UNIVERSIDADE FEDERAL DO PAMPA
CIÊNCIA DA COMPUTAÇÃO

MIGUEL MOURA ANCHIETA

SINTONIA EM BANCO DE DADOS ATRAVÉS DO PARTICIONAMENTO DE TABELAS

Trabalho de Conclusão de Curso

Alegrete

2012

MIGUEL MOURA ANCHIETA

SINTONIA EM BANCO DE DADOS ATRAVÉS DO PARTICIONAMENTO DE TABELAS

Trabalho de Conclusão de Curso apresentado como parte das atividades para obtenção do título de bacharel em Ciência da Computação na Universidade Federal do Pampa.

Orientador: Prof. Dr. Sergio Luis Sardi Mergen

Co-orientador: Prof. Me. Sergio Luis Dill

Alegrete

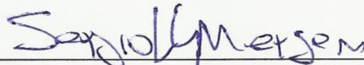
2012

MIGUEL MOURA ANCHIETA

SINTONIA EM BANCO DE DADOS ATRAVÉS DO PARTICIONAMENTO DE TABELAS

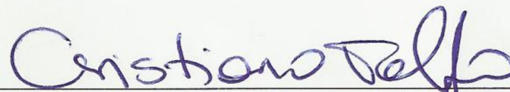
Trabalho de Conclusão de Curso apresentado como parte das atividades para obtenção do título de bacharel em Ciência da Computação na Universidade Federal do Pampa.

Trabalho apresentado e aprovado em: 04 de Janeiro de 2012.
Banca Examinadora:

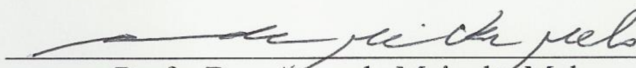


Prof. Dr. Sérgio Luis Sardi Mergen
Orientador
Ciência da Computação - UNIPAMPA

Msc. Sérgio Luis Dill
Coorientador



Prof. Msc. Cristiano Tolfo
Ciência da Computação - UNIPAMPA



Profa. Dra. Amanda Meincke Melo
Ciência da Computação - UNIPAMPA

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus. Tenho a certeza de que sem ele não estaria completando mais esta etapa.

A toda minha família que, de uma forma ou de outra, contribuiu na minha formação. A cada um de vocês, o meu agradecimento.

Agradecimento especial a minha mãe, a minha heroína! Sempre presente, sempre apoiando.

Aos meus amigos, irmãs de todas as horas: Amanda e Isabel. Obrigado por cada etapa vencida.

Ao professor Sergio Dill pela contribuição inicial. Ao professor Sergio Mergen pelo grande auxílio e por aceitar ser orientador na etapa final do trabalho. A todos os meus professores.

A todos vocês, meu muito obrigado!

RESUMO

A sintonia em banco de dados é o processo que visa melhorar o desempenho em sistemas de banco de dados. Esta prática possui vários aspectos e métodos a serem considerados, dentre estes, surge o particionamento de tabelas que contribui no desempenho e administração de um banco de dados através da segmentação de tabelas em fatias menores, proporcionando um menor tempo de acesso e manutenção. Como objetivo deste trabalho, está a realização de um estudo da aplicação de particionamento como uma técnica de sintonia. Par isso, fez-se necessário a revisão da literatura para contextualizar o tema, além do estudo de ferramentas como o sistema gerenciador de banco de dados PostgreSQL para auxiliar na elaboração dos experimentos. Ao todo, foram realizados três experimentos visando o comparativo de desempenho entre tabelas particionadas e não particionadas. Para a realização destes, foram elaborados cenários e utilizadas ferramentas para população das tabelas com dados de teste. Os experimentos realizados permitiram verificar casos em que há ganho de desempenho quando a estratégia de particionamento é verificada.

Palavras-chave: Banco de Dados, Sintonia, Particionamento, PostgreSQL.

ABSTRACT

The tuning of the database is the process that aims to improve performance in database systems. This practice has several aspects and methods to consider, among these, there is the partitioning of tables that contributes to the performance and administration of a database table by segmenting into smaller slices, providing a lower access time and maintenance. Objective of this work, is a study of the application partitioning as a technique for tuning. Pair this, it was necessary to review the literature to contextualize the topic, as well as study tools such as management system PostgreSQL database to assist in the preparation of experiments. In all, three experiments were performed aiming at the comparative performance between partitioned and nonpartitioned tables. To achieve these, scenarios were developed and used tools for population of the tables with test data. The experiments allowed us to verify cases where there is performance gain when the partitioning strategy is verified.

Keywords: Database, Tuning, Partitioning, PostgreSQL.

