INSTITUTO DE ENSINO SUPERIOR DO ESPÍRITO SANTO
FACULDADE DO ESPÍRITO SANTO – MULTIVIX CACHOEIRO DE ITAPEMIRIM
CURSO DE SISTEMAS DE INFORMAÇÃO

FILIPE ERINGER GARRUTH
VINÍCIUS MACHADO DE CARVALHO

ANÁLISE COMPARATIVA DO DESEMPENHO DE SISTEMAS DE BANCO DE DADOS

CACHOEIRO DE ITAPEMIRIM
2014

FILIPE ERINGER GARRUTH
VINÍCIUS MACHADO DE CARVALHO

ANÁLISE COMPARATIVA DO DESEMPENHO DE SISTEMAS DE BANCO DE DADOS

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação na Faculdade do Espírito Santo – Multivix Cachoeiro de Itapemirim, como requisito parcial para obtenção do grau de Bacharel em Sistemas de informação.

Orientador: Prof. Me. Jocimar Fernandes

CACHOEIRO DE ITAPEMIRIM
2014

FILIPE ERINGER GARRUTH
VINÍCIUS MACHADO DE CARVALHO

ANÁLISE COMPARATIVA DO DESEMPENHO DE SISTEMAS DE BANCO DE DADOS

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação na Faculdade do Espírito Santo, como requisito parcial para obtenção do grau de Bacharel em Sistemas de informação.

Aprovado em 01 de dezembro de 2014.

COMISSÃO EXAMINADORA

_____
Prof. Orientador: Me. Jocimar Fernandes

_____
Prof. Convidado: Me. Thiago Caliman Ceschim

_____
Prof. Convidado: Bruno Missi Xavier

A Deus, que se mostrou criador, que foi criativo. Seu fôlego de vida em nós nos foi sustento e nos deu coragem para questionar realidades e propor sempre um novo mundo de possibilidades.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me capacitado, me dado a oportunidade de escrever esse trabalho e por ter me permitido viver até aqui. Agradeço a minha família, que sempre me apoiou. Agradeço aos colegas de trabalho do Instituto Federal do Espírito Santo, que me ajudaram quando eu precisei. Agradeço aos professores Jocimar Fernandes, Alexandre Romanelli, Charles Altoé, Edneia Brambila, Daniel Ventorim, Rafael Guimarães e Cristiano Colombo que me ajudaram muito na construção desse trabalho, bem como me deram dicas excelentes para que eu fizesse um bom trabalho.

Filipe.

Acima de tudo, agradeço à Andréia Machado Ferreira e Samuel Lopes de Carvalho (meus pais). Eles são os responsáveis por minha formação e educação, tudo que sou e tudo o que serei são graças a eles. Agradeço também a Faculdade UNES e seus professores, que me passaram um grande conhecimento durante esses quatro anos que estudei nessa instituição. Agradeço aos meus colegas de classe, e aos meus colegas de trabalho, pelo o compartilhamento de conhecimentos que, juntamente com os professores, foram fundamentais para a minha formação acadêmica. E, por fim, ao meu companheiro de trabalho de conclusão de curso, Filipe Eringer Garruth, pela oportunidade produzir esse trabalho com ele e pela divisão mútua de conhecimentos e experiências.

Vinícius.

“*A sabedoria é a coisa principal; adquire pois a sabedoria, emprega tudo o que possuis na aquisição de entendimento.*
Exalta-a, e ela te exaltará; e, abraçando-a tu, ela te honrará.”

Provérbios 4:7-8.

CARVALHO, Vinícius Machado. GARRUTH, Filipe Eringer. **Análise comparativa do desempenho de sistemas de banco de dados**. 2014. Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação) – Faculdade do Espírito Santo – Multivix Cachoeiro de Itapemirim, Cachoeiro de Itapemirim, 2014.

RESUMO

O crescimento da informática e a necessidade de sua aplicação ao negócio da empresa fazem com que a área de Tecnologia da Informação (TI) deixe de ser vista apenas como um custo e passe a ser pensada como uma ferramenta que agrega valor aos negócios da empresa. Com o uso de sistemas inteligentes é possível gerar informações vitais para os processos gerenciais e tomadas de decisões da organização. São os Sistemas Gerenciadores de Banco de Dados (SGBD's) que gerenciam os dados e informações fornecidos pelos sistemas da empresa. Esses dados e informações são utilizados para gerar os relatórios e cenários que ajudam gestores no processo decisório, na gerência e na administração do empreendimento. Há uma imensa necessidade de agilidade nas consultas e operações que influenciam em uma tomada de decisão e existem diversos SGBD's, cada um com suas características e particularidades, o que torna um pouco complexa a escolha do SGBD que melhor atenderá as necessidades da empresa. Este trabalho analisa e compara o desempenho de três SGBD's: Oracle Database, MySQL e PostgreSQL, bem como a influência de índices na performance dos SGBD's. As análises e comparações são feitas em distintas consultas às bases de dados, cada uma com complexidade diferente. Com base nessas consultas, são realizados testes com diferentes quantidades de registros. Estes testes são efetuados em tabelas sem índices e em tabelas com índices. Portanto, este trabalho provê uma análise comparativa, sob diversos aspectos, desses três SGBD's.

Palavra-chave: SGBD. Indexação. Banco de Dados. Informação. Armazenamento.

CARVALHO, Vinícius Machado. GARRUTH, Filipe Eringer. **Análise comparativa do desempenho de sistemas de banco de dados**. 2014. Trabalho de Conclusão de Curso (Graduação em Sistemas de Informação) – Faculdade do Espírito Santo - Multivix Cachoeiro de Itapemirim, Cachoeiro de Itapemirim, 2014.

ABSTRACT

The growth of computing and the need for its applying in company, make the area of Information Technology not to be seen as only a cost but also as a tool that adds value to the company business. With the use of intelligent systems, it is possible to generate vital information for the management processes and decision making of the organization. The Data Base Management Systems (DBMS) menage the data and information provided by the company systems. This data and information are used to generate the reports and scenery which help managers with decision processes, in management and in the entrepreneur administration.. There is a huge need of agility in queries and operations that influence in decision making and there are several DBMS's, each of them with its characteristics and particularities, what makes the choice of the best DBMS to solve the company needs, a little more complex. This work analyzes and compares the development of three DBMS: Oracle Database, MySQL and PostgreSQL, as well as the influence of indexes of DBMS performance. The analysis and comparisons are made in three different data base queries, each of them with a different complexity. Based on the these queries, tests with different quantity of registers are taken. These tests are made in tables with and without indexes. Therefore, this work provides a comparative analysis, under several aspects, of these three DBMS.

Keywords: DBMS. Indexing. Database. Information. Storage.

LISTA DE ABREVIATURAS E SIGLAS

SGBD – Sistema de Gerenciamento de Banco de Dados.

SQL – Structured Query Language.

DDL – Data Definition Language

1FN – Primeira Forma Normal

2FN – Segunda Forma Normal

3FN – Terceira Forma Normal

XML – Extensible Markup Language

NSF – National Science Foundation

OS – Operational System

IBM – International Business Machines

PL/SQL – Procedural Language/Structured Query Language

< - Menor

<= - Menor ou Igual

= - Igual

>= - Maior ou Igual

> - Maior

LISTA DE QUADROS

LISTA DE FIGURAS

LISTA DE GRÁFICOS

SUMÁRIO

1 INTRODUÇÃO

De acordo com Chandler (2003), a informática está evoluindo exponencialmente nas últimas décadas. Com o surgimento de novas tecnologias, ferramentas voltadas para as organizações são elaboradas com o objetivo de tornar mais simples a vida de um administrador, ofertando suporte a realizar atividades mais rápidas, precisas e com melhores resultados.

Nesse sentido, a era digital consegue seu espaço rapidamente no meio organizacional, fato esse que muda o conceito de como as empresas manteriam armazenadas suas informações. Anteriormente, quando as tecnologias existentes ainda não permitiam o armazenamento de dados digital, os registros eram feitos todos em papel, o que dificultava a organização e a seleção de informações, uma vez que o acesso a elas era dependente de sua localização geográfica. Atualmente, esses registros tem a possibilidade de serem armazenados digitalmente, e em um ambiente organizado sendo possível acessá-los e selecionar informações íntegras de qualquer dispositivo permitido.

Existem diversas formas de armazenar dados em formato digital, algumas delas são arquivos de texto, planilhas, documentos, apresentações e outros. Tais formas de armazenamento são bastante comuns atualmente, mas isso não significa que são suficientes, segundo Date (2004). Uma organização geralmente necessita que os conteúdos desses arquivos sejam tratados a ponto de poder extrair um conjunto de informações que em um determinado momento sejam necessárias para tomar alguma decisão, fato que é possível com arquivos comuns, mas não é viável devido a demanda de tempo.

Organizações necessitam de agilidade nos procedimentos, e essa extração de informações ocorre com facilidade e relativa rapidez quando se usa um banco de dados. Na maioria dos casos, um banco de dados é acompanhado de um software que faz a interação do usuário com os dados. Segundo Rob e Coronel (2011) esse software é chamado de Sistema de Gerenciamento de Banco de Dados (SGBD).

O software de SGBD tem o objetivo principal de fazer a interação entre o usuário ou

um sistema com o banco de dados em si, que se assemelha a um arquivo eletrônico bem estruturado segundo Rob e Coronel (2011). Elmasri e Navathe (2010) dizem que o SGBD é um software que auxilia no processo de definição, construção, manipulação e compartilhamento do banco de dados para usuários ou aplicações.
Nesse sentido, o SGBD se destaca como um componente de grande importância para um banco de dados, sendo que algumas das principais ações no banco de dados são feitas através dele. O SGBD facilita a compreensão dos dados e oferece vantagens como a otimização do desempenho, a segurança dos dados e o aprimoramento de diversos fatores que contribuem para uma experiência adequada com o uso de banco de dados.

Em se tratando de banco de dados relacionais, o SGBD trabalha com uma linguagem padrão denominada *Structured Query Language*, e usualmente é chamada de SQL. O uso da SQL no SGBD permite inserir dados, alterá-los e apagá-los, criar estruturas que possibilitam a organização dos dados dentro de um contexto, e outras funções como a extração de informações a partir de um conjunto de dados. Essas estruturas são comumente criadas no formato tabular, e recebem o nome de “tabelas”. De acordo com Silberschatz, Korth e Sudarshan (1999) os SGBD’s mais robustos são preparados para gerenciar grandes quantidades de informações, o que faz com que as estruturas de armazenamento e os mecanismos de manipulação de dados sejam muito bem projetados.

1.1 Justificativa

Atualmente existem várias empresas que desenvolvem Sistemas de Banco de Dados. Algumas lançam esse produto no mercado e outras os distribuem gratuitamente e deixam o seu código aberto. De acordo com Campos (2006), um sistema de código aberto pode ser entendido não somente por oferecer acesso livre ao seu código fonte, é necessário se enquadrar aos dez critérios da definição de código aberto proposto pela *Open Source Initiative*, que possuem como itens a livre redistribuição, permissão de trabalhos derivados, não discriminação, distribuição da licença, entre outros.

Havendo tantos sistemas de desenvolvedores diferentes, é natural que entre cada

um deles haja alguma diferença de desempenho, até mesmo se for considerado o modo como cada um trabalha. O desempenho em consultas de informações no banco de dados é um fator preponderante para um SGBD. Segundo Silberschatz, Korth e Sudarshan (1999), o custo de tempo de uma consulta ao banco de dados é dado pelo acesso ao disco, uma vez que esse tipo de acesso é lento quando comparado com o acesso a memória principal.

Ainda de acordo com Silberschatz, Korth e Sudarshan (1999), existem várias táticas que são levadas em consideração para agilizar a execução de uma consulta, principalmente quando essa consulta for realizada sobre uma grande quantidade de registros. Com isso é possível entender que a diferença de desempenho entre os SGBD's pode ser mais nítida quando a quantidade de registros a serem consultados for mais expressiva.

Para qualquer empresa atual de grande ou médio porte, as informações são a base de sua existência. Para Rob e Coronel (2011), um bem muito valioso que uma empresa possui é a sua base de dados, porque através dela é possível utilizar recursos que traduzem esses dados em informações que servem como apoio para a tomada de decisão. Com base nessa realidade, grandes organizações necessitam de SGBD's rápidos e robustos para fornecer o conteúdo necessário para uma decisão.

Assim, o retorno dessas informações dependerá diretamente da capacidade de processamento do computador onde o banco de dados estará instalado e também do Sistema de Gerenciamento de Banco de Dados que estará operando. Por esse motivo, a análise contida nesse trabalho pretende demonstrar a importância da escolha do sistema de banco de dados a ser utilizado por uma organização, avaliando as vantagens e desvantagens de cada um que serão apresentadas ao final, visando à escolha da melhor opção que atenderá a necessidade até mesmo do nível estratégico de uma organização. Além disso, este trabalho pode contribuir para o desenvolvimento do conhecimento obtido com o estudo que rendeu o trabalho de conclusão de curso de Lucas Zucoloto Felipe e Welesson Pereira Lopes em 2013 na Faculdade do Espírito Santo - Unes.

1.2 Objetivos

Considerando os fatores e problemas citados na seção anterior, este trabalho tem por objetivo realizar uma análise comparativa do desempenho de três SGBD's. Nessa análise, cada SGBD terá implementado o mesmo modelo de dados e terá uma variação da quantidade de registros em algumas tabelas, sendo executado sobre essas tabelas de tamanhos diferentes algumas consultas de seleção de dados que farão com que o SGBD trabalhe e retorne os resultados em um certo tempo. Esse tempo será considerado para a análise do desempenho do SGBD. Ao final serão exibidos graficamente os resultados obtidos, que servirão para identificar situações em que um ou outro SGBD seja mais apropriado. Os três sistemas que serão utilizados são: PostgreSQL, MySQL e Oracle.

2 INDEXAÇÃO E NORMALIZAÇÃO

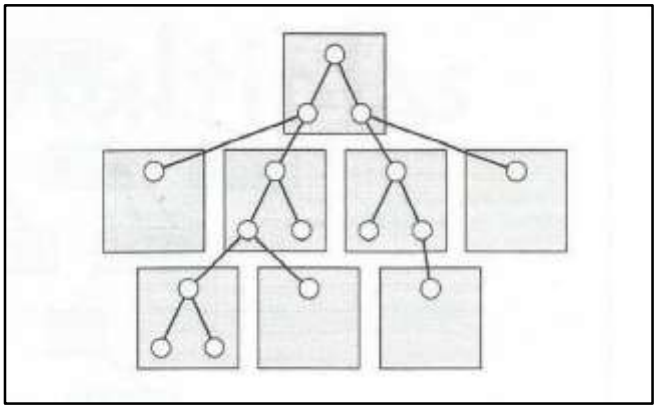
2.1 Indexação

Segundo a documentação do PostgreSQL (2014) os índices são um modo muito comum de melhorar o desempenho do banco de dados. Este permite que o SGBD encontre e exiba registros específicos muito mais rápido do que faria sem o índice. Entretanto, os índices também produzem trabalho adicional para o sistema de banco de dados como um todo devendo, portanto, serem utilizados com sensatez. Stephens (2009) diz que um índice é uma estrutura de banco de dados que faz com que seja mais rápido encontrar registros com base nos valores de um ou mais campos.

Para Rob e Coronel (2011), esse índice vai funcionar da mesma forma de um índice de um livro. Por exemplo, quando alguém deseja encontrar alguma informação em um livro, não faz sentido ler cada página desse livro até encontrar o assunto desejado. Em vez disso, o índice desse livro pode ser verificado para que se possa encontrar o número da página onde determinada informação está, e com esse número em mãos, ir diretamente ao ponto onde o assunto se encontra no livro.

Aprofundando um pouco mais no estudo da indexação em banco de dados, logo surge o conceito de árvores de pesquisa. Assim, é possível perceber que todos os SGBD’s tratados neste trabalho utilizam de alguma forma uma árvore de pesquisa em seu modo de implementação de índices. De acordo com Ziviani (1999) árvores de pesquisa são estruturas de dados bastante eficientes quando o foco é utilizar tabelas que caibam totalmente na memória principal do computador, pois elas atendem condições e requisitos diversificados e conflitantes como o acesso direto e sequencial, facilidade de inserção e remoção de registros e um bom uso de memória. A estrutura visual de uma árvore binária está representada na Figura 1.

Figura 1 - Árvore Binária



Fonte: Drozdek (2005)

De acordo com Drozdek (2005), nos sistemas de banco de dados onde grande parte das informações estão guardadas em discos e fitas, o gasto de tempo para acessar um armazenamento em memória secundária pode ser consideravelmente reduzido pela escolha das estruturas de dados, isso porque já é sabido que o acesso a discos de memória secundária, como discos magnéticos e fitas, são mais lentos que o acesso a memória principal por conter partes mecânicas que levam certo tempo para se locomover até a área do disco onde estão os registros. As árvores B são uma das abordagens utilizadas neste caso.

Segundo Drozdek (2005) uma árvore-B opera junto com a memória de armazenamento secundário e pode ser sintonizada para reduzir os impedimentos impostos por esse tipo de armazenamento. Quando uma árvore de pesquisa possui mais de um registro por nó ela deixa de ser binária, sendo chamadas a partir disso de árvores n-árias, pelo fato de possuírem mais de dois filhos por nó, segundo Ziviani (1999). Comumente nesses casos os nós passam a se chamar páginas.

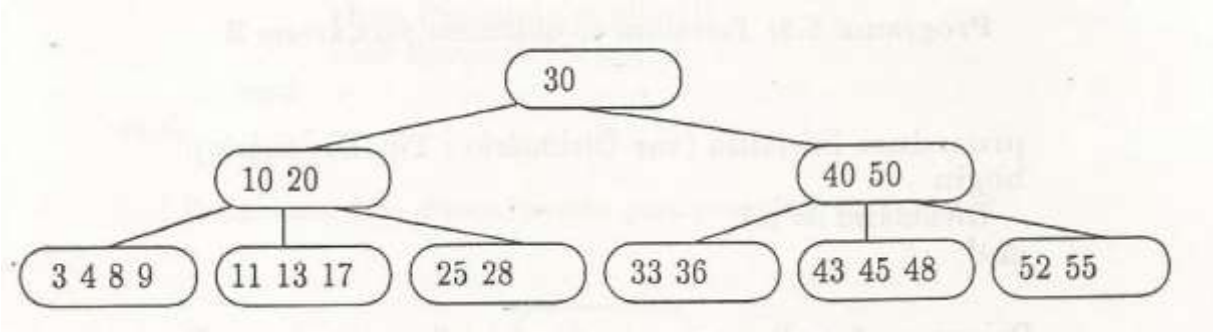
Ziviani (1999) afirma que a árvore B é n-ária e em uma árvore B de ordem m se tem:

1. cada página contém no mínimo m registros (e m + 1 descendentes) e no máximo 2m registros (e 2m + 1 descendentes), exceto a página raiz que pode conter entre 1 e 2m registros;
2. todas as páginas folha aparecem no mesmo nível.

A Figura 2 representa uma árvore B de ordem m = 2 com 3 níveis. Conhecida como

árvore 2-3-4, todas as páginas contêm 2, 3 ou 4 registros, exceto na raiz da árvore que só pode conter um registro. É possível perceber na Figura 2 que os registros são apresentados na ordem crescente. De acordo com Ziviani (1999) esse esquema representa uma extensão natural da organização da árvore binária de pesquisa.

Figura 2 - Árvore B de Ordem 2 com 3 Níveis



Fonte: Ziviani (1999)

De acordo com Drozdek (2005) a figura apresenta um nó de uma árvore B comumente é implementado como uma classe contendo uma matriz de m - 1 células para as chaves, uma matriz de m células de ponteiros para outros nós e, possivelmente, outras informações que facilitem a manutenção da árvore, como o número de chaves em um nó e uma bandeira folha/não-folha.

