SQL de alta complexidade, e sua utilidade é melhorar o desempenho de instruções com essa característica. Como pôde ser visto na seção 4 deste trabalho, o mesmo resultado obtido na execução do SQL original, com um tempo de 25,026 segundos, foi possível com apenas 0,01 segundos com a utilização de uma *view* materializada, uma redução de mais de 99% do tempo de execução.

Dentre as técnicas mostradas neste trabalho, a MV foi que a obteve o melhor desempenho na otimização da instrução SQL, mas isso não significa que seja a melhor técnica para todas as situações. A melhor técnica será aquela que mais se adequa às necessidades do momento. O trabalho de otimização em SQL é bem complexo e a adoção de uma única técnica pode não ser eficaz num determinado sistema ou corporação, podendo ser aplicadas mais de uma técnica para obter o melhor resultado.

## Referências

ALMEIDA, Rodrigo. http://imasters.com.br/artigo/1773/oracle/views-materializadas/. 2004.

CARNEIRO, Alessandro. MOREIRA, Juliano. FREITAS, André. http://www.lbd.dcc.ufmg.br/colecoes/erbd/2009/002.pdf. 2009.

MIRANDA, Willian. http://aprendaplsql.com/2014/10/diferenca-entre-views-e-materialized-views-oracle/. 2014.

OLIVEIRA, Anderson. http://www.devmedia.com.br/oracle-materialized-views-gerenciando-em-standby-logicos-no-oracle/32098. 2015.

PRADO, Fábio. http://www.fabioprado.net/2010/10/clausula-with-para-tunar-queries.html. 2010.

PRADO, Fábio. http://www.profissionaloracle.com.br/gpo/artigo/programacao/111-otimizando-aplicacoes-com-o-uso-de-stored-procedures. 2011.

PRADO, Fábio. http://www.devmedia.com.br/tuning-de-sql-em-bancos-de-dados-oracle-revista-sql-magazine-97/23810. 2012.

SANTOS, Jessica. SILVA, Alexandre. http://ftp.unipar.br/~seinpar/2013/artigos/Jessica%20Correa%20dos%20Santos.pdf. 2013.

SANTOS, Erick. http://www.devmedia.com.br/oracle-database-entendendo-o-conceito-de-join-e-outer-join/31398. 2014.

SARIN, Sumit. Oracle DBA - Dicas e Técnicas. Editora Campus. 2000.

VALIATI, Pedro. http://www.devmedia.com.br/artigo-da-sql-magazine-36-indices-no-oracle-parte-i/6835#ixzz3p2RWIvrs. 2007.