LISTA DE FIGURAS

FIGURA 1 - Fases da Sintonia 16
FIGURA 2 - Problemas Relacionados 17
FIGURA 3 – Triângulo do Desempenho 20
FIGURA 4 - Processamento de uma Consulta 31
FIGURA 5 - Tabela de dados sem nenhum tipo de particionamento 36
FIGURA 6 - Consulta em tabela sem nenhum tipo de particionamento 36
FIGURA 7 - Consulta em tabela distribuída em 3 partições 37
FIGURA 8 - Tabela Particionada em Tablespaces Diferentes 42
FIGURA 9 - Scan de uma consulta ao mês de fevereiro em uma tabela particionada 47
FIGURA 10 - Comparação de Custo Computacional Utilizando em Tabelas 48
FIGURA 11- Criação da Tabela VENDA 53
FIGURA 12 - Inherits Referente a Janeiro 53
FIGURA 13 - Trigger Referente a Janeiro e Fevereiro 54
FIGURA 14 - Modelo Conceitual Banco de Dados Venda 55
FIGURA 15 - Partição na Tabela Venda 57
FIGURA 16 - Consulta com Filtro por Data 61
FIGURA 17 – Consulta com Operador de Comparação “maior igual” 63
FIGURA 18 – Consulta com União 65

LISTA DE TABELAS

TABELA 1 - Exemplos de Condições de Pesquisa 23
TABELA 2 - Pontuação por Operador 24
TABELA 3 - Pontuação por Operando 24
TABELA 4 - Tabela Venda Sem Particionamento 32
TABELA 5 - Partição Referente à Janeiro da Tabela Venda 33
TABELA 6 - Partição Referente à Fevereiro da Tabela Venda 33
TABELA 7 - Partição Referente à Março da Tabela Vend 33
TABELA 8 - Partição Referente à Abril da Tabela Venda 33
TABELA 9 - Partição Referente à Maio da Tabela Venda 34
TABELA 10 - Partição Referente à Julho da Tabela Venda 34
TABELA 11 - Partição Referente à Julho da Tabela Venda 34
TABELA 12 - Partição Referente à Agosto da Tabela Venda 34
TABELA 13 - Partição Referente à Setembro da Tabela Venda 35
TABELA 14 - Partição Referente à Outubro da Tabela Venda 35
TABELA 15 - Dados Suportados no Particionamento. 40
TABELA 16 - Dados Não Suportados no Particionamento. 41
TABELA 17- Modelo Relacional: CLIENTE 55
TABELA 18 - Modelo Relacional: VENDA 56
TABELA 19 - Modelo Relacional: VENDEDOR 56
TABELA 20 - Modelo Relacional: RECLAMAÇÃO 57
TABELA 21 - Ferramentas de Inserção de Dados 58
TABELA 22 - Número de Tuplas X Tamanho 59
TABELA 23 - Comparativo de Consulta a Mil Tuplas 70
TABELA 24 - Comparativo de Consulta a 10 Mil Tuplas 70
TABELA 25 - Comparativo de Consulta a 100 Mil Tuplas 70
TABELA 26 - Comparativo de Consulta a 1 Milhão Tuplas 70
TABELA 27 - Comparativo de Consulta a 10 Milhões Tuplas 71
TABELA 28 - Comparativo de Consulta a Mil Tuplas 71
TABELA 29 - Comparativo de Consulta a 10 Mil Tuplas 71
TABELA 30 - Comparativo de Consulta a 100 Mil Tuplas 71
TABELA 31 - Comparativo de Consulta a 1 Milhão Tuplas 71
TABELA 32 - Comparativo de Consulta a 10 Milhões Tuplas 72
TABELA 33 - Comparativo de Consulta a Mil Tuplas 72
TABELA 34 - Comparativo de Consulta a 10 Mil Tuplas 72
TABELA 35 - Comparativo de Consulta a 100 Mil Tuplas 72
TABELA 36 - Comparativo de Consulta a 1 Milhão Tuplas 72
TABELA 37 - Comparativo de Consulta a 10 Milhões Tuplas 73

LISTA DE GRÁFICOS

GRÁFICO 1 – Consulta com Filtro por Data Com e Sem Particionamento 61
GRÁFICO 2 - Consulta com Filtro por Data Inicial.. 63
GRÁFICO 3- Consulta Utilizando União.. 66

LISTA DE ABREVIATURAS E SIGLAS

BD: *Banco de Dados*

BSD: *Berkeley Software Distribution*

CPU: *Central Processing Unit (Unidade Central de Processamento)*

DBA: *Database Administrator (Administrador de Banco de Dados)*

DPF: *Data Partition Feature*

DW: *Data Warehouse (Armazém de dados)*

GB: *Gigabyte*

HD: *Hard Drive (Disco Rígido)*

I/O: *Input/Output (Entrada/Saída)*

IDE: *Integrated Drive Electronics*

KB: *Kilobyte*

MB: *Megabyte*

MHZ: *Mega Hertz*

ORD: *Banco de Dados Objeto-Relacional*

RAID: *Redundant Array of Independent Drives (Conjunto Redundante de Discos Independentes)*