Segundo a documentação do MySQL (2014) uma estrutura de dados em árvore B é bastante popular para uso em índices de banco de dados. Essa estrutura, na maioria das vezes, é mantida sempre ordenada, permitindo uma pesquisa rápida que retorne resultados exatos (usando um operador de igualdade) ou faixas de resultados (por exemplo, maior que, menor que, entre outros operadores). Ainda segundo a documentação do MySQL os nós da árvore B podem ter muitos filhos, diferentemente da árvore binária, que é limitada a dois filhos por nó.

De acordo com a documentação do PostgreSQL (2014), o mesmo fornece vários tipos de índices: árvore-B, Hash, GiST, SP-GiST e GIN. Cada tipo de índice utiliza um algoritmo diferente que é mais adequado para diferentes tipos de consultas. Por padrão, o comando CREATE INDEX cria índices usando árvore-B, que se encaixam às situações mais comuns. Em particular, o planejador de consultas do PostgreSQL vai considerar o uso de um índice em árvore-B sempre que a coluna indexada está

envolvida em uma comparação utilizando um dos seguintes operadores: <, <=, =, >=, >. Construções de consultas utilizando combinações desses operadores, como “BETWEEN” e “IN” também podem ser implementados com uma pesquisa de índice árvore-B. Além disso, uma condição “IS NULL” ou “IS NOT NULL” em uma coluna indexada pode ser usado com um índice árvore-B.

De acordo com a documentação do PostgreSQL (2014), o seu otimizador também pode utilizar um índice em árvore-B para consultas envolvendo o padrão de operadores de harmonização “LIKE” e “~” se o padrão é uma constante e está ancorado no início da cadeia, como por exemplo “*col LIKE ‘foo%’”* ou “*col LIKE ‘~foo’”*, e não *“col LIKE ‘%bar*”. Também é possível usar índices árvore-B para “ILIKE” e “~*”, mas apenas se o padrão começa com caracteres não-alfabéticos, ou seja, os caracteres que não são afetados pela conversão maiúsculas/minúsculas.

A documentação do Oracle cita que seu sistema de banco de dados fornece vários esquemas de indexação, que fornecem funcionalidades de desempenho complementar. Os índices no Oracle podem ser classificados da seguinte forma:

- Índices árvore-B: Esse é o tipo padrão de índices no Oracle. De acordo com a documentação do Oracle (2014) eles são excelentes para chave primária e índices altamente seletivos. Usado como índices concatenados, os índices árvore-B podem recuperar dados ordenados pelas colunas indexadas. Índices árvore-B possui os seguintes subtipos:
    - Index-organized tables: uma tabela organizada por índices difere de uma grande quantidade de registros organizados porque os dados são os próprios índices.
    - Índices de chave reversa: Neste tipo de índice, os bytes da chave de índice são invertidos, por exemplo, 103 é armazenado como 301. A reversão de bytes espalha inserções no índice ao longo de muitos blocos.
    - Índices decrescentes: Este tipo de índice seleciona dados de índice em uma determinada coluna ou mais em ordem decrescente.
    - Índice de cluster árvore-B: Este tipo de índice é utilizado para indexar uma chave da tabela de cluster. Em vez de apontar para uma linha, a

chave aponta para um bloco que contém linhas relacionadas com a chave do cluster.

- Índices bitmap e bitmap join: Em um índice bitmap, uma entrada de índice utiliza um bitmap para apontar para várias linhas. Diferentemente dos índices árvore B que apontam para uma única linha. Um índice bitmap join é um índice bitmap comum para a junção de duas ou mais tabelas;

- Índices baseados em função: Esse tipo de índice inclui colunas que são transformadas por uma função, como a função UPPER por exemplo, ou incluída em uma função. Índices árvore B ou bitmap podem ser baseados nessa função;

- Índices de domínio de aplicativo: Este tipo de índice é criado pelo usuário para dados em um domínio específico do aplicativo. O índice físico não necessita de utilizar uma estrutura de índice tradicional e pode ser armazenado Oracle como tabelas ou externamente como um arquivo.

Com isso, o SGBD usa os índices para procurar de forma objetiva e precisa um valor dentro do banco, pois os ponteiros especificam a localização em que um determinado valor está armazenado.

2.2 Normalização

A normalização de um banco de dados é basicamente a aplicação de técnicas sobre o relacionamento entre as tabelas que evitam que aconteça redundância de dados, mistura de mais de um contexto em uma mesma tabela e alguns outros problemas. Para Rob e Coronel (2011) a normalização pode ser definida como um procedimento que faz correções estruturais e correções nas tabelas visando diminuir a redundância de dados, fazendo com que as anomalias se minimizem. A redundância de dados ocupa espaço desnecessário em disco e dificulta a manutenção do banco de dados. Se as informações de um cliente, por exemplo, estiverem em várias tabelas, e quando alguma informação desse cliente tiver de ser alterada, será preciso alterar em todos os locais onde esse cliente se encontra. Fazer essa

alteração quando existe apenas uma tabela com informações de cliente seria relativamente menos trabalhoso. Machado (2011) diz que a normalização tem por objetivo minimizar as falhas em um projeto de banco de dados, e extinguir a mistura de contextos em uma tabela, bem como suas redundâncias desnecessárias. Em linhas gerais, a normalização torna o banco de dados relacional mais organizado e consistente, além de fornecer uma forma eficiente de acesso aos registros.

Um dos problemas que a normalização apresenta é o desempenho nas consultas. Normalmente quanto maior o nível de normalização de um modelo de banco de dados, menor é o seu desempenho. Por esse motivo, algumas vezes é mais vantajoso desnormalizar um banco de dados até certo ponto para que o desempenho do mesmo seja desejável.

A normalização opera baseada em algumas fases que são chamadas de formas normais. Existem no total cinco formas normais para banco de dados, que são chamadas de primeira forma normal (1FN), segunda forma normal (2FN), terceira forma normal (3FN), quarta forma normal (4FN) e quinta forma normal (5FN). Entretanto, na prática, as três primeiras formas normais são as mais utilizadas. A 1FN, 2FN e a 3FN foram propostas por E. F. Codd, que também propôs o modelo relacional de dados, segundo Halpin (2001), e as outras formas normais foram introduzidas mais tarde para atender casos adicionais.

A seguir tem-se uma descrição detalhada das três primeiras formas normais.

2.2.1 Primeira forma normal

De acordo com Halpin (2001), uma tabela está na primeira forma normal se, e somente se todas as colunas estiverem armazenando valores atômicos, ou seja, cada coluna deve armazenar apenas um valor e não um conjunto de valores. É possível entender a partir disso que a primeira forma normal não aceita repetições ou mais de um valor por campo.

Para deixar uma tabela na primeira forma normal é preciso primeiramente retirar da tabela os elementos que se repetem. Tomando como exemplo a tabela de projetos

de uma empresa fictícia representada no quadro 1:

Quadro 1 - Tabela de Projetos Desnormalizada

PROJ_NUM	PROJ_NAME	EMP_NUM	EMP_NAME	JOB_CLASS	CHG_HOUR	HOURS
15	Evergreen	103	June E. Arbough	Engº Eletricista	84,5	23,8
		101	John G. News	Projetista de Banco de Dados	105	19,4
		105	Alice K. Johnson	Projetista de Banco de Dados	105	35,7
		106	William Smithfield	Programador	35,75	12,6
		102	David H. Senior	Analista de Sistemas	96,75	23,8
18	Amber Wave	114	Annelise Jones	Projetista de Aplicações	48,1	25,6
		118	James J. Frommer	Suporte Geral	18,36	45,3
		104	Anne K. Ramoras	Analista de Sistemas	96,75	32,4
		112	Darlene M. Smithson	Analista SSD	45,95	45
22	Rolling Tide	105	Alice K. Johnson	Projetista de Banco de Dados	105	65,7
		104	Anne K. Ramoras	Analista de Sistemas	96,75	48,4
		113	Delbert K. Joenbrood	Projetista de Aplicações	48,1	23,6
		111	Geoff B. Wabash	Suporte Escriturário	26,87	22
		106	William Smithfield	Programador	35,75	12,8
25	Starflight	107	Maria M. Alonzo	Programador	35,75	25,6
		115	Travis B. Bawangi	Analista de Sistemas	96,75	45,8
		101	Geoff B. Wabash	Projetista de Banco de Dados	105	56,3
		114	Annelise Jones	Projetista de Aplicações	48,1	33,1
		108	Ralph B. Washinton	Analista de Sistemas	96,75	23,6
		118	James J. Frommer	Suporte Geral	96,75	30,5
		112	Darlene M. Smithson	Analista SSD	45,95	41,4

Fonte: Rob e Coronel (2011)

Analisando o quadro 1 é possível perceber que há grupos de repetição, ou seja, cada ocorrência de um número de projeto, identificado na coluna PROJ_NUM, se refere a um grupo de entradas relacionadas. Para exemplificar, o projeto Starflight (PROJ_NUM = 25) exibe 7 registros que se relacionam, pois eles compartilham a característica PROJ_NUM = 25.

Para converter a tabela do quadro 1 para a primeira forma normal é necessário realizar alguns procedimentos. Primeiramente deve-se eliminar os grupos de repetição. Segundo Rob e Coronel (2011) é importante começar apresentando os dados em formato de tabela, fazendo com que cada célula tenha um valor único e não apresente grupos de repetição. Para fazer isso é necessário eliminar os valores nulos, para que cada repetição contenha um valor adequado. Somente essa alteração já converte a tabela para a primeira forma normal, como é exibido no quadro 2.

Quadro 2 - Tabela de Projetos na 1FN

PROJ_NUM	PROJ_NAME	EMP_NUM	EMP_NAME	JOB_CLASS	CHG_HOUR	HOURS
15	Evergreen	103	June E. Arbough	Engº Eletricista	84,5	23,8
15	Evergreen	101	John G. News	Projetista de Banco de Dados	105	19,4
15	Evergreen	105	Alice K. Johnson	Projetista de Banco de Dados	105	35,7
15	Evergreen	106	William Smithfield	Programador	35,75	12,6
15	Evergreen	102	David H. Senior	Analista de Sistemas	96,75	23,8
18	Amber Wave	114	Annelise Jones	Projetista de Aplicações	48,1	25,6
18	Amber Wave	118	James J. Frommer	Suporte Geral	18,36	45,3
18	Amber Wave	104	Anne K. Ramoras	Analista de Sistemas	96,75	32,4
18	Amber Wave	112	Darlene M. Smithson	Analista SSD	45,95	45
22	Rolling Tide	105	Alice K. Johnson	Projetista de Banco de Dados	105	65,7
22	Rolling Tide	104	Anne K. Ramoras	Analista de Sistemas	96,75	48,4
22	Rolling Tide	113	Delbert K. Joenbrood	Projetista de Aplicações	48,1	23,6
22	Rolling Tide	111	Geoff B. Wabash	Suporte Escrituário	26,87	22
22	Rolling Tide	106	William Smithfield	Programador	35,75	12,8
25	Starflight	107	Maria M. Alonzo	Programador	35,75	25,6
25	Starflight	115	Travis B. Bawangi	Analista de Sistemas	96,75	45,8
25	Starflight	101	Geoff B. Wabash	Projetista de Banco de Dados	105	56,3
25	Starflight	114	Annelise Jones	Projetista de Aplicações	48,1	33,1
25	Starflight	108	Ralph B. Washinton	Analista de Sistemas	96,75	23,6
25	Starflight	118	James J. Frommer	Suporte Geral	96,75	30,5
25	Starflight	112	Darlene M. Smithson	Analista SSD	45,95	41,4

Fonte: Rob e Coronel (2011)

A partir dessa alteração é preciso agora definir a chave primária da tabela, ou seja, quais elementos serão os que identificam de forma única um registro. Neste caso o campo PROJ_NUM não pode ser uma chave primária adequada, pois ele não pode identificar um registro específico. Por exemplo, o valor 15 de PROJ_NUM pode identificar um entre cinco funcionários. Para resolver esse problema é preciso criar uma chave primária composta, isto é, uma chave primária com dois ou mais atributos. Uma chave primária que identificaria exclusivamente um registro deve ser composta, neste caso, por PROJ_NUM e EMP_NUM. Por exemplo, sabendo que PROJ_NUM = 25 e EMP_NUM = 107, os valores dos atributos PROJ_NAME, EMP_NAME, JOB_CLASS, CHG_HOUR E HOURS serão, respectivamente, Starflight, Maria D. Alonzo, Programmer, 35.75 e 24.6.

2.2.2 Segunda Forma Normal

De acordo com Rob e Coronel (2011) a conversão para a segunda forma normal somente é feita quando o modelo que está na primeira forma normal possuir chave primária composta, ou seja, uma chave primária que contém dois ou mais atributos. Se o modelo na primeira forma normal já possuir um único campo em sua chave primária, então automaticamente esse modelo já estará na segunda forma normal. A

tabela mostrada no quadro 2 está apenas na primeira forma normal, uma vez que possui chave primária composta pelos campos PROJ_NUM e EMP_NUM.

Para converter a tabela do quando 2 para a segunda forma normal é preciso realizar algumas mudanças. Para exemplificar de forma prática, duas etapas devem ser seguidas para essa conversão, segundo Rob e Coronel (2011). Primeiramente deve-se apresentar cada componente da chave em uma linha separada, e após isso apresentar a chave original na última linha, por exemplo:

PROJ_NUM
EMP_NUM
PROJ_NUM EMP_NUM

Analisando o exemplo acima é possível perceber que as chaves podem identificar contextos diferentes. Neste caso, PROJ_NUM pode identificar um projeto, EMP_NUM identifica um empregado e, na última linha, PROJ_NUM e EMP_NUM identifica a qual projeto um empregado está designado. Sabendo disso, é possível dividir a tabela original em três novas tabelas, que podem ser PROJETO, EMPREGADO E DESIGNAÇÃO.

A partir do momento em que a tabela original é dividida, é preciso agora distribuir os atributos que são dependentes de cada chave. Por exemplo, PROJ_NAME é o nome de um projeto, então esse campo está relacionado com a chave PROJ_NUM que possui o número do projeto como valor. EMP_NAME, JOB_CLASS e CHG_HOUR são atributos relacionados ao empregado, e estão ligados a chave EMP_NUM, que tem como valor o número de identificação de um funcionário. Sabendo desses detalhes é possível agora distribuir os atributos de acordo com suas chaves, como é mostrado a seguir:

PROJETO (**PROJ_NUM**, PROJ_NAME)
EMPREGADO (**EMP_NUM**, EMP_NAME, JOB_CLASS, CHG_HOUR)
DESIGNAÇÃO (**PROJ_NUM**, **EMP_NUM**, ASSIGN_HOURS)

Os atributos mostrados em negrito e sublinhado são as chaves primárias das

tabelas. A tabela DESIGNAÇÃO contém uma chave primária composta de dois atributos, que são PROJ_NUM e EMP_NUM. Neste caso, um atributo que faz parte de ao menos uma chave é conhecido como atributo primário. Portanto, PROJ_NUM e EMP_NUM são atributos primários. O número de horas gastas por cada funcionário em cada projeto era dependente na tabela original de PROJ_NUM e EMP_NUM, por identificarem um projeto e um funcionário. Sendo assim, na tabela DESIGNAÇÃO foi adicionado o atributo ASSIGN_HOUR para conter essa informação. A estrutura visual das tabelas pode ser conferida nos quadros 3, 4 e 5:

Quadro 3 - Tabela de projetos na 2FN

PROJ_NUM	PROJ_NAME
15	Evergreen
18	Amber Wave
22	Rolling Tide
25	Starflight

Fonte: Desenvolvido pelo autor.

Quadro 4 - Tabela de empregados na 2FN

EMP_NUM	EMP_NAME	JOB_CLASS	CHG_HOUR
103	June E. Arbough	Engº Eletricista	84,5
101	John G. News	Projetista de Banco de Dados	105
105	Alice K. Johnson	Projetista de Banco de Dados	105
106	William Smithfield	Programador	35,75
102	David H. Senior	Analista de Sistemas	96,75
114	Annelise Jones	Projetista de Aplicações	48,1
118	James J. Frommer	Suporte Geral	18,36
104	Anne K. Ramoras	Analista de Sistemas	96,75
112	Darlene M. Smithson	Analista SSD	45,95
113	Delbert K. Joenbrood	Projetista de Aplicações	48,1
111	Geoff B. Wabash	Suporte Escriturário	26,87
107	Maria M. Alonzo	Programador	35,75
115	Travis B. Bawangi	Analista de Sistemas	96,75
108	Ralph B. Washinton	Analista de Sistemas	96,75

Fonte: Desenvolvido pelo autor.

Quadro 5 - Tabela de designações na 2FN

PROJ_NUM	EMP_NUM	ASSIGN_HOURS
15	103	23,8
15	101	19,4
15	105	35,7
15	106	12,6
15	102	23,8
18	114	25,6
18	118	45,3
18	104	32,4
18	112	45
22	105	65,7
22	104	48,4
22	113	23,6
22	111	22
22	106	12,8
25	107	25,6
25	115	45,8
25	101	56,3
25	114	33,1
25	108	23,6
25	118	30,5
25	112	41,4

Fonte: Desenvolvido pelo autor.

A partir desse ponto, Rob e Coronel (2011) dizem que algumas anomalias presentes na tabela do quadro 1 já foram eliminadas. Por exemplo, caso seja preciso alterar, adicionar ou excluir um projeto, seria necessário ir apenas à tabela PROJETO e fazer as alterações em uma linha somente.

2.2.3 Terceira Forma Normal

Para converter o modelo criado até aqui para a terceira forma normal primeiro é preciso entender o conceito de determinação. De acordo com Rob e Coronel (2011) a afirmação “A determina B” significa que, conhecendo o valor de A é possível saber o valor de B. Por exemplo, neste caso, na tabela de empregados, conhecendo o valor de JOB_CLASS é possível saber o valor do atributo CHG_HOUR. Isso caracteriza também uma dependência transitiva, pois o atributo JOB_CLASS não é ou não faz parte da chave primária da tabela de empregados.

Um modelo está na terceira forma normal quando ele está na segunda forma normal e não contém dependências transitivas. Analisando as tabelas montadas até aqui é

possível perceber que somente a tabela de empregados possui dependência transitiva. Sendo assim, para fazer a conversão para a terceira forma normal Rob e Coronel (2011) sugerem primeiramente apresentar o atributo determinante das dependências transitivas como uma chave primária de uma nova tabela. Após isso é preciso identificar os atributos dependentes desse determinante e apresentá-los. Por exemplo, até aqui se tem o seguinte:

JOB_CLASS -> CHG_HOUR

O símbolo -> indica que JOB_CLASS determina CHG_HOUR. Uma nova tabela será criada contendo esses dois atributos, sendo o JOB_CLASS a chave primária. O nome da nova tabela é relacionado ao seu contexto. Nesse caso o nome CARGO parece adequado. A partir desse momento é preciso agora retirar os atributos dependentes das dependências transitivas. Na tabela de empregados será necessário eliminar o atributo CHG_HOUR. É importante observar que o atributo JOB_CLASS permanece na tabela de empregados, agora como uma chave estrangeira. Após a conclusão da conversão do modelo para a terceira forma normal, a tabela CARGO e a tabela EMPREGADOS ficaram como é mostrado nos quadros 6 e 7 respectivamente.

Quadro 6 - Tabela CARGO

JOB_CLASS	CHG_HOUR
Engº Eletricista	84,5
Projetista de Banco de Dados	105
Programador	35,75
Analista de Sistemas	96,75
Projetista de Aplicações	48,1
Suporte Geral	18,36
Analista SSD	45,95
Suporte Escriturário	26,87

Fonte: Desenvolvido pelo autor.