RAM: *Random Acess Memory (Memória de Acesso Aleatório)*

SATA: *Serial AT Attachment*

SCSI: *Small Computer System Interface*

SGBD: *Sistema Gerenciador de Banco de Dados*

SGBDOR: *Sistema de Gerenciamento de Dados Objeto Relacional*

SO: *Sistema Operacional*

SQL: *Structured Query Language (Linguagem de Consulta Estruturada)*

TB: *Terabyte*

TI: *Tecnologia da Informação*

SUMÁRIO

RESUMO 5
ABSTRACT 6
LISTA DE FIGURAS 7
LISTA DE TABELAS 8
LISTA DE GRÁFICOS 9
LISTA DE ABREVIATURAS E SIGLAS 10
1 INTRODUÇÃO 14
1.1 Motivação 14
1.2 Objetivo 14
1.3 Metodologia 15
1.4 Organização 15
2. SINTONIA EM BANCO DE DADOS 16
2.1. Problemas de Hardware 18
2.2 Configuração de Parâmetros 20
2.3 Sistema Operacional 21
2.3.1 Sistema Operacional Linux 21
2.3.2 Sistema Operacional Windows 22
2.4 Sintonia de Instruções SQL 22
2.4.1 Análise da Consulta 23
2.4.2 Otimizador de Consulta 30
2.5 Considerações do Capitulo 31
3 PARTICIONAMENTO DE TABELAS 32
3.1 Partição Horizontal 38
3.2 Partição Vertical 38
3.3 Quando Particionar? 39
3.4 Chave de Partição 39
3.5 Cláusula de Particionamento 41
3.6 Partição de Dados 42
3.7 Escolhendo a Tabela 43
3.7.1 Tabelas de Sistemas Transacionais 43

3.7.2 Tabelas de Sistemas Analíticos 43
3.8 Quando Aplicar o Particionamento de Tabela 44
3.9 Definição do Critério de Particionamento 45
3.10 Benefícios do Particionamento de Tabelas 47
3.10.1 Gerenciamento Aprimorado 47
3.10.2 Desempenho de Consulta Aprimorado 47
3.10.3 Otimização dos Custos de Armazenamento 48
3.10.4 Capacidade Ampliada de Tabelas 49
3.10.5 Flexibilidade na Alocação dos Índices 49
3.10.6 Contenção do Impacto a Falhas 49
3.11 Desvantagens do Particionamento de Tabelas 50
4 PARTICIONAMENTO EM POSTGRESQL 51
4.1 Particionamento 52
4.1.1 Particionamento por Intervalo 52
4.1.2 Particionamento por Lista 52
4.2 Implementando o Particionamento 52
5. PROJETO E ANÁLISE DO EXPERIMENTO 55
5.1 Modelagem do Banco de Dados 55
5.1.1 Estratégia de Particionamento 57
5.2 Configurações 58
5.2.1 Carga de Dados 58
5.2.2 Condução dos testes 59
5.2.3 Plataformas utilizadas 59
5.3 Plano de Testes 60
5.3.1 Experimento um: Consulta com Filtro por Data 60
5.3.2 Experimento dois: Consulta com Filtro usando operador de comparação “Maior Igual” 62
5.3.3 Experimento três: Consulta Utilizando União 64
6 CONSIDERAÇÕES FINAIS 67
REFERÊNCIAS 68
ANEXOS 70
Anexo 1: Resultados da Consulta com Filtro por Data 70
Anexo 2: Resultado consulta com Filtro usando operador de comparação “Maior Igual” 71

Anexo 3: Resultados da Consulta Utilizando União 72
Anexo 4 : Banco de Dados Venda Sem Partição 73
Anexo 5: Banco de Dados Venda Com Partição 74

1 INTRODUÇÃO

1.1 Motivação

A criação de informações organizadas, bem como o seu armazenamento, torna-se cada vez mais um sinônimo de poder, logo, um gerador de dinheiro para as organizações que as detêm. Por isso, há cada vez mais investimentos que melhorem a forma de como esses dados serão armazenados, processados e, principalmente, recuperados.

Frente a esse cenário, há necessidade de sistemas gerenciadores de banco de dados (SGBD), porém, também se faz necessário que estes sistemas provejam acesso ao banco de dados (BD) de forma eficiente em tempo hábil, não importando se este BD possua pequenos ou grandes volumes de dados. Esta agilidade se faz necessária, pois, as organizações dependem do funcionamento dinâmico de seus serviços e, em muitos casos, esta é vital para negociar de forma compensatória com seus fornecedores ou clientes, além de contribuir para que decisões sejam tomadas de forma eficiente. Mas, nem sempre, esses serviços funcionam de forma correta, podendo gerar vários problemas como lentidão entre uma consulta e sua resposta, o que afeta o sistema de informação (SI) da organização.

Para melhorar o desempenho de acesso a dados existe a sintonia em banco de dados que, conforme CASTOLDI (2005), são técnicas executadas pelo administrador de banco de dados (DBA) com o intuito de que aplicações sejam executadas de forma mais rápida, reduzindo o tempo de resposta das consultas e transações, com isso, melhorando o desempenho geral. Porém, para garantir essa melhora, são necessárias várias técnicas e mudanças em muitos aspectos, dentre estes a troca de hardware, configuração de softwares, alterações na modelagem dos dados ou configuração de parâmetros. Ainda assim, é necessário lidar com o crescimento desenfreado de muitas bases de dados, por isso, muitas técnicas e funcionalidades são implementadas para melhorar o custo de consultas em estruturas gigantescas, dentre estas, surge o assunto deste trabalho: particionamento de tabelas.

1.2 Objetivo

O trabalho proposto neste documento tem como objetivo principal apresentar o particionamento de tabelas mediante pesquisa bibliográfica realizada e, verificar se esta funcionalidade pode trazer benefício no desempenho de consultas realizadas. Mas, para isso, é necessária uma abordagem ampla dos conceitos inerentes à realização de sintonia em banco de dados, visto que está área é bastante extensa e o particionamento representa um ponto frente a tantas outras abordagens existentes na sintonia.

1.3 Metodologia

Para apresentar o particionamento foi escolhida uma abordagem teórica e prática. Primeiramente, a abordagem teórica se faz necessária para dar embasamento à abordagem prática. O SGBD PostgreSQL foi utilizado tanto para desenvolvimento teórico deste trabalho como para obtenção dos resultados práticos finais.

Para validar os conceitos abordados neste trabalho de graduação, fez-se necessário a aplicação das técnicas de particionamento na forma de experimentos sob uma base de dados. Os resultados obtidos desta simulação são analisados com intuito de extrair dados para ilustrar e validar os princípios propostos na elaboração deste documento. Também foi possível realizar um comparativo entre uma base de dados sem particionamento versos esta mesma base de dados particionada e, assim, mostrar ganho ou perda de desempenho.

1.4 Organização

No capítulo 1, é abordado o assunto estudado, traçando os objetivos e o cenário da área.

O capítulo 2 introduz a tarefa de sintonia em banco de dados. É utilizada uma abordagem ampla dos principais pontos da sintonia com o objetivo de contextualizar a área para, posteriormente, focar no tema central deste trabalho.

O capítulo 3 aborda um tipo específico de sintonia em banco de dados, o particionamento. Para isto, é realizado um levantamento teórico do tema, bem como objetivos e recursos. Para exemplificar, é utilizada a base de dados exemplo a qual está disponível na seção “Anexos” deste documento.

O capítulo 4 aborda de forma prática o particionamento, para isso, foi utilizado o sistema gerenciador de banco de dados PostgreSQL.

O capítulo 5 destina-se ao projeto e análise dos experimentos propostos.

E, por fim, o capítulo 6 destina-se a conclusão deste trabalho.

2. SINTONIA EM BANCO DE DADOS

Neste capítulo, para melhor compreensão da tarefa de particionamento, são discutidos os métodos que envolvem o trabalho de sintonia em banco de dados (BD), visto que particionamento é um desses processos envolvidos na sintonia de um BD.

Segundo IKEMATU (2003), sintonia é o "(...) ajuste de algo para que funcione melhor". Em um contexto mais amplo, sintonizar uma estação de rádio com o intuito de obter um som sem interferências ou ruído é um tipo de sintonia, pois é realizado um ajuste no aparelho, afinando-o com uma onda de rádio e sintonizado este ao comprimento desejado dessa mesma onda. A sintonia, ou ajuste, pode ser realizado em um software, em um jogo para computador, hardware ou em um sistema gerenciador de banco de dados (SGBD), dentre tantas outras possibilidades.

Segundo (SOUZA 2009, p.2), sintonia "diz respeito ao ajuste do Sistema Gerenciador de Banco de Dados (SGBD) para melhor utilização dos recursos, provendo um uso eficaz e eficiente do SGBD".

O processo de sintonia em um BD pode envolver mudanças em muitos aspectos: troca de hardware, configuração dos softwares ou alterações na modelagem dos dados. Ainda segundo (SOUZA 2009, p.2), as ações de sintonia podem ser divididas em três grandes grupos:

1 Refinamento do esquema das relações e as consultas/atualizações feitas no BD;
2 Configuração do sistema operacional em uso e;
3 Configuração dos parâmetros dos SGBDS.

Conforme o mesmo autor, 20% das ações de sintonia são destinadas ao refinamento das consultas, 35% são destinadas à configuração do SGBD e 45% a configuração do sistema operacional (SO). Esta divisão é demostrada na Figura 1.

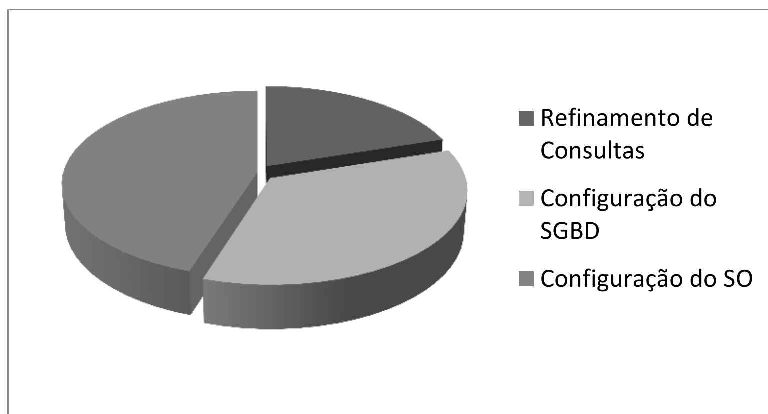


FIGURA 1 - Fases da Sintonia

Antes de aplicar as fases de sintonia, é necessário verificar os possíveis problemas causados por hardware ou software, pois a aplicação pode não possuir um bom fluxograma ou o hardware pode não ser adequado para o uso proposto. Segundo PINCINI (2011):

> Cerca de 60% dos problemas são relacionados ao mau uso de expressões *Structured Query Language* (SQL), 20% relacionados à má modelagem do BD, 10% dos problemas são relacionados à má configuração do SGBD e 10% dos problemas são relacionados à má configuração do SO.