Quadro 7 - Tabela EMPREGADO

EMP_NUM	EMP_NAME	JOB_CLASS
103	June E. Arbough	Engº Eletricista
101	John G. News	Projetista de Banco de Dados
105	Alice K. Johnson	Projetista de Banco de Dados
106	William Smithfield	Programador
102	David H. Senior	Analista de Sistemas
114	Annelise Jones	Projetista de Aplicações
118	James J. Frommer	Suporte Geral
104	Anne K. Ramoras	Analista de Sistemas
112	Darlene M. Smithson	Analista SSD
113	Delbert K. Joenbrood	Projetista de Aplicações
111	Geoff B. Wabash	Suporte Escrituário
107	Maria M. Alonzo	Programador
115	Travis B. Bawangi	Analista de Sistemas
108	Ralph B. Washington	Analista de Sistemas

Fonte: Desenvolvido pelo autor.

Na próxima sessão serão abordadas as principais características dos sistemas de banco de dados que foram utilizados neste trabalho, seguida de uma descrição de cada um deles para título de conhecimento.

3 SISTEMAS DE BANCO DE DADOS UTILIZADOS

Os sistemas de gerenciamento de banco de dados utilizados para os testes foram o PostgreSQL, o MySQL e o Oracle. A seguir é encontrado uma descrição detalhada sobre cada um deles.

3.1 PostgreSQL

Segundo Milani (2008) o SGBD relacional PostgreSQL teve origem em um projeto denominado POSTGRES na Universidade Berkeley, na Califórnia (EUA), em 1986. Nesse projeto uma equipe foi designada para criar um novo modelo de armazenamento de dados e suas respectivas regras. Esse projeto obteve o apoio de alguns órgãos como o *Army Research Office* (ARO) e *National Science Foundation* (NSF).

Como a popularidade do projeto foi crescendo muito na época, o mesmo foi encerrado e a partir dele foi criado o software Postgre95. Essa versão trouxe consigo uma grande mudança que foi a inclusão da linguagem SQL, que substituiria a linguagem PostQUEL que era usada anteriormente. No ano seguinte algumas melhorias foram implementadas, e como o nome Postgre95 já estava desatualizado a ferramenta mudou novamente de nome e passou a se chamar PostgreSQL. Atualmente o software está na versão 9.2.2, e passa por atualizações constantemente.

Segundo Silva (2006) o PostgreSQL é bem abastecido de drivers e camadas de software que o torna compatível com ferramentas de criação de relatórios e também com linguagens de programação importantes, como Java, Visual Basic e Delphi. Um fator que não chega a ser um problema, mas certamente é uma ausência muito sentida é o não suporte ao XML. Em softwares que fazem trocas de dados entre sistemas heterogêneos é esperado que diferentes bancos de dados atuem em cada lado, e, segundo Silva (2006) o mercado vem colocando o XML como a melhor opção para padronizar a comunicação entre sistemas desse tipo.

Em se tratando de sincronização, Silva (2006) cita que o PostgreSQL pode ser

usado em empresas de médio e pequeno porte que tenham a necessidade de manter seus dados sincronizados entre a matriz e as filiais. Além disso, Silva (2006) ainda cita que o PostgreSQL possui uma boa habilidade para lidar com bases de dados muito grandes, como um *datawarehouse*, que pode facilmente ter seus bancos de dados chegando a terabytes.

3.2 MySQL

Segundo Neves e Ruas (2005) o MySQL é um sistema de gerenciamento de banco de dados relacional *open source*, e é um dos sistemas mais utilizados e mais conhecidos a nível mundial. O sistema foi desenvolvido pela empresa sueca MySQL AB Limited Company, que também faz vendas de conjuntos de ferramentas que estão relacionados com a tecnologia MySQL. O MySQL possui duas formas de licenciamento. É possível usar o SGBD como um produto gratuito e também é possível adquirir o software comercialmente, segundo Machado (2006), e nesse modelo de uso o usuário conta com direitos de suporte e alguns outros benefícios.

Um dos destaques do MySQL é a sua velocidade, segundo Machado (2006), o que permite que esse sistema de banco de dados seja usado em dispositivos que contam com recursos mais baixos. Por esse motivo, sites pequenos e médios surgem como grande parte dos usuários que optam por esse sistema. Com a disponibilidade em sistemas operacionais importantes como Windows, Linux, Unix, Solaris, Mac OS X, FreeBSD, HP-UX, IBM AIX e outros, o MySQL conta com essa ampla gama de possibilidades de funcionamento.

3.3 Oracle Database

O sistema de gerenciamento de banco de dados Oracle foi criado no final dos anos 70 e até nos dias de hoje vem se destacando como um banco de dados que oferece um bom suporte a aplicações pesadas. Além do banco de dados, a Oracle desenvolve também ferramentas que servem para construir aplicações computacionais que usa sua base de dados. Esse SGBD também possui uma linguagem própria que é chamada PL/SQL, totalmente compatível com a SQL comum.

A Oracle divide seus produtos de banco de dados em várias edições, sendo que cada uma delas é voltada para uma determinada área de negócio. As versões do banco de dados da Oracle são a Enterprise Edition, Standard Edition, Standard Edition One, Express Edition, Oracle Personal Edition e Oracle Database Lite.

4 ESTUDO DE CASO - TESTES

Como já descrito na primeira sessão desse trabalho, o objetivo principal dos testes está relacionado a verificação e a variação do desempenho da consulta SQL utilizando sistemas de banco de dados diferentes e em tabelas de tamanhos diferentes, identificando situações em que um ou outro SGBD se sobressaia. Para a medição do tempo necessário de cada consulta foi preciso utilizar três softwares, um para cada SGBD.

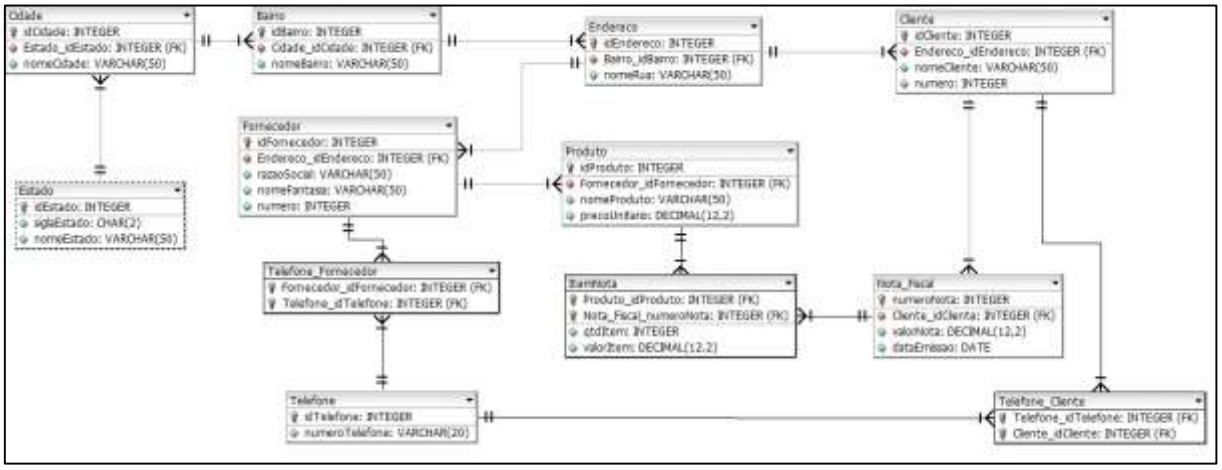
Estes programas interagem com o banco de dados através de um editor de SQL que é fornecido por cada programa, e quando uma consulta de seleção de dados é executada, o tempo que o SGBD leva para selecionar as informações é exibido. Para o Oracle Database foi usado o software Oracle SQL Developer versão 5.0.2.15. Este software é uma ferramenta desenvolvida pela própria Oracle para fornecer uma opção de administrar um banco de dados de forma básica.

Para o MySQL foi usado o Toad for MySQL versão 7.2.0.2922. Este software tem basicamente as mesmas funções do Oracle SQL Developer, bem como o editor de SQL que exibe o tempo de processamento de uma consulta de seleção. Para o PostgreSQL foi usado o PgAdminIII versão 1.18.1. O PgAdminIII acompanha o PostgreSQL em algumas opções de download e também possui a função que é interessante para este trabalho, que é um editor de SQL que exibe o tempo gasto em cada consulta de seleção.

4.1 Estrutura do Banco de Dados

O modelo de dados foi escolhido pensando em simular um ambiente mais próximo possível de um ambiente real. Levando em consideração que operações com compra e venda são muito utilizadas atualmente, e que existem múltiplos sistemas de vendas que utilizam um banco de dados para gerenciar esse processo. Contudo, é preciso esclarecer que o modelo adotado não se trata de um modelo real e não foi aplicado em nenhum banco de dados em funcionamento, portanto, para um uso profissional, ele deve ser repensado e até mesmo atualizado. O modelo do banco de dados utilizado pode ser conferido na figura 3.

Figura 3 - Modelo de Dados Utilizado



Fonte: Desenvolvido pelo autor

Pra a criação dessas tabelas foi preciso escrever toda a estrutura das mesmas em uma linguagem de definição de dados, normalmente chamada por sua sigla DDL (Data Definition Language). A linguagem de definição de dados é responsável pela criação da estrutura do modelo do banco de dados, como as tabelas, as chaves primárias, os índices, as chaves estrangeiras e outros. A DDL é uma das categorias em que a SQL é dividida e é uma opção muito utilizada para criar a estrutura de um banco de dados.

Como cada um dos três SGBD's utilizados possuem particularidades em sua interpretação de DDL, a estrutura dos comandos variou um pouco de um SGBD para outro. Os comandos para a criação das tabelas no MySQL estão descritos no quadro 8:

Quadro 8 – Comandos para criação das tabelas no MySQL

```
1  CREATE TABLE estado(
2    idEstado      integer      not null AUTO_INCREMENT,
3    siglaEstado   char(2)      not null,
4    nomeEstado    varchar(50)  not null,
5    CONSTRAINT pk_estado PRIMARY KEY (idEstado)
6  );
7
8  CREATE TABLE cidade(
9    idCidade      integer      not null AUTO_INCREMENT,
10   nomeCidade    varchar(50)  not null,
11   idEstado      integer      not null,
```

```
12   CONSTRAINT pk_cidade PRIMARY KEY (idCidade),
13   CONSTRAINT fk_cidade_estado FOREIGN KEY (idEstado) REFERENCES estado(idEstado)

14 );
15
16 CREATE TABLE bairro(
17   idBairro    integer      not null AUTO_INCREMENT,
18   nomeBairro  varchar(50)  not null,
19   idCidade    integer      not null,
20   CONSTRAINT pk_bairro PRIMARY KEY (idBairro),
21   CONSTRAINT fk_bairro_cidade FOREIGN KEY (idCidade) REFERENCES cidade(idCidade)
22 );
23
24 CREATE TABLE endereco(
25   idEndereco  integer     not null AUTO_INCREMENT,
26   nomeRua     varchar(50) not null,
27   idBairro    integer     not null,
28   CONSTRAINT pk_endereco PRIMARY KEY (idEndereco),
29   CONSTRAINT fk_endereco_bairro FOREIGN KEY (idBairro) REFERENCES
bairro(idBairro)
30 );
31
32 CREATE TABLE telefone(
33   idTelefone      integer     not null AUTO_INCREMENT,
34   numeroTelefone  varchar(20) not null,
35   CONSTRAINT pk_telefone PRIMARY KEY (idTelefone)
36 );
37
38 CREATE TABLE fornecedor(
39   idFornecedor   integer       not null AUTO_INCREMENT,
40   razaoSocial    varchar(50)   not null,
41   nomeFantasia   varchar(50)   not null,
42   idEndereco     integer       not null,
43   numero         integer       not null,
44   CONSTRAINT pk_fornecedor PRIMARY KEY (idFornecedor),
45   CONSTRAINT fk_fornecedor_endereco FOREIGN KEY (idEndereco) REFERENCES
endereco(idEndereco)
46 );
47
48 CREATE TABLE telefone_fornecedor(
49   idFornecedor  integer  not null,
50   idTelefone    integer  not null,
51   CONSTRAINT pk_telefone_fornecedor PRIMARY KEY (idFornecedor, idTelefone),
52   CONSTRAINT fk_tel_forn_fornecedor FOREIGN KEY (idFornecedor) REFERENCES
53   fornecedor(idFornecedor),
```

```
54   CONSTRAINT fk_tel_forn_telefone FOREIGN KEY (idTelefone) REFERENCES
telefone(idTelefone)
55 );
56
57 CREATE TABLE cliente(
58   idCliente      integer     not null AUTO_INCREMENT,
59   nomeCliente    varchar(50) not null,
60   idEndereco     integer     not null,
61   numero         integer     not null,
62   CONSTRAINT pk_cliente PRIMARY KEY (idCliente),
63   CONSTRAINT fk_cliente_endereco FOREIGN KEY (idEndereco) REFERENCES
endereco(idEndereco)
64 );
65
66 CREATE TABLE telefone_cliente(
67   idCliente  integer  not null,
68   idTelefone integer  not null,
69   CONSTRAINT pk_telefone_cliente PRIMARY KEY (idCliente, idTelefone),
70   CONSTRAINT fk_tel_cli_cliente FOREIGN KEY (idCliente) REFERENCES
cliente(idCliente),
71   CONSTRAINT fk_tel_cli_telefone FOREIGN KEY (idTelefone) REFERENCES
telefone(idTelefone)
72 );
73
74 CREATE TABLE produto(
75   idProduto      integer         not null AUTO_INCREMENT,
76   nomeProduto    varchar(50)     not null,
77   precoUnitario  numeric(12,2)   not null,
78   idFornecedor   integer         not null,
79   CONSTRAINT pk_produto PRIMARY KEY (idProduto),
80   CONSTRAINT fk_produto_fornecedor FOREIGN KEY (idFornecedor) REFERENCES
81   fornecedor(idFornecedor)
82 );
83
84 CREATE TABLE notaFiscal(
85   numeroNota   integer        not null AUTO_INCREMENT,
86   valorNota    numeric(12,2)  not null,
87   dataEmissao  date           not null,
88   idCliente    integer        not null,
89   CONSTRAINT pk_nf PRIMARY KEY (numeroNota),
90   CONSTRAINT fk_nf_cliente FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
91 );
92
93  CREATE TABLE itemNota(
94    numeroNota   integer       not null,
95    idProduto    integer       not null,
```

```
96    qtdItem      integer       not null,
97    valorItem    numeric(12,2) not null,
98    CONSTRAINT pk_itemnota PRIMARY KEY (numeroNota, idProduto),
99    CONSTRAINT fk_itemNf_nf FOREIGN KEY (numeroNota) REFERENCES
notaFiscal(numeroNota),
100   CONSTRAINT fk_itemNf_produto FOREIGN KEY (idProduto) REFERENCES
produto(idProduto)
101  );
```

Fonte: Desenvolvido pelo autor

Os comandos para a criação das tabelas no PostgreSQL estão apresentados no quadro 9:

Quadro 9 – Comandos para criação das tabelas no PostgreSQL

```
1   CREATE TABLE estado(
2     idEstado      serial       not null,
3     siglaEstado   char(2)      not null,
4     nomeEstado    varchar(50)  not null,
5     CONSTRAINT pk_estado PRIMARY KEY (idEstado)
6   );
7
8   CREATE TABLE cidade(
9     idCidade      serial       not null,
10    nomeCidade    varchar(50)  not null,
11    idEstado      integer      not null,
12    CONSTRAINT pk_cidade PRIMARY KEY (idCidade),
13    CONSTRAINT fk_cidade_estado FOREIGN KEY (idEstado) REFERENCES
14     estado(idEstado));
15  CREATE TABLE bairro(
16    idBairro    serial       not null,
17    nomeBairro  varchar(50)  not null,
18    idCidade    integer      not null,
19    CONSTRAINT pk_bairro PRIMARY KEY (idBairro),
20    CONSTRAINT fk_bairro_cidade FOREIGN KEY (idCidade) REFERENCES
21     cidade(idCidade)
22  );
23
24  CREATE TABLE endereco(
25    idEndereco  serial      not null,
26    nomeRua     varchar(50) not null,
27    idBairro    integer     not null,
28    CONSTRAINT pk_endereco PRIMARY KEY (idEndereco),
29    CONSTRAINT fk_endereco_bairro FOREIGN KEY (idBairro) REFERENCES
30      bairro(idBairro)
```

```
31  );
32
33  CREATE TABLE telefone(
34    idTelefone      serial      not null,
35    numeroTelefone  varchar(20) not null,
36    CONSTRAINT pk_telefone PRIMARY KEY (idTelefone)
37  );
38
39  CREATE TABLE fornecedor(
40    idFornecedor   serial        not null,
41    razaoSocial    varchar(50)   not null,
42    nomeFantasia   varchar(50)   not null,
43    idEndereco     integer       not null,
44    numero         integer       not null,
45    CONSTRAINT pk_fornecedor PRIMARY KEY (idFornecedor),
46    CONSTRAINT fk_fornecedor_endereco FOREIGN KEY (idEndereco) REFERENCES
47  endereco(idEndereco)
48  );
49
50  CREATE TABLE telefone_fornecedor(
51    idFornecedor  integer  not null,
52    idTelefone    integer  not null,
53    CONSTRAINT pk_telefone_fornecedor PRIMARY KEY (idFornecedor, idTelefone),
54    CONSTRAINT fk_tel_forn_fornecedor FOREIGN KEY (idFornecedor) REFERENCES
60      fornecedor(idFornecedor),
61    CONSTRAINT fk_tel_forn_telefone FOREIGN KEY (idTelefone) REFERENCES
62      telefone(idTelefone)
63  );
64
65  CREATE TABLE cliente(
66    idCliente      serial      not null,
67    nomeCliente    varchar(50) not null,
68    idEndereco     integer     not null,
69    numero         integer     not null,
70    CONSTRAINT pk_cliente PRIMARY KEY (idCliente),
71    CONSTRAINT fk_cliente_endereco FOREIGN KEY (idEndereco) REFERENCES
72      endereco(idEndereco)
73  );
74
75  CREATE TABLE telefone_cliente(
76    idCliente  integer  not null,
77    idTelefone integer  not null,
78    CONSTRAINT pk_telefone_cliente PRIMARY KEY (idCliente, idTelefone),
79    CONSTRAINT fk_tel_cli_cliente FOREIGN KEY (idCliente) REFERENCES
80      cliente(idCliente),
81    CONSTRAINT fk_tel_cli_telefone FOREIGN KEY (idTelefone) REFERENCES
```

```
82      telefone(idTelefone)
83  );
84  CREATE TABLE produto(
85    idProduto      serial          not null,
86    nomeProduto    varchar(50)     not null,
87    precoUnitario  numeric(12,2)   not null,
88    idFornecedor   integer         not null,
89    CONSTRAINT pk_produto PRIMARY KEY (idProduto),
90    CONSTRAINT fk_produto_fornecedor FOREIGN KEY (idFornecedor) REFERENCES
91      fornecedor(idFornecedor)
92  );
93
94  CREATE TABLE notaFiscal(
95    numeroNota   serial         not null,
96    valorNota    numeric(12,2)  not null,
97    dataEmissao  date           not null,
98    idCliente    integer        not null,
99    CONSTRAINT pk_nf PRIMARY KEY (numeroNota),
100   CONSTRAINT fk_nf_cliente FOREIGN KEY (idCliente) REFERENCES
101     cliente(idCliente)
102 );
103
104 CREATE TABLE itemNota(
105   numeroNota   integer       not null,
106   idProduto    integer       not null,
107   qtdItem      integer       not null,
108   valorItem    numeric(12,2) not null,
109   CONSTRAINT pk_itemnota PRIMARY KEY (numeroNota, idProduto),
110   CONSTRAINT fk_itemNf_nf FOREIGN KEY (numeroNota) REFERENCES
111     notaFiscal(numeroNota),
112   CONSTRAINT fk_itemNf_produto FOREIGN KEY (idProduto) REFERENCES
113     produto(idProduto)
114 );
```

Fonte: Desenvolvido pelo autor.