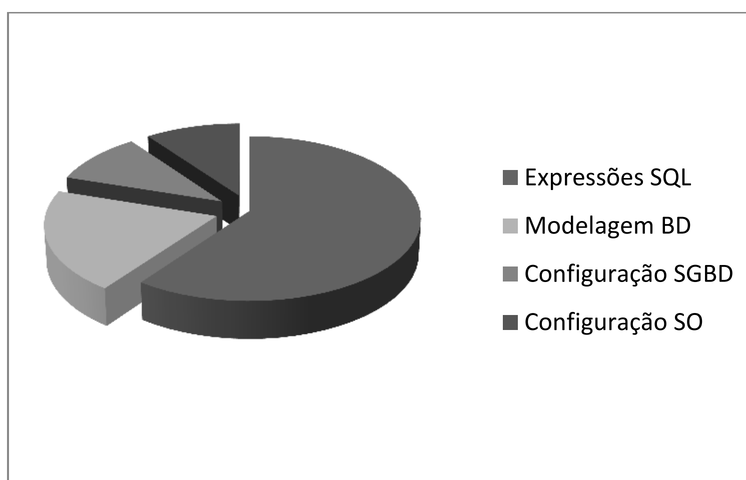


FIGURA 2 - Problemas Relacionados

É necessário que haja uma abordagem mais ampla de todas as partes de um sistema, incluíndo o hardware, sistema operacional, rede e do próprio SGBD, além de se ter conhecimento da aplicação que utiliza a base de dados. Essa abordagem de verificação é necessária visto que o baixo desempenho pode estar ligado ao banco de dados ou não, havendo a possíbilidade de os outros componentes já citados influenciarem o desempenho do banco de dados.

Não é aconselhavel realizar muitas alterações ao mesmo tempo, sendo recomendado que sejam divididas, realizadas e analisadas aos poucos, visto que se ouverem vários procedimentos ao mesmo tempo, a tarefa de análise não será feita de forma correta, além de possíveis erros que surjam durante o processo sejam descobertos e corrigidos de forma mais eficaz.

Antes de alterar o BD, é necessário verificar problemas de software e hardware, pois, muitas vezes, este não é adequado ou a aplicação não possui um bom fluxograma.

É preciso ter enfoque global, ou seja, entender todas as partes do sistema, tais como: hardware, sistema operacional, banco de dados, linguagem de consulta e a própria aplicação, pois, a baixa performance pode não estar ligada ao BD e sim ao um dos componentes citados.

A tarefa deve ser dividida em partes, modificando uma parte de cada vez. É necessário fazer uma análise, seguida de uma alteração e, só assim, ver os resultados, um de cada vez. Se vários procedimentos forem realizados ao mesmo tempo, fica difícil verificar qual procedimento causou algo, o que dificulta a solução de algum imprevisto.

Sintonia deve ser planejada e não pode ser realizada apenas por ser, sem um motivo importante. Um parâmetro alterado de forma arbitrária e mal configurado pode onerar o trabalho ou, em casos mais graves, fazer com que o BD pare de funcionar.

Segundo os princípios de Pareto (também conhecido como princípio 80-20), para muitos fenômenos, 80% das consequências advêm de 20% das causas. Voltando esses princípios para a sintonia em banco de dados, 80% da resolução dos problemas são resultantes de 20% de ajustes, por isso, é recomendável não perder tempo tentando resolver todos os problemas e, sim, focar nos principais.

E, por fim, é importante entender o tipo de aplicação que está em uso, já que cada uma possui suas características e, por isso, requer um tipo de atenção distinta.

2.1. Problemas de Hardware

O hardware é a base para o funcionamento dos sistemas, com isso, pode ser um grande problema para o desempenho do SGBD, ou seja, este tem grande impacto em uma aplicação e, se o mesmo for utilizado de forma incorreta ou apresentar uma qualidade ruim, poderá onerar o desempenho do banco de dados. Por isso, é necessário verificar o hardware do servidor de banco de dados (ou da máquina que ele estiver 'rodando') antes de realizar alterações. WILES (2011) cita alguns componentes que devem ter maior atenção.

Segundo WILES (2011), "quanto mais *Random Access Memory* (RAM) se tem, mais memória cache de disco se terá". Este tipo de memória tem grande impacto se considerarmos que o desempenho de entrada/saída (I/O) dela é mais rápido do que o I/O de acesso a disco.