Nestas duas sequências de comandos DDL para criação de tabelas é possível perceber uma alteração. Onde são escritas as chaves primárias das tabelas fornecedor, cliente, produto e notafiscal, o tipo difere de integer para serial do MySQL para o PostgreSQL respectivamente, e também é possível notar a ausência do AUTO_INCREMENT na frente das definições das chaves primárias nos comandos do PostgreSQL. Essas observações são apenas uma das particularidades da interpretação da DDL de cada SGBD, mas ambos os comandos escritos possuem o mesmo objetivo nessas linhas que é a definição de um

incremento automático do campo em uma unidade.

Os comandos para a criação das tabelas no Oracle estão apresentados no quadro 10:

Quadro 10 – Comandos para criação das tabelas no Oracle Database

```
CREATE TABLE estado(
  idEstado      integer      not null,
  siglaEstado   char(2)      not null,
  nomeEstado    varchar(20)  not null,
  CONSTRAINT pk_estado PRIMARY KEY (idEstado)
);

CREATE TABLE cidade(
  idCidade      integer      not null,
  nomeCidade    varchar(50)  not null,
  idEstado      integer      not null,
  CONSTRAINT pk_cidade PRIMARY KEY (idCidade),
  CONSTRAINT fk_cidade_estado FOREIGN KEY (idEstado) REFERENCES
    estado(idEstado)
);

CREATE TABLE bairro(
  idBairro    integer      not null,
  nomeBairro  varchar(50)  not null,
  idCidade    integer      not null,
  CONSTRAINT pk_bairro PRIMARY KEY (idBairro),
  CONSTRAINT fk_bairro_cidade FOREIGN KEY (idCidade) REFERENCES
    cidade(idCidade)
);

CREATE TABLE endereco(
  idEndereco  integer     not null,
  nomeRua     varchar(50) not null,
  idBairro    integer     not null,
  CONSTRAINT pk_endereco PRIMARY KEY (idEndereco),
  CONSTRAINT fk_endereco_bairro FOREIGN KEY (idBairro) REFERENCES
    bairro(idBairro)
);

CREATE TABLE telefone(
  idTelefone      integer     not null,
  numeroTelefone  varchar(20) not null,
  CONSTRAINT pk_telefone PRIMARY KEY (idTelefone)
```

```
39  );

40  CREATE TABLE fornecedor(
41    idFornecedor   integer       not null,
42    razaoSocial    varchar(50)   not null,
43    nomeFantasia   varchar(50)   not null,
44    idEndereco     integer       not null,
45    numero         integer       not null,
46    CONSTRAINT pk_fornecedor PRIMARY KEY (idFornecedor),
47    CONSTRAINT fk_fornecedor_endereco FOREIGN KEY (idEndereco) REFERENCES
48      endereco(idEndereco)
49  );
50
51  CREATE TABLE telefone_fornecedor(
52    idFornecedor  integer  not null,
53    idTelefone    integer  not null,
54    CONSTRAINT pk_telefone_fornecedor PRIMARY KEY (idFornecedor, idTelefone),
55    CONSTRAINT fk_tel_forn_fornecedor FOREIGN KEY (idFornecedor) REFERENCES
56      fornecedor(idFornecedor),
57    CONSTRAINT fk_tel_forn_telefone FOREIGN KEY (idTelefone) REFERENCES
58      telefone(idTelefone)
59  );
60
61  CREATE TABLE cliente(
62    idCliente      integer     not null,
63    nomeCliente    varchar(50) not null,
64    idEndereco     integer     not null,
65    numero         integer     not null,
66    CONSTRAINT pk_cliente PRIMARY KEY (idCliente),
67    CONSTRAINT fk_cliente_endereco FOREIGN KEY (idEndereco) REFERENCES
68      endereco(idEndereco)
69  );
70
71  CREATE TABLE telefone_cliente(
72    idCliente  integer  not null,
73    idTelefone integer  not null,
74    CONSTRAINT pk_telefone_cliente PRIMARY KEY (idCliente, idTelefone),
75    CONSTRAINT fk_tel_cli_cliente FOREIGN KEY (idCliente) REFERENCES
76      cliente(idCliente),
77    CONSTRAINT fk_tel_cli_telefone FOREIGN KEY (idTelefone) REFERENCES
78      telefone(idTelefone)
79  );
80
81  CREATE TABLE produto(
82    idProduto      integer         not null,
83    nomeProduto    varchar(50)     not null,
```

```
84    precoUnitario  numeric(12,2)   not null,
85    idFornecedor   integer         not null,
86    CONSTRAINT pk_produto PRIMARY KEY (idProduto),
87    CONSTRAINT fk_produto_fornecedor FOREIGN KEY (idFornecedor) REFERENCES
88      fornecedor(idFornecedor)
89  );
90
91  CREATE TABLE notaFiscal(
92    numeroNota   integer        not null,
93    valorNota    numeric(12,2)  not null,
94    dataEmissao  date           not null,
95    idCliente    integer        not null,
96    CONSTRAINT pk_nf PRIMARY KEY (numeroNota),
97    CONSTRAINT fk_nf_cliente FOREIGN KEY (idCliente) REFERENCES
98      cliente(idCliente)
99  );
100
101 CREATE TABLE itemNota(
102   numeroNota   integer       not null,
103   idProduto    integer       not null,
104   qtdItem      integer       not null,
105   valorItem    numeric(12,2) not null,
106   CONSTRAINT pk_itemnota PRIMARY KEY (numeroNota, idProduto),
107   CONSTRAINT fk_itemNf_nf FOREIGN KEY (numeroNota) REFERENCES
108     notaFiscal(numeroNota),
109   CONSTRAINT fk_itemNf_produto FOREIGN KEY (idProduto) REFERENCES
110     produto(idProduto)
111 );
```

Fonte: Desenvolvido pelo autor

Assim é possível identificar que nos campos de definição de chave primária o tipo de dado é novamente INTEGER. Essas alterações não diferem uma estrutura da outra, todos os comandos definem exatamente a mesma estrutura, e as diferenças no texto são apenas particularidades de interpretação de cada SGBD.

Os testes serão realizados em tabelas com diferentes quantidades de dados. Como no modelo escolhido a tabela que registra as vendas é a tabela notafiscal, essa tabela conterá três quantidades diferentes de dados e será alvo dos testes. Os três tamanhos que essa tabela assumirá durante os testes será de dez mil registros, cem mil registros e um milhão de registros.

Para preencher esse volume de dados em tempo hábil foi preciso construir um pequeno programa que cria um arquivo com todo o script de inserção dos dados necessário para os testes. Este programa foi desenvolvido exclusivamente para este trabalho, na linguagem C#. O código escrito para esse processo pode ser visualizado no quadro 11.

Quadro 11 – Software de geração de dados aleatórios

```
1  static void Main(string[] args){
2            StreamWriter wr = new
StreamWriter(@"C:\Users\Filipe\Desktop\arquivo.sql", true);
3            Random random = new Random();
4            int qtdItens;
5            int j, x;
6            int gerado;
7            for (int i = 0; i < 10000; i++){
8            Console.WriteLine("Gerando script de inserção " + (i + 1));
9            wr.WriteLine("INSERT INTO notafiscal VALUES (" + (i + 1) + ", " +
random.Next(500, 500000) + ", '20090108'," + random.Next(1, 6) + ");");
10           wr.WriteLine();
11           qtdItens = random.Next(1, 11);
12           j = 0;
13           wr.WriteLine("-- PRODUTOS DA NOTA " + (i + 1));
14           x = 0;
15           List<int> gerados = new List<int>();
16           while (j < qtdItens){
17                 gerado = random.Next(1, 11);
18                 x = 0;
19                 while (x < gerados.Count){
20                       if (gerado == gerados[x])
21                       {
22                       gerado = random.Next(1, 11);
23                       x = 0;
24                       }
25                       else
26                       {
27                       x++;
28                       }
29                 }
30                 gerados.Add(gerado);
31                 wr.WriteLine("INSERT INTO itemnota VALUES (" + (i + 1) + ", " +
gerado + ", " + random.Next(1, 50) + ", " + random.Next(20, 1000) + ");");
32                 wr.WriteLine();
33                 j++;
```

```
34                  }
35            }
36            wr.Close();
37     }
38 }
```

Fonte: Desenvolvido pelo autor.

O pequeno programa do quadro 11 faz basicamente um *loop* com a quantidade de repetições necessárias para inserir dez mil registros, cem mil registros e um milhão de registros. Alterando apenas o tamanho do *loop for* na linha 19, que no exemplo está como 10000 (dez mil), é possível criar o *script* para a quantidade desejada de registros.

Além de servir para a geração de um script de inserção em massa de dados em um banco de dados, esse programa também utiliza a classe Random, do framework .NET, para gerar aleatoriamente produtos associados a cada registro na tabela notafiscal, que são armazenados na tabela itemnota. Com isso, foi possível simular um ambiente um pouco mais perto da realidade, com registros fictícios de notas fiscais e de produtos associados a essas notas.

É possível entender com isso que, variando a quantidade de registros da tabela notafiscal, a tabela itemnota também irá variar, e pode ficar com um número muito maior de registros do que a tabela notafiscal. Isso acontece porque para cada nota gerada aleatoriamente, são gerados também produtos aleatórios para a referida nota, que pode ser apenas um produto ou vários produtos. Através do relacionamento entre essas duas tabelas é possível saber a que nota fiscal pertence cada item de venda. Por esse motivo, a quantidade de registros na tabela itemnota tende a ficar maior que a quantidade de registros em notafiscal.

A partir do momento em que todos os arquivos são gerados, o próximo passo é executar o script contido nesses arquivos. Fazendo isso, o banco de dados estará então populado com a quantidade de dados necessária para que sejam feitos os testes deste trabalho.

4.2 Comando de Seleção de Dados

Para fazer a seleção dos dados é preciso executar um comando que foi desenvolvido justamente para esse fim, trata-se do comando SELECT. O comando SELECT tem a finalidade de examinar e selecionar as informações armazenadas em uma determinada tabela ou até mesmo mostrar informações que são geradas a partir de dados presentes em várias tabelas. Segundo Machado (2011), o comando SELECT é uma das operações mais utilizadas em um SGBD por ser o principal meio de coletar informações de bancos de dados relacionais e trazê-las para o usuário ou o sistema utilizado.

A estrutura básica de um comando SELECT consiste em três partes:

- SELECT: define quais serão as colunas da tabela em que o SGBD irá resgatar os dados nela contidos. Se especificado o caractere “*”, todos os campos da tabela são selecionados.
- FROM: indica qual, ou quais tabelas possuem os campos anteriormente definidos no comando SELECT.
- WHERE: expõe ao SGBD a condição ou critério para a seleção das informações presentes na tabela. Na cláusula WHERE, pode-se definir a seleção de uma única informação, ou de um conjunto de informações específicas de acordo com o valor de um determinado campo.

Além da estrutura básica, é possível também integrar ao comando SELECT alguns recursos como a funções de agregação, seleção de dados agrupados e comparação de resultados das funções de agregação.

As funções de agregação são basicamente recursos que permitem a realização de cálculos matemáticos sobre um conjunto de valores e estes retornam um único valor. Algumas das funções mais utilizadas são a AVG, que calcula a média de um grupo de valores, a SUM, que realiza a soma de um grupo de valores, a COUNT, que retorna a quantidade de valores, a MIN, que retorna o valor mínimo presente em um grupo de valores e a MAX, que retorna o valor máximo presente em um grupo de valores.

É possível perceber que todas as funções de agregação já citadas operam sobre um grupo de valores. Esse grupo de valores deve ser especificado no comando SELECT través do recurso GROUP BY. Esse recurso vai definir por qual campo a seleção será agrupada, para que os resultados de média, soma, valor máximo, valor mínimo e quantidade de registros sejam desejáveis. Um exemplo do funcionamento do GROUP BY é uma seleção da média dos salários dos funcionários de cada departamento de uma organização. Nesse caso, o GROUP BY estaria especificando o campo da tabela que identificaria cada departamento e a função de agregação que seria utilizada para esse fim seria a AVG.

Ainda é possível realizar no comando SELECT a comparação de valores retornados pelas funções de agregação. Essa comparação é feita pelo recurso HAVING. O HAVING funciona de forma parecida com a cláusula WHERE, porém, o HAVING usa como parâmetro de comparação os resultados das funções de agregação. Um exemplo do uso do HAVING seria a seleção dos departamentos de uma organização em que a média dos salários dos funcionários seja maior que 3000.

No quadro 12 é possível conferir um exemplo de um SELECT utilizando os exemplos citados anteriormente, com as funções de agregação, a cláusula GROUP BY e HAVING, selecionando de um modelo fictício os departamentos que tenham a média do salário de seus funcionários igual ou superior a 3000:

Quadro 12 – Exemplo de comando select

```
1  SELECT nome_departamento
2  FROM departamento d,
3  empregado e
4  WHERE d.cod_empregado = e.cod_empregado
5  GROUP BY d.cod_departamento
6  HAVING AVG(e.salario) >= 3000;
```

Fonte: Desenvolvido pelo autor.

Levando em consideração os fatores sobre o comando SELECT citados até aqui, elaborou-se para este trabalho alguns comandos distintos que foram executados nos

sistemas de banco de dados objetos deste estudo. O primeiro comando é bem simples e é executado sobre apenas uma tabela, que é justamente a tabela que varia a quantidade de registros, a tabela notafiscal:

```
SELECT * FROM notafiscal WHERE valornota > 3000;
```

O comando descrito acima fará o sistema de banco de dados procurar todos os dados das notas fiscais (especificado pelo caractere *) existentes na tabela notafiscal que possuam como valor do campo valornota um número maior que três mil, em outras palavras, o SGBD irá retornar todas as notas fiscais tenham superado o valor de três mil reais em itens.

Primeiramente essa consulta foi executada sem nenhum índice criado para nenhum campo, apenas os índices padrões que o próprio SGBD cria para a chave primária de cada tabela. Após esse primeiro momento, foi criado um índice para a coluna valornota, da tabela notafiscal, tomando base para essa criação um ponto apresentado na seção 4.2 deste trabalho, que menciona que colunas utilizadas em cláusula WHERE são mais eficientes quando são indexadas.

O objetivo de preparar consultas distintas é de mostrar que sistemas de banco de dados podem ter desempenhos diferentes quando o processamento exigido para a consulta varia. Neste primeiro caso apenas uma tabela foi usada, não sendo aplicado nenhuma junção com outra tabela.

O segundo comando elaborado também é simples e muito parecido com o primeiro. A única mudança é que, em vez de o sistema de banco de dados retornar os dados das notas fiscais com valor superior a três mil, irá retornar agora o nome dos clientes para qual foram emitidas notas fiscais com valor superior a três mil reais. O comando é apresentado no quadro 13.

Quadro 13 – Segundo comando select

```
1  SELECT DISTINCT c.nomeCliente
2  FROM cliente c
3          INNER JOIN notafiscal nf ON nf.idCliente = c.idCliente
4  WHERE nf.valornota > 3000;
```

Fonte: Desenvolvido pelo autor.

É possível perceber agora que foi utilizada uma junção entre a tabela cliente e a tabela nota fiscal, especificada no comando pela cláusula INNER JOIN. É sabido que um comando que tenha de fazer um relacionamento entre duas ou mais tabelas pode ser executado mais lentamente do que um comando executado sobre apenas uma tabela. Por esse motivo essa pequena variação do comando foi incluída neste estudo de caso.

Também é possível perceber o uso da cláusula DISTINCT nesse último comando. A cláusula DISTINCT evita que o nome do cliente seja repetido todas as vezes que o sistema de banco de dados encontrar uma nota fiscal com valor superior a três mil que esteja associada a esse cliente. Como na relação entre as tabelas um mesmo cliente pode possuir várias notas fiscais com valor acima de três mil, sem o uso da cláusula DISTINCT esse comando repetiria o nome do cliente cada vez que encontrasse uma nota fiscal emitida para esse cliente com valor superior a três mil, fazendo com que o resultado do comando ficasse muito maior desnecessariamente.

Para um terceiro momento elaborou-se um comando SELECT que usasse a cláusula ORDER BY. A cláusula ORDER BY define em que ordem os dados serão mostrados, podendo ser ordenado por qualquer coluna de uma tabela. Levando em consideração o modelo de dados apresentado na figura 3, foi proposto selecionar os nomes dos clientes, os valores das notas fiscais e a data de emissão das notas fiscais que foram emitidas no período de 16-03-2014 a 22-03-2014, sendo que os resultados serão apresentados ordenados pela data. O comando para realizar essa consulta é apresentado no quadro 14:

Quadro 14 – Terceiro comando select

```
1  SELECT c.nomecliente,
2         nf.valornota,
3         nf.dataemissao
4    FROM notafiscal nf INNER JOIN cliente c ON nf.idcliente = c.idcliente
5    WHERE nf.dataemissao BETWEEN '16-03-2014' AND '22-03-2014'
6    ORDER BY nf.dataemissao;
```

Fonte: Desenvolvido pelo autor.

O relacionamento entre as tabelas é aplicado na prática nos últimos dois comandos, onde a cláusula INNER JOIN compara os valores de colunas que vieram de tabelas associadas, ou seja, os registros das duas tabelas são usados para que os dados sejam gerados a partir do relacionamento entre as duas.

Assim como nos dois primeiros casos, primeiramente este segundo comando foi executado sem nenhum índice criado para nenhum campo. Logo após esse teste, foi criado um índice na coluna dataemissao, da tabela notafiscal. Os fatores considerados para a criação desses índices também foram os pontos levantados na seção 4.3 deste trabalho, principalmente no que tange a frequência da pesquisa e as comparações feitas na cláusula WHERE.

4.3 Comando de Criação de Índice

A estrutura interna do índice foi abordada em detalhes na seção 2, e resumidamente funciona como o índice de um livro, que aponta o número da página onde a informação desejada está. Se não houvesse um índice no livro, seria gasto muito mais tempo procurando a informação desejada página por página. Sendo assim, o índice permite um acesso mais rápido aos registros de uma tabela baseando-se no conteúdo de uma ou várias colunas. As tabelas de índices ficam totalmente ocultas ao usuário, pois são utilizadas nos “bastidores” do SGBD, tornando-as transparentes para a sua utilização.

Para a criação de índices é preciso considerar alguns pontos, segundo Machado (2011). São eles:

- O índice deve ser de coisas a serem pesquisadas com frequência.
- Devem-se indexar as chaves estrangeiras quando precisar de *joins* mais eficientes e performáticos.
- As colunas que são regularmente utilizadas em *joins* devem ser indexadas, porque o sistema pode executar esses *joins* de modo muito mais rápido, e como tempo de resposta é algo vital no desenvolvimento de projetos físicos de banco de dados, é recomendável a sua utilização.
- O índice deve ser criado sempre em colunas que são pesquisadas por um intervalo de valores.
- O índice deve ser criado sempre em colunas que são utilizadas em cláusulas WHERE.