O tipo de *hard drive* (HD) utilizado deve proporcionar acesso rápido aos dados, por isso, WILDES (2011) cita que "discos que utilizem tecnologia *Small Computer System Interface* (SCSI) ou *Serial AT Attachment* (SATA) são os mais recomendados em função da sua alta taxa de transferência".

Além do tipo de disco empregado, pode-se utilizar *Redundant Array of Independent Drives* (RAID) que é uma forma de criar subsistemas de armazenamento composto por vários discos individuais, essa prática pode trazer desempenho e segurança. RAID são dois ou mais discos, por isso, é recomendável que se utilize o maior número possível de discos. Com as versões mais recentes do SGBD PostgreSQL, é possível colocar *tablespaces*, tabelas e índices em diferentes discos com o intuito de auxiliar no desempenho. WILDES (2011) sugere que tabelas que são utilizadas com maior frequência sejam armazenadas em discos com tecnologias SCSI e as com menor uso em discos mais lentos como SATA ou *Advanced Technology Attachment* (IDE).

Se o banco de dados utilizar muitas funções complexas, é necessário o maior número possível de unidade central de processamento (CPU). Segundo AVILA (2006), um sistema multiprocessado é o modo mais barato de conectar vários sistemas computacionais diferentes, assim o processamento de cada transação executada concorrentemente pode ser feita em um processador diferente, havendo uma diminuição na disputa por um processador e, consequentemente, gerando um melhor desempenho. WILDES (2011) aconselha a utilizar mais memória RAM ou melhores discos se não forem necessárias tantas funções complexas.

Levando em consideração o hardware, não há apenas o disco rígido como um possível gargalo, é necessário considerar que todo o conjunto físico que forma o hardware deva estar tecnicamente uniforme entre si. Entende-se como uniforme os componentes tecnologicamente de acordo e balanceados. CASTOLDI (2005, p. 16) exemplifica da seguinte forma: “(...) não adianta possuir um processador que pode trabalhar a uma frequência de 400 Mega Hertz (MHz) (...) se o módulo de memória trabalha a 333 MHz, o mesmo vale para a placa mãe e assim por diante”.

Componentes defasados podem acarretar prejuízos, exemplo disso é fazer uso de equipamentos de hardware desatualizados com softwares modernos que requeiram muitos recursos, com isso, o funcionamento será onerado.

Segundo (AVILA 2006, p. 21), “o equilíbrio na tarefa de sintonia é encontrado quando houver uma distribuição de forma correta dos recursos de CPU, I/O e memória”, o qual é demonstrado na Figura 3.

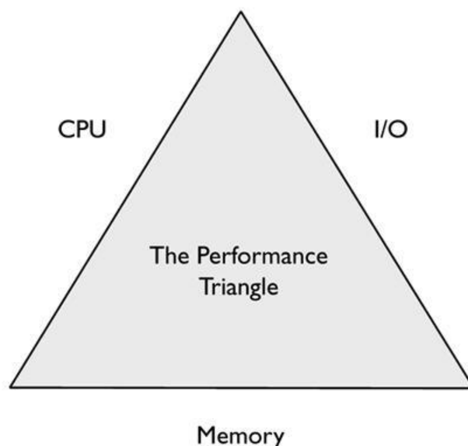


FIGURA 3 – Triângulo do Desempenho

Fonte: Advanced DBA Certification Guide and Reference for DB2 *apud* AVILA 2006, p21.

Componentes como CPU, discos rígidos e memórias devem ser atualizados, pois influenciam no desempenho, entretanto, antes de investir financeiramente nessas atualizações, pode-se optar por realizar ajustes em parâmetros do SGBD, visto que a alocação apropriada de recursos tais como memória, melhoram a performance da cache ou reduzem a paginação.

2.2 Configuração de Parâmetros

Um parâmetro mal configurado pode onerar um banco de dados, por isso, é importante que os mesmos estejam configurados da melhor forma possível. Segundo PICINI (2011), "a chave para afinar as pesquisas é manter os registros e índices da base de dados contíguos".

Um dos parâmetros é o de ajustar os caminhos de acesso, pois a base de dados deve encontar os dados em ordem para retorá-los ao usuário para que ele realize a pesquisa.

O I/O de disco, que é mais lento, tende a prejudicar a performance de várias aplicações de software, por isso, é necessário ajustar I/O e a estrutura física, distribuíndo dados em diferentes discos (se existirem), dessa forma, evitando contenção dos dados.

Cada plataforma exige ajustes específicos então, é possível monitorar e definir esses parâmetros levando em consideração CPU, I/O e memória, visto que pode haver parâmetros de ajustes distintos conforme versões de gerenciadores para um determinado sistema operacional.

Ajustar de forma apropriada os recursos de memória para as estruturas do SGBD pode beneficiar o desempenho. Alocar esses recursos de memória da melhor forma possível e definir

corretamente o *buffer* do banco de dados contribue na redução da paginação, que é o processo de virtualização da memória que faz a subdivisão desta memória física em pequenas partições.

Segundo PANCINI (2011), “depois de ajustes referentes a BD e plataformas operacionais, será necessário realizar ajustes na topologia de acesso as aplicações para obter o desempenho desejado necessário para a camada cliente (aplicações com estrututras em 3 camadas)”.

2.3 Sistema Operacional

Apesar de o hardware ter grande influência sobre a performance do SGBD, “o sistema operacional também pode ser fator determinante, pois o mesmo é quem dá suporte as transações e necessidades de acesso a disco e a memória do SGBD” (CASTOLDI, 2005, p. 17).

É necessário que o DBA verifique o melhor sistema operacional (SO) disponível para a plataforma de hardware que será destinado e para a distribuição do SGBD, além de realizar a configuração dos parâmetros, principalmente os básicos.

Se o SO estiver instalado sem problemas, por exemplo: problemas em bibliotecas e parâmetros referentes ao uso do hardware instalado, o SGBD estará funcionando sem gargalos desnecessários, trabalhando de uma forma mais segura ou com menos instabilidade.

Sistemas operacionais possuem uma instalação ampla, ou seja, quando instalados trazem consigo outros programas, complementos e, em muitas vezes, conteúdos multimídias ou jogos. Esses programas não serão úteis para o sistema e, como todo programa, estes diminuem espaço em memória e utilizam recursos computacionais. Dessa forma, visando melhor o desempenho, é aconselhável que sejam instalados apenas os que serão úteis para o funcionamento do SGBD e suas necessidades. Alguns programas podem ainda abrir lacunas para ações maliciosas sobre o SGBD. É importante salientar que isto deve ser considerando apenas se a máquina em questão for usada com o único proposito de ser o servidor de BD.

2.3.1 Sistema Operacional Linux

Linux é um SO, programa responsável pelo funcionamento do computador, que faz a comunicação entre hardware e software, sendo o termo geralmente usado para designar qualquer SO que utilize o núcleo Linux (BERBERT, 2011). Uma das características do Linux é a de ser um SO aberto, por isso, quando se tem um SO da distribuição Linux como plataforma para um

servidor SGBD, há vantagens e as possibilidades de customizações são bem amplas. Segundo CASTOLDI (2005):

> Esta customização parte desde as opções de arquiteturas de hardware suportadas, passando por diferentes sistemas de arquivos disponíveis, chegando até o ponto de permitir um manejo mais aprofundado de parâmetros de configuração do ambiente que afetam o seu funcionamento direto com o hardware.

O desempenho dos softwares que estão rodando sob o SO pode ser melhorado com parâmetros configurados levando em consideração o tipo de hardware em uso.

2.3.2 Sistema Operacional Windows

Conforme CASTOLDI (2005), o Microsoft Windows difere do Linux por ser um SO comercial (proprietário), além de possuir menor possibilidade de customização das configurações para a realização da sintonia em BD. Dentre as possibilidades, o administrador de banco de dados (DBA) pode realizar alterações para reduzir o uso de memória pelos processos do próprio SO, ou ainda desativar serviços desnecessários ou diminuir os efeitos gráficos.

2.4 Sintonia de Instruções SQL

Segundo definição de (RAMAREZ, 1999), “uma linguagem de consulta estruturada (SQL), é uma linguagem de pesquisa declarativa para banco de dados relacionais”, entende-se como BD relacional o “conceito abstrato que define maneiras de armazenar, manipular e recuperar dados estruturados unicamente na forma de tabelas, construindo um banco de dados” (DATE, 2004). Ainda, segundo AVILA (2006), SQL é uma linguagem para interface com banco de dados relacionais, onde todos os programas e usuários que necessitam realizar qualquer tarefa junto ao BD devem fornecer comandos escritos na linguagem SQL. AVILA (2006) cita que “SQL tornou-se um padrão em banco de dados relacionais”.

Essas instruções fornecem acesso dos programadores, aplicações e usuários ao banco de dados, por isso, SQL pode ser um dos componentes que causa a grande parte dos problemas de desempenho ao sistema, conforme afirma (IKEMATU, 2006). Esta perda de desempenho advém de consultas mal projetadas ou mal construídas que, em um primeiro momento, podem não acarretar problemas, porém, com o aumento da necessidade por consultas e do crescimento da base de dados do sistema, o tempo de consulta dessas instruções pode sofrer alterações que tornem o sistema mais lento. Em outras palavras, consultas mal escritas podem afetar o desempenho dos SGBD pelo simples fato de que o SGBD terá que despender muito mais tempo

escolhendo a forma adequada de executá-la. Por isso, o trabalho de sintonia em instruções SQL, um dos aspectos mais cogitados quando for necessário realizar buscas para encontrar gargalos no SGBD.

Contudo, segundo CASTOLDI (2005), “nem todas as instruções SQL permitem ou possuem opções para realizar a sintonia”, por isso, não é recomendável consumir muito tempo realizando a otimização em grande parte da sintaxe SQL. Ainda, segundo CASTOLDI (2005):

> As condições de consulta SQL são as que permitem maior possibilidade de otimização, porém, é preciso considerar que, para otimizar uma consulta que baseia-se na sintaxe, é necessário desconsiderar fatores como: índices, tamanho da tabela, armazenamento, etc., ou seja, fatores que não são sintáticos.