Embora a utilização de índices seja para agilizar o resultado de uma busca no banco de dados, existem situações em que não são aconselháveis a sua criação. Por exemplo, quando se sabe que o resultado de uma busca depende de um campo cujo o mesmo só pode ter dois valores, é evidente que essa consulta retornará uma grande quantidade de registros. Nesse caso, índices não devem ser usados. Segundo Machado (2011) “a criação dos índices depende muito do projeto de banco de dados [...] estão muito ligados às necessidades de velocidade na recuperação da informação, e na execução rápida de uma operação de JOIN.”

Para criar um índice é necessário executar um comando SQL, aplicando-o a uma tabela e especificando uma ou mais colunas dessa tabela. O comando para a criação é o CREATE INDEX, e sua sintaxe básica é:

CREATE [UNIQUE] INDEX <nome do índice>
ON <nome da tabela> (<coluna (s)>);

Também é possível excluir um índice. No PostgreSQL e no Oracle Database o comando de remoção de um índice é o seguinte:

DROP INDEX <nome do índice>;

No MySQL o comando é um pouco diferente, sendo necessário especificar o nome

da tabela na qual o referido índice está criado, ficando assim:

DROP INDEX <nome do índice> ON <nome da tabela>;

É interessante executar uma análise das tabelas após a criação ou remoção de um índice. O comando para a realização desta análise é o ANALYSE no PostgreSQL. Segundo a documentação do PostgreSQL (DOCUMENTAÇÃO, 2014) “o comando ANALYSE coleta estatísticas sobre o conteúdo do banco de dados [...], posteriormente, o planejador de comandos utiliza estas estatísticas para ajudar a determinar o plano de execução mais eficiente para os comandos”. Para o MySQL o comando varia um pouco e é o “ANALYZE TABLE <nome da tabela>;”, sendo necessário especificar o nome da tabela que será analisada. No Oracle Database o comando é “ANALYZE TABLE <nome da tabela> COMPUTE STATISTICS;”.

Os comandos para a criação dos índices usados neste trabalho foram:

CREATE INDEX ind_valornota ON notafiscal(valornota);
CREATE INDEX ind_notafiscal ON notafiscal(dataemissao);

4.4 Dinâmica do Trabalho

A dinâmica dos testes seguirá os seguintes passos:

1. Execução da consulta no banco de dados cujo a tabela notafiscal possua dez mil registros, por dez vezes;
2. Obter o tempo que cada consulta levou para ser executada e calcular a média aritmética desses tempos;
3. Execução da consulta no banco de dados cujo a tabela notafiscal possua cem mil registros, por dez vezes;
4. Obter o tempo que cada consulta levou para ser executada e calcular a média aritmética desses tempos;
5. Execução da consulta no banco de dados cujo a tabela notafiscal possua um milhão de registros, por dez vezes;
6. Obter o tempo que cada consulta levou para ser executada e calcular a média

aritmética desses tempos;

7. Criar um índice no campo adequado para otimizar a consulta;
8. Repetir os passos 1 a 6.

Esses oito passos apresentados serão seguidos para os três sistemas de banco de dados estudados neste trabalho. Primeiramente as consultas foram feitas no PostgreSQL, depois no MySQL e por último no Oracle Database.

Todos os testes foram realizados em um mesmo computador notebook com processador Intel(R) Core(™) i5-3337U 1.80 GHz, memória RAM de 4 GB, sistema operacional Windows 7 Ultimate, versão do PostgreSQL foi a 9.2.2, a do MySQL foi 5.6.19 e a do Oracle foi a Enterprise 11g Release 2.

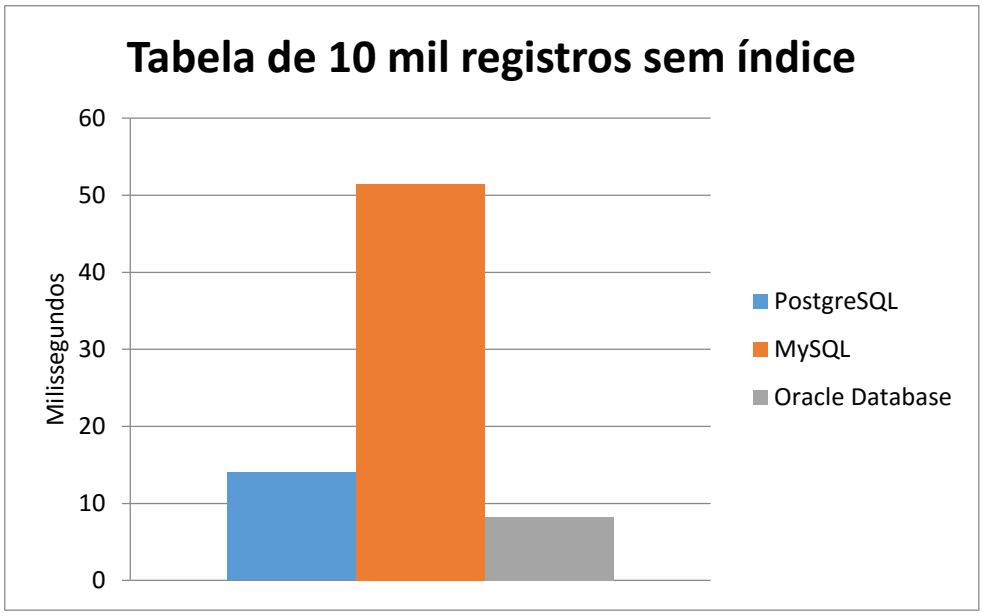
4.5 Resultados Obtidos: Tabela de Dez Mil Registros

Com dez mil registros na tabela, as consultas foram executadas com relativa rapidez. É compreensível que o resultado da consulta seja exibido quase que imediatamente, uma vez que uma tabela de dez mil registros pode ser considerada pequena. Por mais que o resultado tenha sido apresentado de forma rápida, ainda assim houve diferenças significativas de desempenho entre cada sistema de banco de dados testado ao executar cada consulta de diferentes níveis de complexidade mostradas na seção 4.2 deste trabalho.

4.5.1 Primeira consulta

Ao executar a consulta que utiliza apenas uma tabela sem nenhum índice criado o PostgreSQL obteve uma média de 14 milissegundos de tempo gasto para realizar essa operação. O MySQL levou uma média de 51,5 milissegundos e o Oracle Database 8,2 milissegundos. Em média o Oracle Database executou a operação 41% mais rápido que o PostgreSQL e 84% mais rápido que o MySQL. O PostgreSQL executou a consulta 72% mais rápido que o MySQL. Esses resultados podem ser vistos mais nitidamente no gráfico apresentado no gráfico 1.

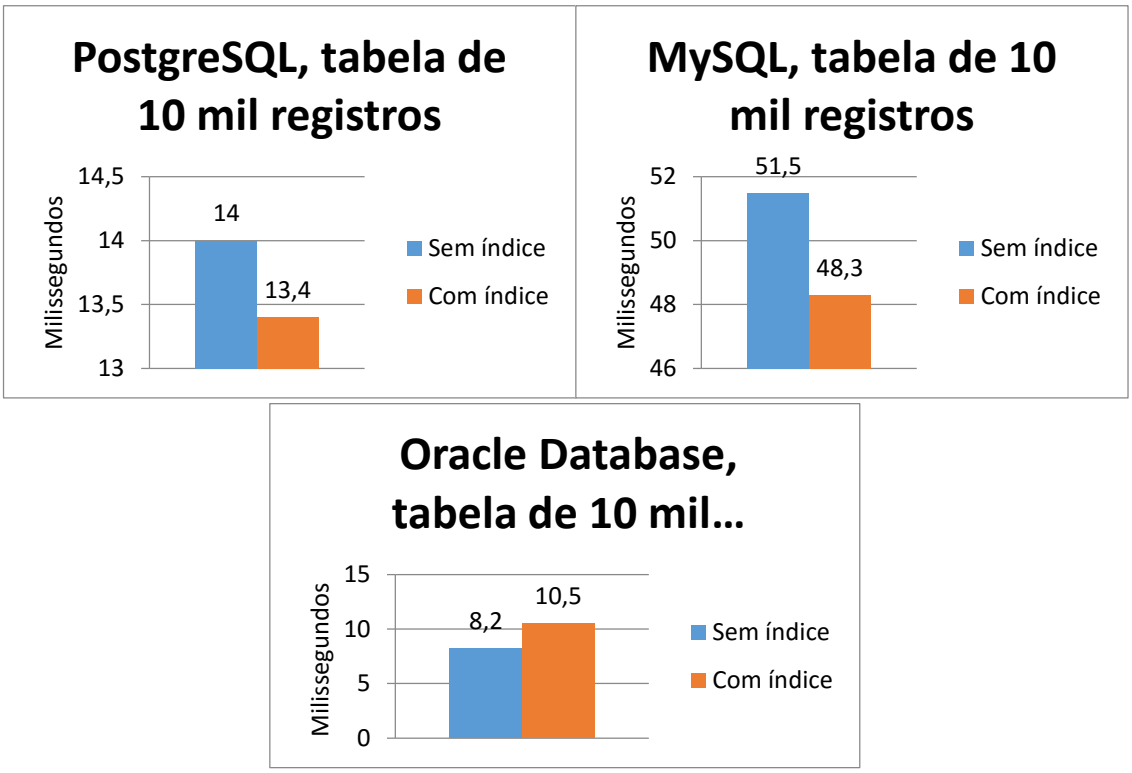
Gráfico 1 – Tabela de 10 mil registros sem índice (Select sobre uma tabela)



Fonte: Desenvolvido pelo autor.

Após a criação do índice como descrito na seção 4.3 deste trabalho, os resultados obtidos foram realmente mais rápidos em relação aos comandos executados sem nenhum índice. A exceção foi o Oracle Database. Os sistemas PostgreSQL e MySQL comprovaram que ao utilizar um índice criado no campo adequado a consulta é executada mais rápida. As diferenças de desempenho entre os sistemas de banco de dados, no entanto, se mantiveram praticamente as mesmas de quando a consulta foi executada sem nenhum índice criado. A variação da performance de cada SGBD após a criação do índice está explícita na gráfico 2.

Gráfico 2 – Variação de desempenho após a criação do índice

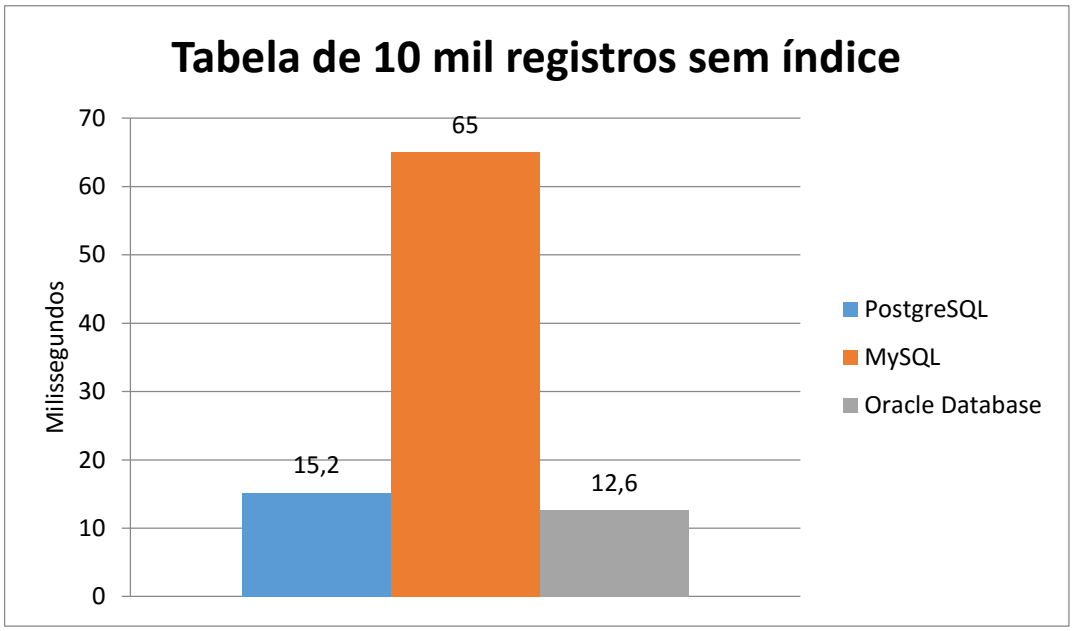


Fonte: Desenvolvido pelo autor.

4.5.2 Segunda consulta

Quando a consulta executada foi a do segundo caso apresentado na seção 4.2 deste trabalho, que representa a consulta que utiliza um relacionamento entre duas tabelas, sem nenhum índice criado, o resultado foi ligeiramente diferente do primeiro caso. O Oracle Database executou a consulta cerca de 19% mais rápido que o PostgreSQL e 81% mais rápido que o MySQL. O PostgreSQL executou a consulta 72% mais veloz que o MySQL. No gráfico 3 pode ser conferido os resultados desse segundo caso.

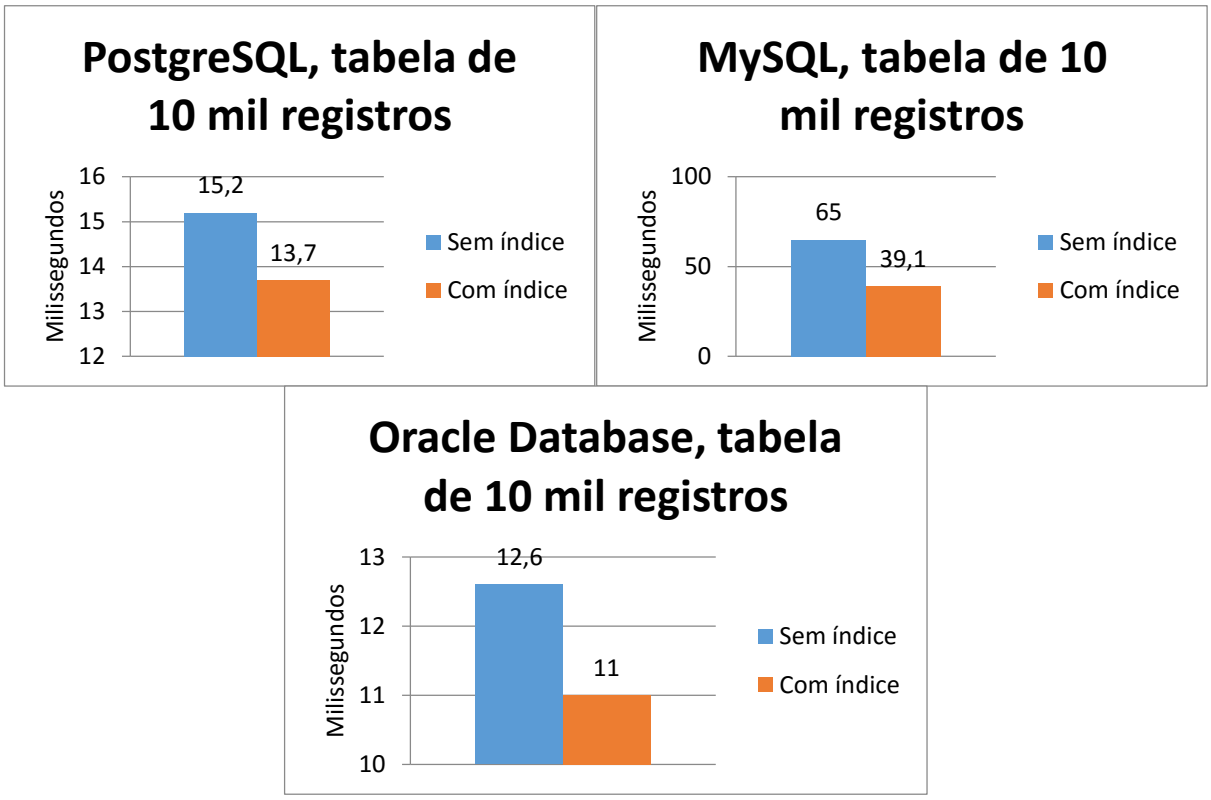
Gráfico 3 – Tabela de 10 mil registros sem índice



Fonte: Desenvolvido pelo autor.

Ao criar o índice como descrito na seção 4.3 deste trabalho, na consulta deste segundo caso, foi comprovado por todos os sistemas de banco de dados estudados a otimização da consulta em no mínimo 9% e chegando a 40% de otimização em alguns casos no MySQL. Assim como no caso anterior, as diferenças de desempenho entre os sistemas de banco de dados não mudaram após a criação do índice, e mantiveram praticamente a mesma porcentagem descrita na análise comparativa das consultas sem índice, conforme pode ser visto no gráfico 4.

Gráfico 4 – Variação de desempenho após a criação do índice

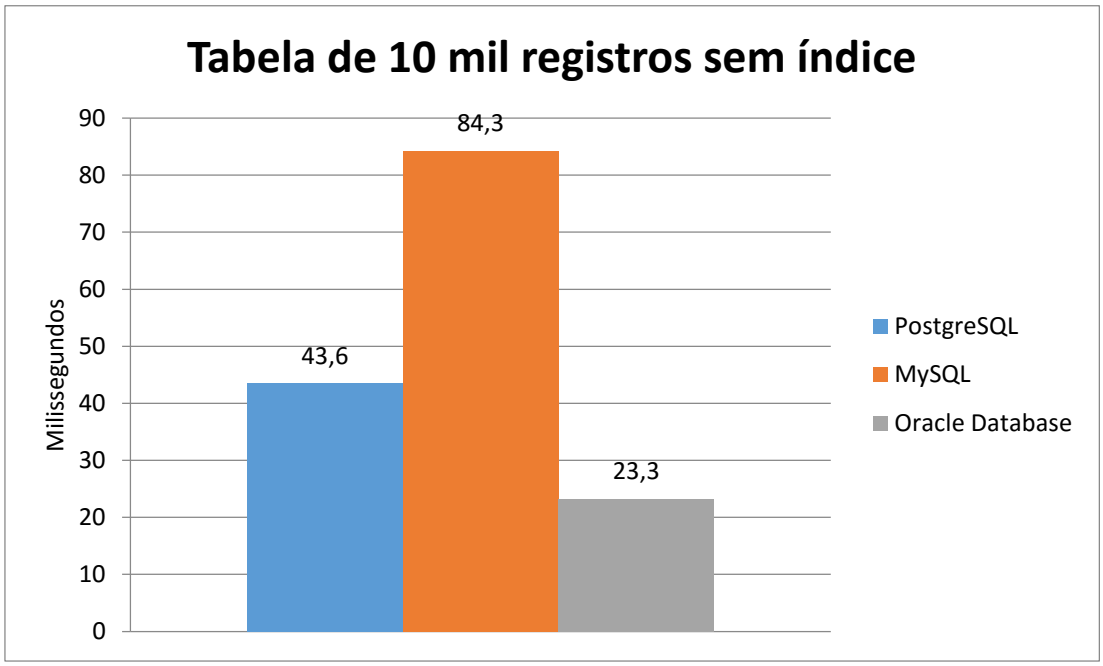


Fonte: Desenvolvido pelo autor.

4.5.3 Terceira consulta

Na terceira consulta realizada, na qual os resultados foram filtrados pela data de emissão de cada nota fiscal, foram obtidas as diferenças de desempenho entre cada SGBD. Quando a consulta foi executada sem nenhum índice criado, o PostgreSQL levou em média 43,6 milissegundos para exibir os resultados, o MySQL levou uma média de 84,3 milissegundos e o Oracle Database 23,3 milissegundos. Em números, o Oracle Database executou a operação cerca de 46% mais rápido que o PostgreSQL e 72% mais rápido que o MySQL e até aqui é o SGBD que possuiu o melhor desempenho nos testes realizados. O gráfico 5 representa visualmente os resultados aqui tratados.

Gráfico 5 – Tabela de 10 mil registros sem índice

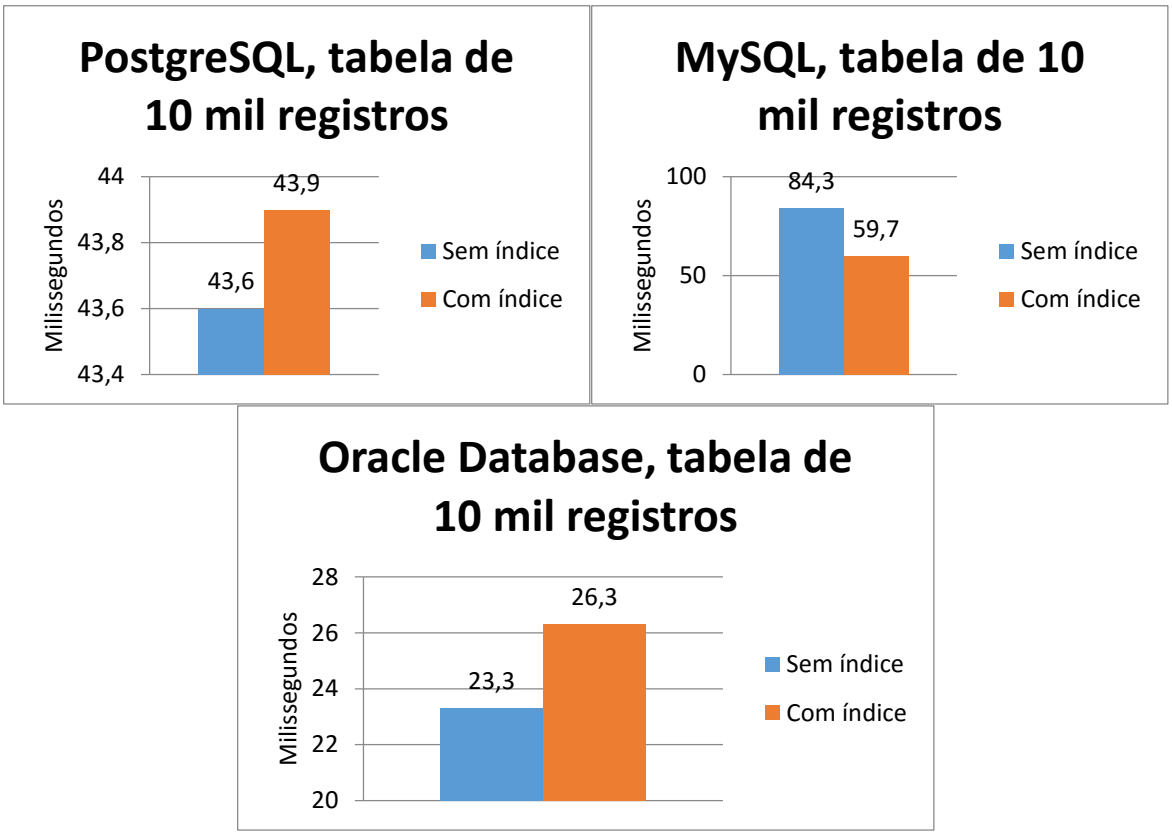


Fonte: Desenvolvido pelo autor.

Após a criação do índice, os resultados foram diferentes e surpreendentes. Assim como o caso do Oracle Database, na primeira consulta, todos os sistemas de banco de dados nessa terceira consulta levaram mais tempo para trazer os resultados com a utilização de um índice do que sem a utilização do mesmo. É possível ter uma visão mais clara desse fato no gráfico 6. Os motivos para o acontecimento dessa anomalia não fazem parte do escopo deste trabalho, sendo possível o estudo das causas e circunstâncias que levam o SGBD a se comportar dessa forma em outro trabalho futuro de conclusão de curso, realizando um estudo complementar do presente trabalho.

A diferença de desempenho entre cada sistema de banco de dados após a criação do índice manteve sua porcentagem que foi exibida na análise dos resultados sem índice. Até o momento, o Oracle Database possui o melhor desempenho nas situações testadas, seguido de perto pelo PostgreSQL que possui o segundo melhor desempenho, e por último, até o momento, o MySQL possui o pior desempenho. Nos próximos tópicos, os testes serão realizados utilizando tabelas de tamanhos diferentes, o que pode interferir diretamente na diferença de desempenho de cada SGBD.

Gráfico 6 – Variação do desempenho após a criação do índice



Fonte: Desenvolvido pelo autor.

4.6 Resultados Obtidos: Tabela de Cem Mil Registros

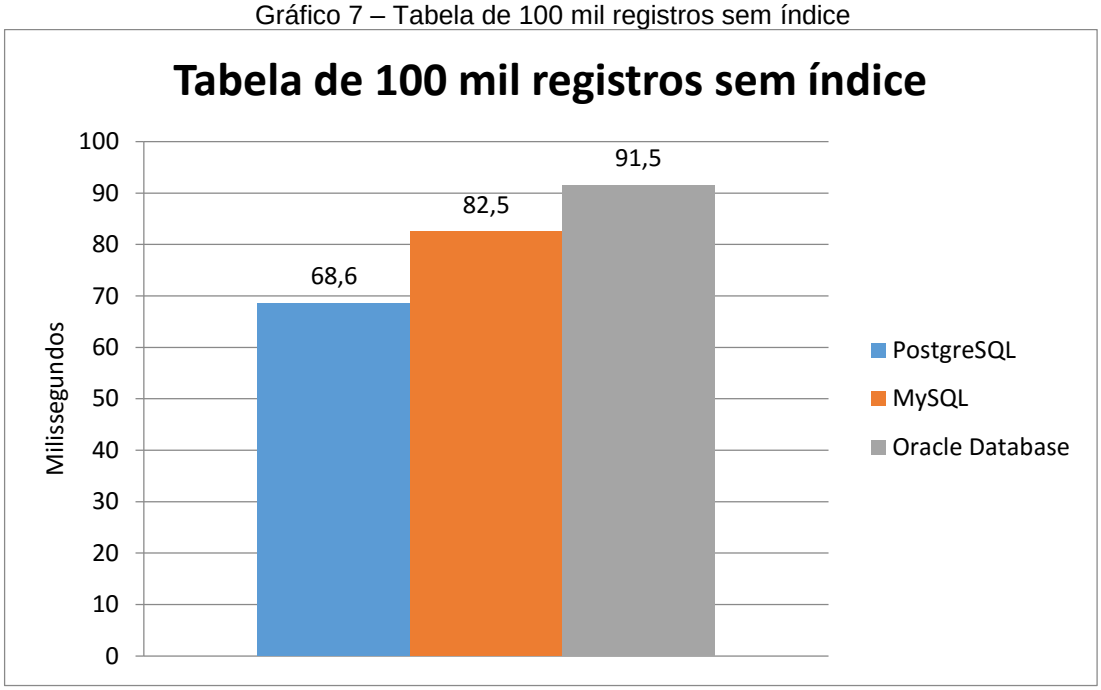
Aumentando o volume de dados em dez vezes, as consultas aos SGBD's ainda são rápidas, já que uma tabela com cem mil registros não é, para o SGBD, uma tabela realmente grande. Porém o nível de complexidade das consultas e os índices fazem com que seja perceptível uma grande diferença no comportamento dos SGBD's analisados.

4.6.1 Primeira consulta

Após a execução da consulta em apenas uma tabela, sem a criação de índices, o Oracle Database, que na tabela com dez mil registro não indexada foi o mais eficiente, teve o pior resultado, uma média de 91,5 milissegundos, 9 milissegundos a mais que o segundo mais eficiente, o MySQL, que com 82,5 milissegundos foi 9% mais rápido que o Oracle Database. O PostgreSQL foi o SGBD que teve o melhor desempenho, obtendo uma média de 68,6 milissegundos de tempo gastos para a

consulta, 16,85% mais veloz que o MySQL e 25,03% mais veloz que o Oracle Database. Um comparativo gráfico sobre o desempenho dos SGBD's analisados é mostrado no gráfico 7.

Gráfico 7 – Tabela de 100 mil registros sem índice



Fonte: Desenvolvido pelo autor.

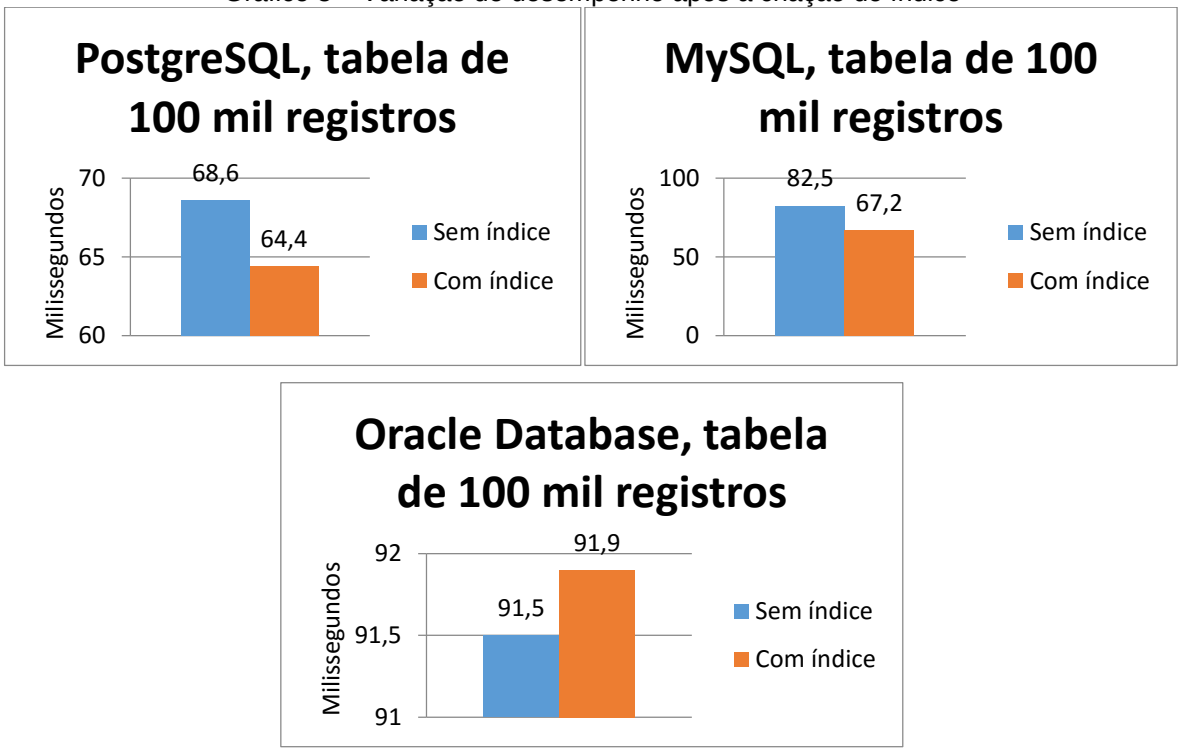
Novas consultas foram realizadas nessa tabela que continha cem mil registros, porém, desta vez, utilizando o index exposto na seção 4.3 desta obra. O PostgreSQL e o MySQL tiveram uma pequena melhora de desempenho, a mudança representou um ganho de 6,13% e 18,55% respectivamente.

O Oracle Database, na média, foi um pouco mais lento nos testes efetuados após a indexação, foi uma diferença muito pequena, passou de 91,5 milissegundos para 91,9, a perda de desempenho foi de 0,43%.

Mesmo com os diferentes comportamentos, e tendo o MySQL com a melhor otimização após a criação do índice, o PostgreSQL continuou sendo o SGBD que obteve o melhor desempenho na consulta a uma tabela com cem mil registros. O MySQL foi o segundo melhor, com uma diferença de 2,8 milissegundos, uma performance de 4,34% inferior à performance do PostgreSQL. O Oracle Database foi 26,88% mais lento do que o MySQL e 29,93% mais lento que o PostgreSQL. O

gráfico 8, mostrado abaixo, exibe a variação de desempenho individual após a indexação.

Gráfico 8 – Variação de desempenho após a criação do índice

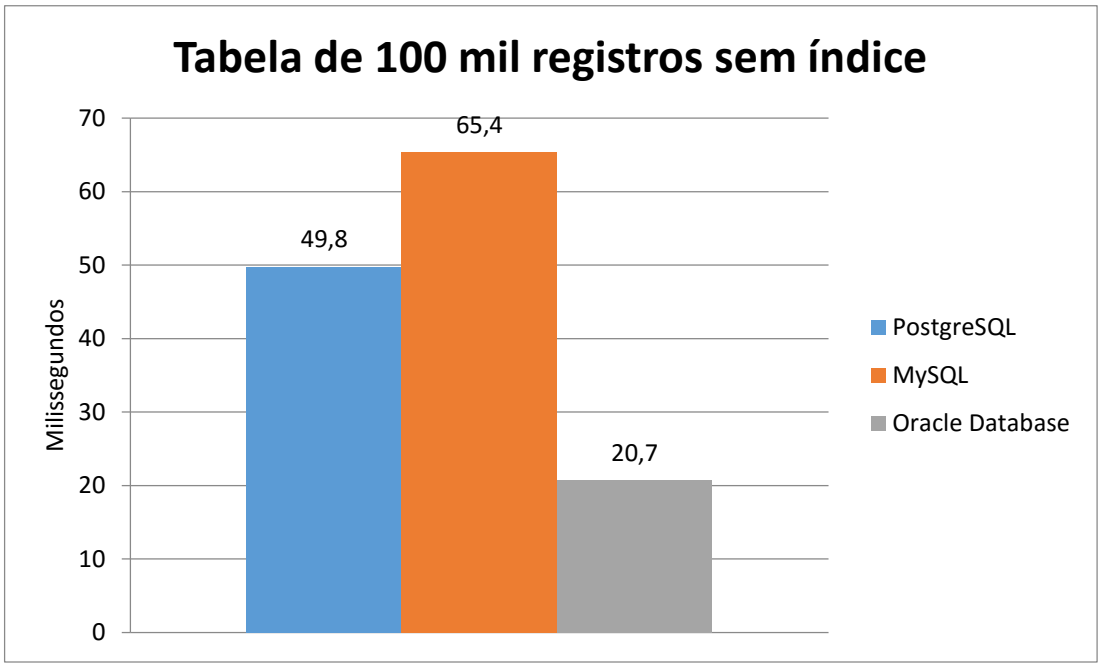


Fonte: Desenvolvido pelo autor.

4.6.2 Segunda consulta

Após a execução da segunda consulta, sem o índice, onde temos, além dos dados da tabela de cem mil registros, um inner join, é possível que notar que o Oracle Database, que na primeira consulta com cem mil registros foi o pior, teve um rendimento muito acima da média dos outros dois SGBD's. Com uma média de execução de 20,7 milissegundos, o Oracle Database obteve um desempenho 58,44% melhor que o PostgreSQL (49,8 milissegundos) e 68,35% melhor que o MySQL (65,4 milissegundos). O PostgreSQL foi superior ao MySQL em 31,32%. É possível ver comparativo de desempenho entre os três SGBD's analisados logo abaixo, no Gráfico 9.

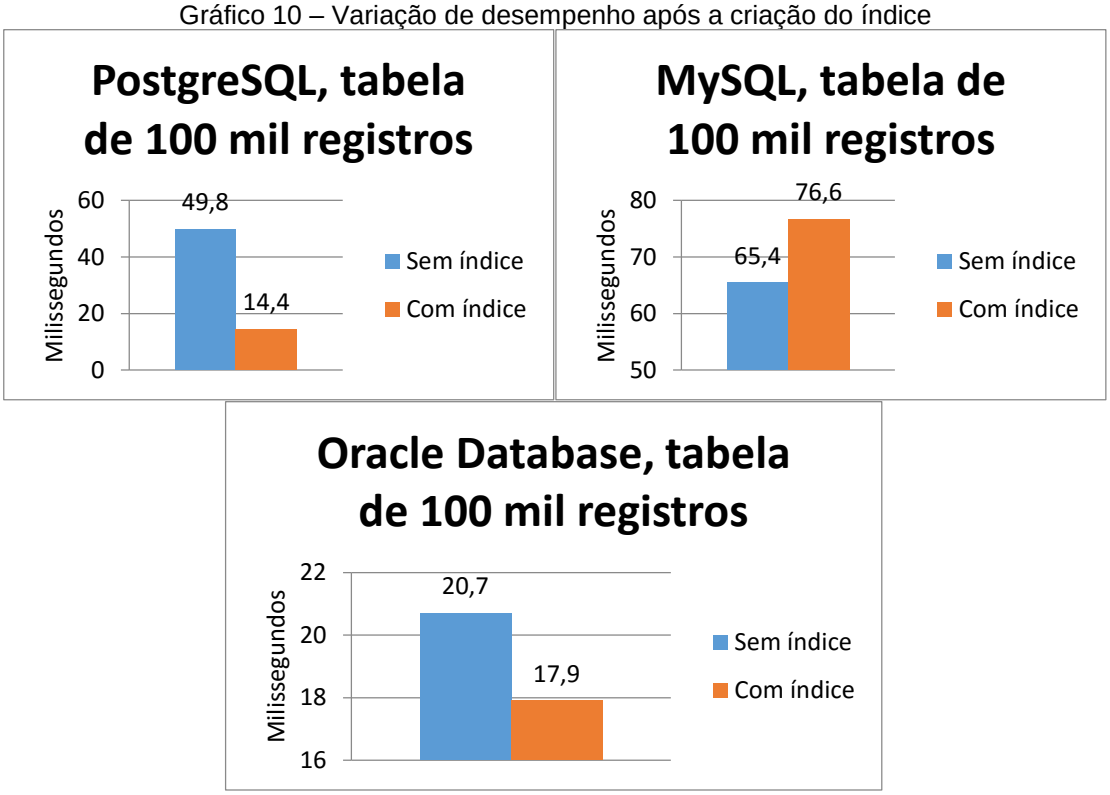
Gráfico 9 – Primeira consulta na tabela não indexada com 100 mil registros.



Fonte: Desenvolvido pelo autor.

Executando a mesma consulta acima, desta vez com índice, o PostgreSQL apareceu como o SGBD com o melhor desempenho médio. O PostgreSQL obteve os resultados da consulta indexada 346,52% mais rápido do que a consulta sem índice, o que fez com que o resultado fosse 24,3% melhor que o desempenho do Oracle Database e 531,9% melhor que o MySQL. O Oracle Database conseguiu aumentar sua performance em 13,52% em relação à consulta não indexada. Já o MySQL, piorou seu desempenho, deixando de retornar os registros em 65,4 milissegundos para retornar em 76,6 milissegundos, um desempenho 17,12% inferior ao resultado obtido anteriormente. A variação do desempenho de cada SGBD após a criação do índice pode ser conferida no gráfico 10.

Gráfico 10 – Variação de desempenho após a criação do índice

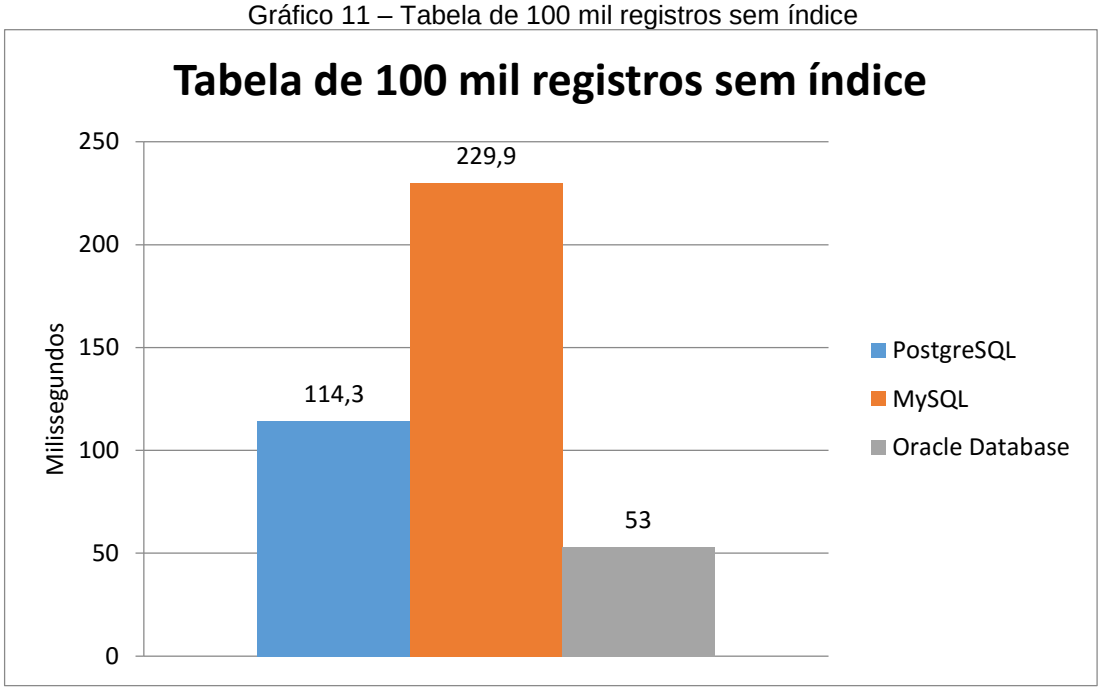


Fonte: Desenvolvido pelo autor.

4.6.3 Terceira consulta

Os resultados dos testes da terceira consulta à tabela sem índices com cem mil registros, que utiliza também o comando "inner join" para selecionar dados de outras tabelas, mostraram que o Oracle Database teve o melhor desempenho, tendo uma média de execução de 53 milissegundos, menos da metade da média do PostgreSQL, que foi de 114,3 milissegundos, e 76% mais rápido que o MySQL, que obteve uma média de 229,9 milissegundos. O MySQL foi 50% mais lento que o PostgreSQL, se tornando assim o sistema gerenciador de banco de dados mais lento neste quesito, como pode ser visualizado no gráfico 11.

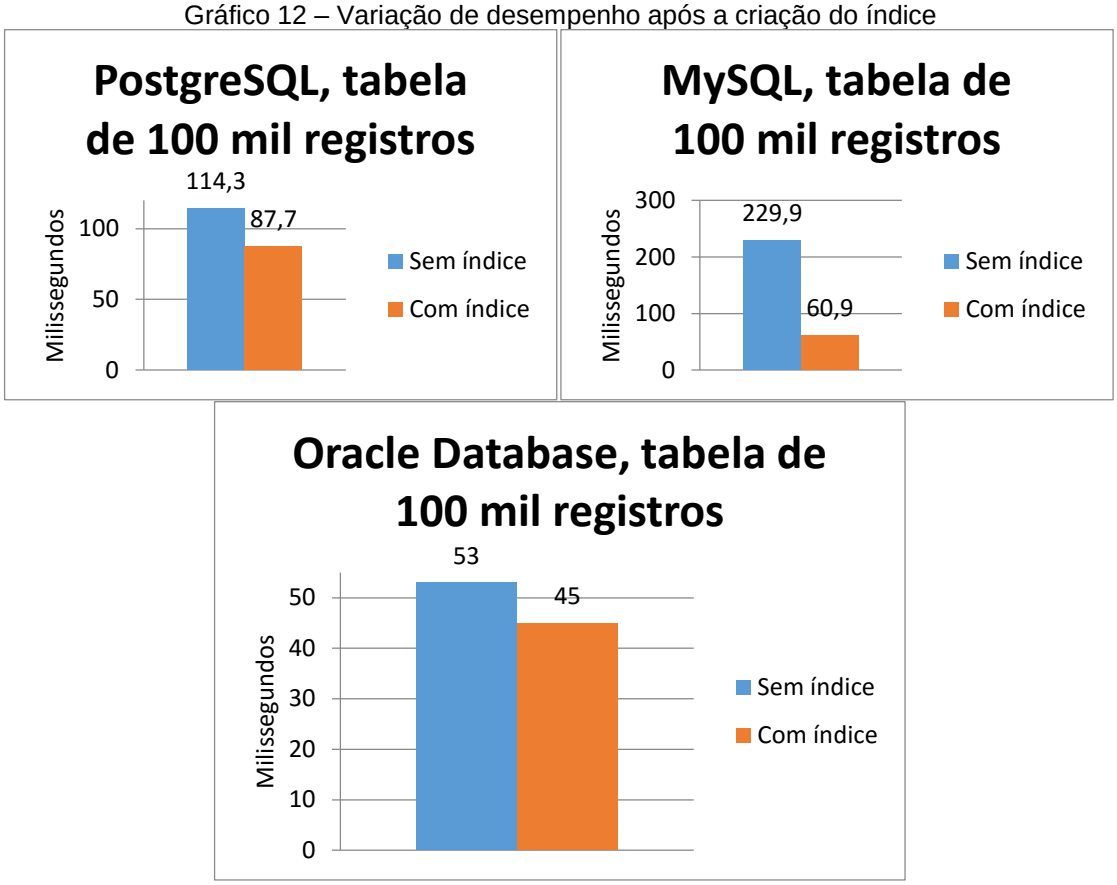
Gráfico 11 – Tabela de 100 mil registros sem índice



Fonte: Desenvolvido pelo autor.

A nova bateria de testes que foi realizada utilizando a terceira consulta à tabela com cem mil registros, desta vez com índice, fez com que o MySQL melhorasse em 73% o seu desempenho (passou de 229,9 milissegundos para 60,9), fazendo com que o tempo médio da execução fosse 30% menor do que o tempo médio do PostgreSQL, que teve a média de 87,7. O Oracle Database teve uma média de 45 milissegundos e, mesmo após a criação do índice, continuou sendo o SGBD mais rápido. Sua performance foi 26% melhor que a do MySQL, e 48% melhor que a do PostgreSQL. Essa variação de desempenho pode ser conferida no gráfico 12.

Gráfico 12 – Variação de desempenho após a criação do índice



Fonte: Desenvolvido pelo autor.

4.7 Resultados Obtidos: Tabela de Um Milhão de Registros

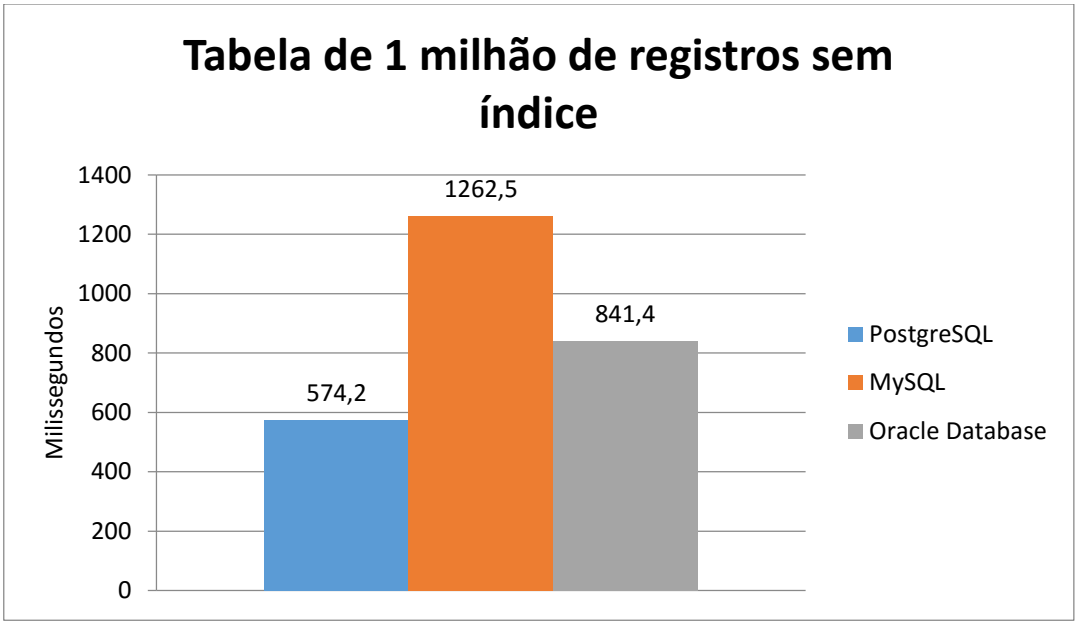
As consultas que envolviam a tabela com um milhão de registros foram executadas pelos sistemas de banco de dados com relativa lentidão, mas nada absurdamente devagar. A demora ao retornar o resultado da consulta se deu mesmo pela quantidade de registros da tabela, e quando os índices foram criados, houve uma otimização significativa do desempenho. A seguir é possível conferir uma análise dos resultados para essa situação.

4.7.1 Primeira consulta

Assim como as outras consultas houve também uma diferença interessante no desempenho dos sistemas testados. Quando o comando foi executado sem nenhum índice criado o PostgreSQL obteve um média de 574,2 milissegundos de tempo gasto para executar a operação. O MySQL ficou com média de 1262,5 milissegundos e o Oracle Database gastou em média 841,4 milissegundos. Em

números o PostgreSQL executou a consulta sem índice cerca de 31% mais rápido que o Oracle Database e 54% mais rápido que o MySQL. O Oracle Database foi cerca de 33% mais rápido que o MySQL. Os números podem ser melhor visualizados no gráfico 13.

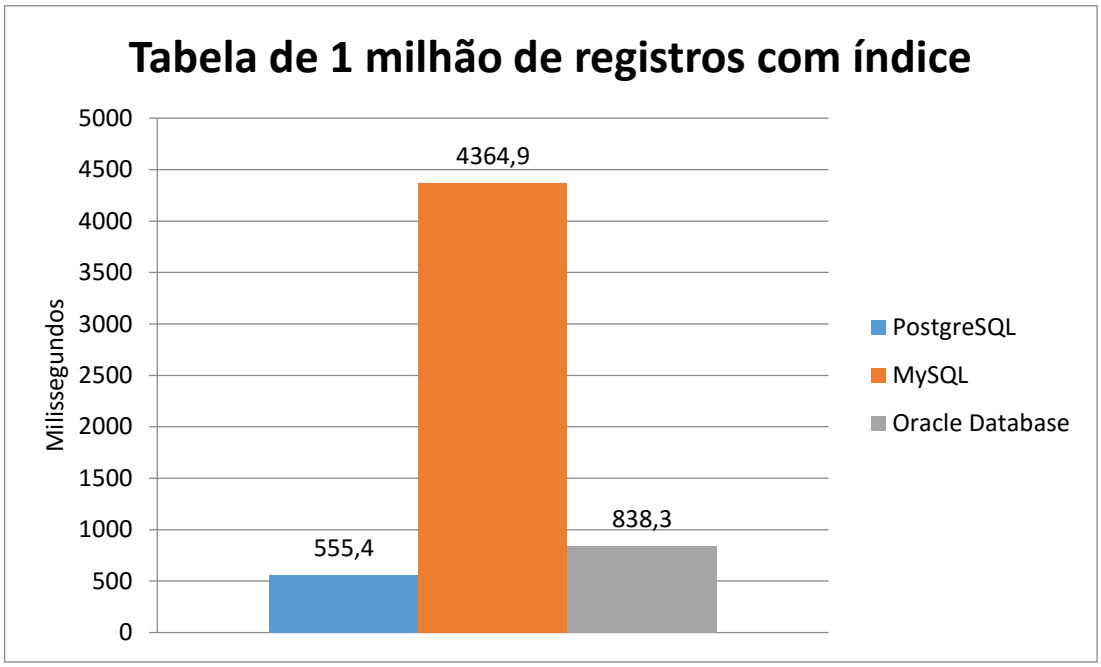
Gráfico 13 – Tabela de 1 milhão de registros sem índice



Fonte: Desenvolvido pelo autor.

Ao criar o índice, os sistemas de banco de dados testados foram mais eficientes na consulta, exceto o MySQL que demorou mais a realizar o comando após a criação do índice. Assim, a diferença de desempenho entre os SGBD's testados após a criação do índice variou um pouco em relação às consultas executadas sem nenhum índice, conforme o gráfico 14.

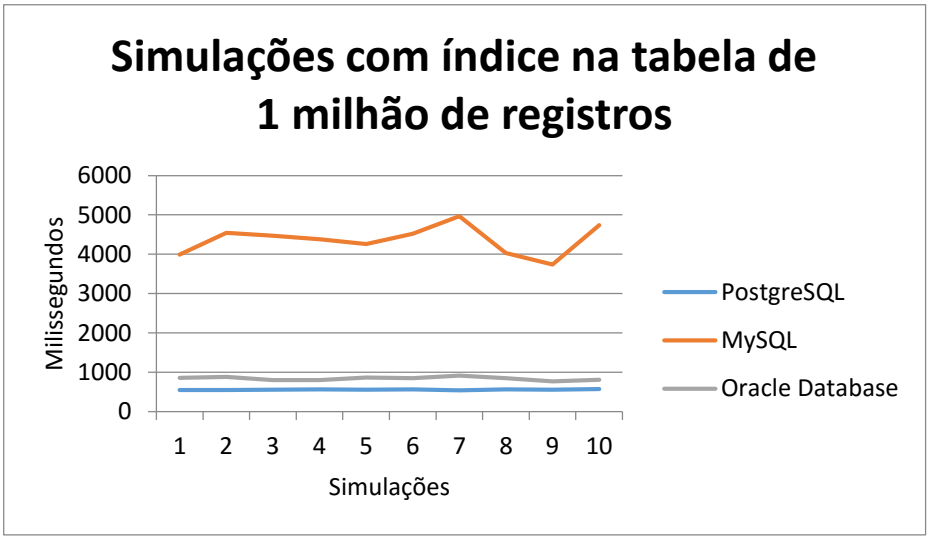
Gráfico 14 – Tabela de 1 milhão de registros com índice



Fonte: Desenvolvido pelo autor.

A variação do desempenho de cada sistema antes e depois da criação do índice foi significativa para os três sistemas, sendo que apenas o MySQL apresentou uma diferença negativa após a criação do índice. Essa anomalia do MySQL pode ser conferida visualmente no gráfico 15, que mostra a diferença expressiva de tempo entre os SGBD's em cada uma das 10 consultas.

Gráfico 15 – Simulações com índice na tabela de 1 milhão de registros.

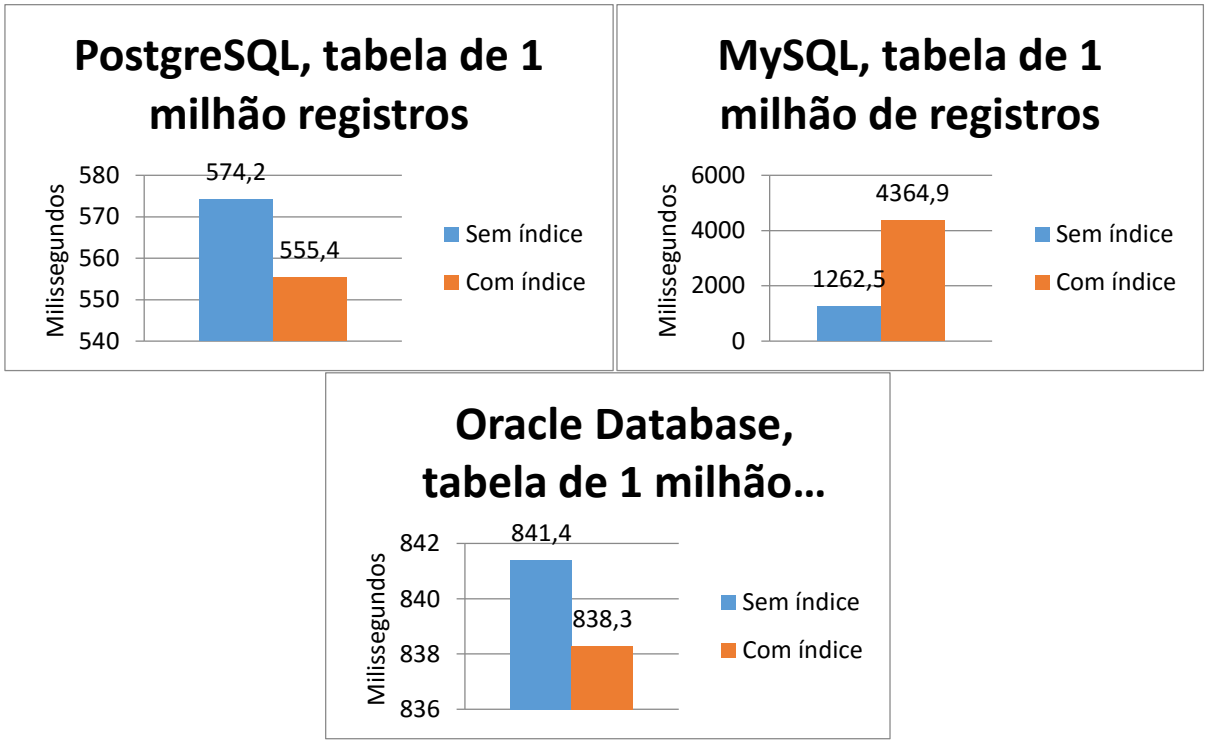


Fonte: Desenvolvido pelo autor.

O gráfico 16 mostra a diferença de desempenho antes e depois da criação do índice

em cada sistema de banco de dados, onde é possível visualizar o quanto a consulta foi otimizada no PostgreSQL e no Oracle Database e o quanto a consulta foi prejudicada usando o MySQL.

Gráfico 16 – Variação do desempenho após a criação do índice



Fonte: Desenvolvido pelo autor.

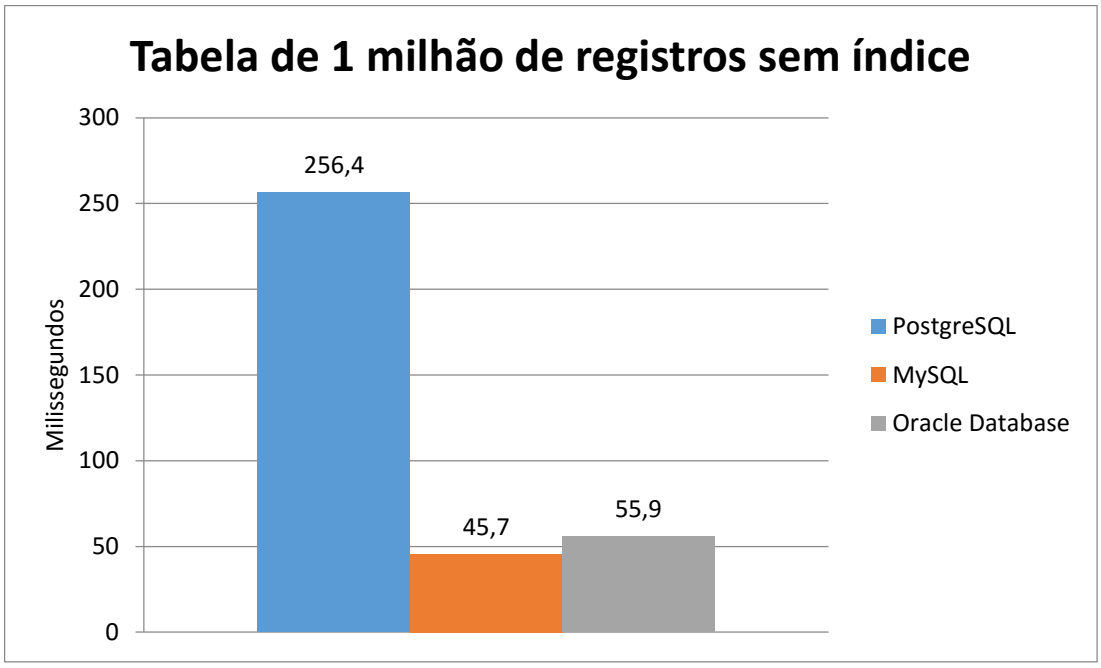
A seguir é possível conferir a análise feita sobre os resultados obtidos pela segunda consulta elaborada.

4.7.2 Segunda consulta

Na segunda consulta executada, houve mais uma vez uma anomalia referente ao tempo gasto para fazer a consulta por parte do MySQL, ou seja, novamente a consulta demorou mais tempo para ser executada com a existência do índice do que sem ela. Quando executada a consulta sem nenhum índice o PostgreSQL obteve uma média de 256,4 milissegundos de tempo gasto para realizar a operação, o MySQL em média cerca de 45,7 milissegundos, uma diferença muito significativa em relação ao PostgreSQL, e o Oracle gastou em média 55,7 milissegundos para executar o comando. É possível dizer então que o MySQL foi cerca de 82% mais rápido que o PostgreSQL e 18% mais rápido que o Oracle Database, que vinha

tendo os melhores desempenhos até aqui. O gráfico 17 mostra visualmente a diferença de desempenho de cada SGBD.

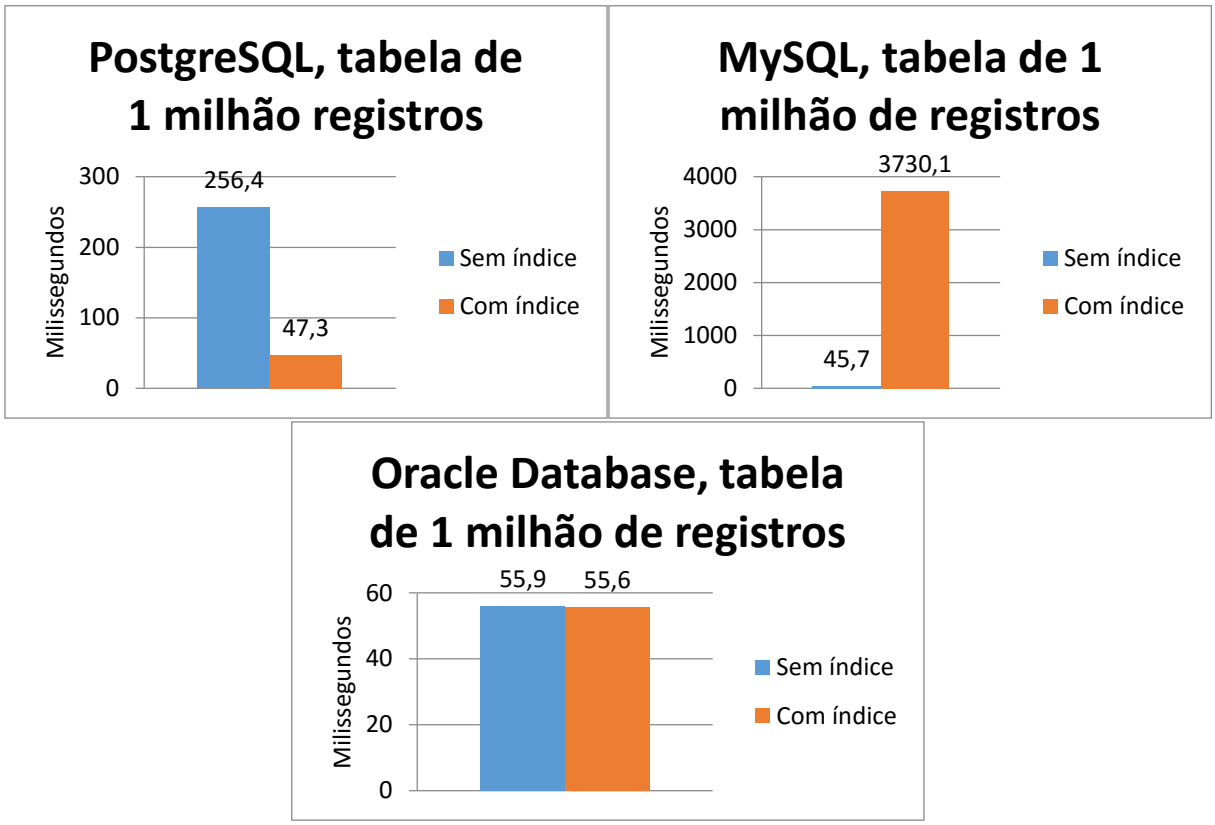
Gráfico 17 – Tabela de 1 milhão de registros sem índice



Fonte: Desenvolvido pelo autor.

Após a criação do índice os tempos gastos para cada SGBD variaram um pouco. O PostgreSQL otimizou sua consulta em quase 82%, obtendo uma média agora de 47,3 milissegundos para executar a segunda consulta. O Oracle Database não mostrou diferenças significativas de desempenho e o MySQL novamente apresentou uma piora do seu desempenho com índice em cerca de 98%. É possível conferir visualmente esses resultados no gráfico 18.

Gráfico 18 – Variação de desempenho após a criação do índice.

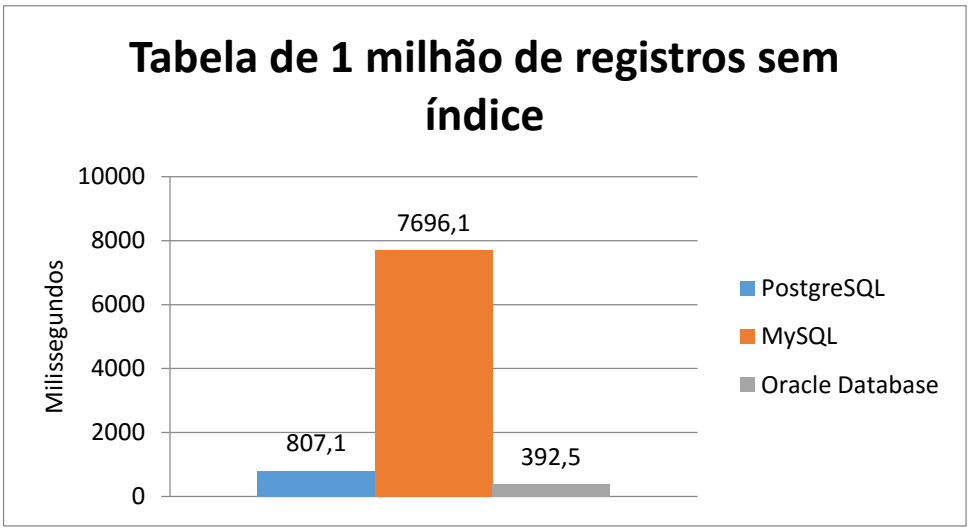


Fonte: Desenvolvido pelo autor.

4.7.3 Terceira consulta

Na terceira consulta executada todos os sistemas de banco de dados se comportaram da forma como esperada, que a otimização da consulta a pós a criação de um índice. Quando a consulta foi executada sem nenhum índice criado o PostgreSQL levou em média 807,1 milissegundos para executar a operação. O MySQL gastou em média 7696,1 milissegundos e o Oracle Database média 392,5 milissegundos. Para expressar melhor os resultados, o Oracle Database foi cerca de 51% mais rápido que o PostgreSQL, e aproximadamente 94% mais rápido que o MySQL. O Oracle Database volta a ter o melhor desempenho nesse teste específico, obtendo resultados bem superiores aos outros SGBD's testados. O gráfico 19 contém a representação visual desses resultados.

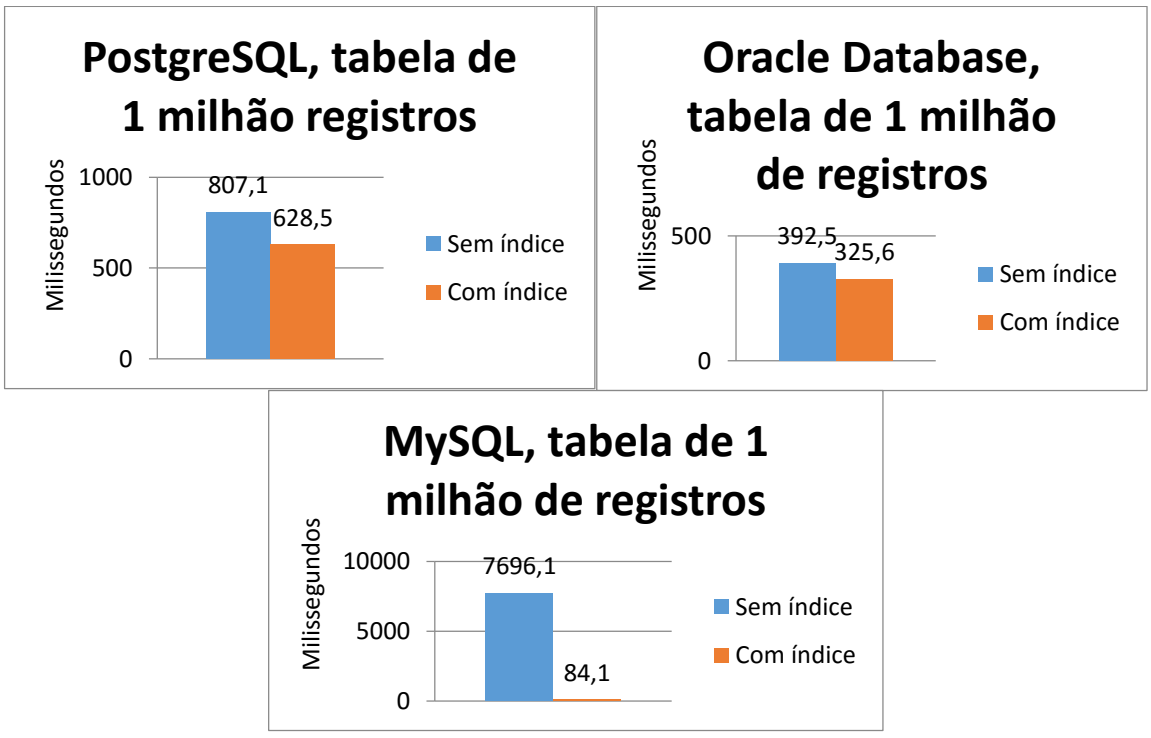
Gráfico 19 – Tabela de 1 milhão de registros sem índice.



Fonte: Desenvolvido pelo autor.

Após a criação do índice, todos os SGBD's otimizaram suas consultas, inclusive o MySQL que vinha apresentando algumas anomalias em que a consulta com o índice demorava mais do que a consulta sem índice. O PostgreSQL otimizou sua consulta em 22%, o MySQL otimizou incríveis 98% da consulta e o Oracle Database otimizou cerca de 17%. O gráfico 20 representa esses resultados.

Gráfico 20 – Variação de desempenho após a criação do índice.



Fonte: Desenvolvido pelo autor.

5 ESTUDOS FUTUROS

Os bancos de dados relacionais são adotados como principal solução de armazenamento de dados desde que o modelo relacional de dados foi proposto por Edgar Codd, um pesquisador da IBM, em torno de 1970. Com a popularização das redes sociais, lojas virtuais e vários portais que armazenam bilhões de dados em suas bases, os bancos de dados relacionais foram apresentando problemas de desempenho, principalmente no que diz respeito a escalabilidade. O Walmart por exemplo, uma grande loja virtual, faz mais de 1 bilhão de transações de clientes por hora e seus bancos de dados possuem mais de 2,5 petabytes. O Facebook, a rede social mais usada no mundo, possui em seu banco de dados aproximadamente 40 bilhões de fotos. Estes dois exemplos mostram que o volume de dados e de informações presentes no mundo vem crescendo rapidamente. É justamente esse o maior problema dos bancos de dados relacionais, a dificuldade de usar esse modelo com uma demanda por escalabilidade crescente.

Com o fato de os dados estarem crescendo exponencialmente a solução encontrada por pesquisadores foi escalar o banco de dados em sistemas distribuídos, o que se tornou uma dificuldade para administradores de bancos de dados relacionais, pois precisavam particionar dados em máquinas diferentes, e manipular tabelas em servidores diferentes pode ser muito problemático. Por conta desses problemas, as soluções não relacionais começaram a surgir.

Assim, desenvolvedores começaram a discutir um modelo alternativo para solução de problemas e dificuldades enfrentadas nos modelos relacionais. O objetivo era desenvolver uma nova maneira de armazenar informações flexibilizando o banco de dados para as características particulares de cada organização. Flexibilidade essa que é essencial para atender os requisitos de sistemas distribuídos. O NoSQL surgiu dessa maneira.

O termo NoSQL foi usado primeiramente em 1998, por Carlos Strozzi, e se tratava de um nome para um banco de dados relacional que era leve, de código livre e que não possuía interface para SQL. Em 2009, o nome foi usado para descrever um modelo de banco de dados que era capaz de ajustar os dados quando o modelo

relacional não atendia aos requisitos pretendidos. O NoSQL não veio para substituir o modelo relacional tradicional, e sim para criar uma alternativa para organizações que precisam de um modelo de banco de dados que seja flexível e que suporte seu volume de dados.

Mesmo que existam muitos bancos de dados NoSQL, o movimento para usá-lo somente ganhou força quando algumas empresas famosas começaram a utilizar implementações próprias de bancos de dados NoSQL para fornecer serviços para seus sistemas distribuídos. O Google, por exemplo, criou o Big Table, a Amazon criou o Dynamo, em 2008 o Facebook desenvolveu o Cassandra para tratar seu grande volume de dados. O Cassandra se mostrou tão eficiente que o Twitter deixou o MySQL para utilizar o banco de dados desenvolvido pelo Facebook.

Considerando todo esse contexto histórico e funcional apresentado sobre o NoSQL, é possível compor um trabalho futuro que relacione as funcionalidades dos dois paradigmas de sistemas de banco de dados, bem como o uso de testes de carga e de seleção de dados, como feito no presente trabalho, a fim de detectar situações em que seja mais apropriado o uso de um ou outro tipo de banco de dados

6 CONCLUSÃO

Ao realizar os testes nos sistemas de banco de dados testados neste trabalho, foi possível observar que cada sistema implementa sua busca de dados no comando SELECT de forma diferente, pois os tempos de cada consulta variaram significativamente de um SGBD para outro. Cada um possui uma estrutura particular e é isso que enriquece o vasto mundo dos SGBD's, ofertando opções de escolha para cada situação.

Considerando os fatores citados anteriormente, este trabalho realizou um estudo sobre o comportamento de três dos muitos SGBD's existentes em consultas de seleção de dados que utilizaram tabelas de diferentes tamanhos, e posteriormente um índice, que é uma estrutura conhecida como uma forma de melhorar o desempenho de consultas envolvendo o campo indexado.

Os três sistemas tiveram um desempenho esperado quando a consulta foi realizada sem nenhum índice criado. O destaque ficou com o Oracle Database que obteve o melhor desempenho em quase todas as situações, seguido do PostgreSQL e depois o MySQL. Algumas anomalias foram detectadas durante os testes como, por exemplo, a consulta executada sobre um campo indexado levar mais tempo para retornar o resultado do que uma consulta executada sobre um campo não indexado. Isso favorece que se aplique futuramente um estudo a fim de detectar quais os motivos de tal fato ter acontecido.

De acordo com os resultados dos testes, o Oracle Database tem um bom desempenho quase todas as situações, se destacando entre os demais principalmente quando a quantidade de registros nas tabelas for muito grande, como um milhão de registros por exemplo. O PostgreSQL segue o Oracle Database de perto em algumas situações, sofrendo pouca diferença. É possível destacar a qualidade do PostgreSQL em consultas de seleção de dados envolvendo uma só tabela, a qual o seu desempenho foi superior ao de todos os outros SGBD's testados. O MySQL apresentou os melhores resultados quando a consulta foi realizada sobre um campo indexado em tabela de tamanho grande, em torno de um milhão de registros.

Observando os fatores citados é possível perceber que cada SGBD se saiu melhor em uma situação específica. Para fins de comparação, no trabalho de conclusão de curso de Lucas Zucoloto Felipe e Welesson Pereira Lopes em 2013 na Faculdade do Espírito Santo – Unes, que falou sobre o desempenho de banco de dados utilizando práticas de normalização e desnormalização no PostgreSQL e no MySQL, o modelo que foi até certo ponto desnormalizado teve um melhor desempenho em algumas situações, o que não garantiu que isso sempre acontecerá, pois em outros testes o modelo em um nível de normalização maior o desempenho foi praticamente igual. Por conseguinte foi expresso no trabalho citado que nos dois SGBD's o melhor custo benefício foi quando os testes foram executados em um modelo na segunda forma normal.

Portanto, ao analisar os resultados dos dois trabalhos é possível estabelecer uma relação direta, pois ambos testaram o desempenho de sistemas de banco de dados usando apenas uma metodologia diferente, o que agrega valor a pesquisa e enriquece um pouco mais a gama de informações existentes sobre desempenho de SGBD's, o que é fundamental para uma empresa que busca um software de banco de dados que atenda suas necessidades específicas em seu ambiente de dados particular.

7 REFERÊNCIAS

CAMPOS, Augusto. **O que é software livre**. BR-Linux. Florianópolis, março, 2006.

CHANDLER, Alfred Dupont. **O Século Eletrônico:** a história da evolução da indústria eletrônica e de informática. Campus, 2003.

DATE, C. J. **Introdução a Sistemas de Banco de Dados**. 8. ed. São Paulo: Elsevier, 2004. 871 p.

DOCUMENTAÇÃO do MySQL. 2014. Disponível em: <http://dev.mysql.com/doc/>. Acesso em: 02 dez. 2014.

DOCUMENTAÇÃO do PostgreSQL 8.0.0: ANALYZE. 2014. Disponível em: <http://pgdocptbr.sourceforge.net/pg80/sql-analyze.html>. Acesso em: 16 set. 2014.

DROZDEK, Adam. **Estrura de Dados e Algoritmos em C++.** São Paulo: Pioneira Thomson Learning, 2005. 573 p.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de Banco de Dados.** 6. ed. São Paulo: Pearson Addison Wesley, 2011. 788 p.

HALPIN, Terry. **Information modeling and relational databases**: From Conceptual Analysis to Logical Design. San Francisco: Morgan Kaufmann Publishers, 2001. 763 p.

MACHADO, Carlos; Cruz, Luiz. **O MySQL ganha músculos**. Info Exame, São Paulo, n. 27, abr. 2006.

MACHADO, Felipe Nery Rodrigues. **Banco de dados:** projeto e implementação. 2. ed. São Paulo: Érica, 2011. 398 p.

MILANI, André. **PostgreSQL**: Guia do Programador. São Paulo: Novatec, 2008. 392p.

NEVES, Pedro M. C.; RUAS, Rui P. F. **O guia prático do mysql.** 1. ed. Lisboa: Centro Atlântico, 2005. 406 p.

ROB, Peter; CORONEL, Carlos. **Sistemas de Banco de Dados**: Projeto, Implementação e Administração. 8. ed. São Paulo: Cengage Learning, 2011. 711 p.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S.. **Sistema de Banco de Dados**. 3. ed. São Paulo: Pearson Makron Books, 1999. 778 p.

SILVA, Helio. **Missão crítica é para Postgre**. Info Exame, São Paulo, n. 27, abr.

2006.

STEPHENS, Rod. **Beginning Database Design Solutions**. Indianapolis: Wiley Publishing, Inc., 2009. 552 p.

ZIVIANI, Nivio. **Projeto de Algoritmos:** Com implementações em Pascal e C. 4. ed. São Paulo: Pioneira, 1999. 267 p.