TABELA 1 - Exemplos de Condições de Pesquisa

... WHERE dataNascimento <> “1990-20-10”
... WHERE idade = 20
...WHERE nome LIKE “MIGUEL” OR nome LIKE “P%”

As condições de consulta são as que permitem maior possibilidade de otimização, mas as que contêm junções e subconsultas são as mais lentas. Essa característica ocorre, pois o processo de leitura em tabelas se torna mais intenso e, com isso, exige um aumento na capacidade do computador em processar dados, conhecido como *throughput*.

2.4.1 Análise da Consulta

Conforme (CASTOLDI 2005, p. 35), nesta primeira fase, “o otimizador examina cada cláusula da consulta e determina se ela pode ser útil na limitação do volume de dados que devem ser percorridos”, ou seja, se a cláusula é útil como argumento de pesquisa ou como parte dos critérios de união.

As cláusulas que são utilizadas como argumento de pesquisa são referidas como otimizáveis e, para recuperação mais rápida, podem fazer uso de um índice. A utilização de uma cláusula otimizável pode fazer uso de índice para realizar a recuperação, de forma mais rápida, de uma informação. Já uma expressão que não é otimizável não pode limitar a pesquisa e, com isso, índices não são úteis para essas expressões.

Em seu trabalho, CASTOLDI (2005) adota uma forma de contagem de pontos com o intuito de ter um referencial para verificar se a sintaxe utilizada em uma consulta irá proporcionar um melhor desempenho, em outras palavras, se a sintaxe que foi adotada em uma

consulta está o mais próxima do ideal. A contagem de pontos pode ser observada na Tabela 2 que contém a pontuação por operador e na Tabela 3 com a pontuação por operando. Em ambas, há uma ordenação da condição de pesquisa da forma: pior para melhor.

TABELA 2 - Pontuação por Operador

Operador	***Pontos***
<>	0
LIKE	3
<=	5
<	5
>=	5
>	5
=	10

TABELA 3 - Pontuação por Operando

Operando	***Pontos***
NULO	0
Tipo de Dados Caractere	0
Tipo de Dados Temporal	1
Outro tipo De Dados Numérico	1
Tipo de Dados Numérico Exato	2
Expressão com Vários Operandos	3
Parâmetro Sozinho	5
Coluna Sozinha	5
Literal Sozinho	10

Utilizando como exemplo a instrução a seguir:

... WHERE smallint_column = 1000

Conforme (SQL Magazine edição Nº 11 *apud* CASTOLDI 2005, p. 36), esta instrução obteria um total de 27 pontos. O resultado foi obtido utilizando o seguinte cálculo:

- ✓ 2 pontos pelo tipo de dados do operando (smallint_column) ser um número exato;
- ✓ 5 pontos para a coluna (smallint_columm) sozinha à esquerda;
- ✓ 10 pontos para o literal (1000) sozinho à direita;
- ✓ 10 pontos pelo operando de igualdade;

Para melhor compreensão, outro exemplo a seguir:

... WHERE char_column >= varchar_column || “x”

Ainda, conforme (SQL Magazine edição Nº 11 *apud* CASTOLDI 2005, p. 36), esta segunda instrução obteria uma contagem de pontos igual a 13 que, obviamente, possui contagem inferior ao exemplo da primeira instrução. Para obter a contagem de pontos de valor 13, foi aplicado o seguinte cálculo:

- ✓ 0 ponto pelo tipo de dados operando caracter (char_column);
- ✓ 0 pontos pelo tipo de dados operando VARCHAR (varchar_column);
- ✓ 3 pontos pela expressão com vários operandos (varchar_comlumn || “x”) à direita;
- ✓ 5 pontos para a coluna (char_column) sozinha à esquerda;
- ✓ 5 pontos pelo operador maior que ou igual a;

A contagem de pontos que foi exemplificada para uma condição de pesquisa varia dependendo do fornecedor. Por isso, deve-se levar mais em consideração a ordem e o conceito desta técnica de otimização. Porém, conforme CASTOLDI (2005):

> SGBDs modernos possuem um otimizador que possui várias outras regras que exigem informações além do escopo da própria instrução SQL, porém, todos estes recorrem a uma contagem de pontos quando não existem outras informações disponíveis.

A forma de pontuação que foi citada realiza o auxilio em questões onde existam duas instruções SQL que possuam sintaxes diferentes e, que produziram de forma previsível e regular, as mesmas saídas, onde estas são chamadas de transformações “uma da outra”.

Para melhor compreensão, conforme CASTOLDI (2005), os exemplos a seguir:

Expressão #1:

... WHERE col1 < col2

```
AND col2 = col3
AND col1 = 5
```

Expressão #2:

```
... WHERE 5 < col2
AND col2 = col3
AND col1 = 5
```

Os SGBDs, em sua maioria, realizam essa transformação de forma automática. Contudo, não tentarão as transformações quando a expressão contiver parênteses e NOTs. Para compreensão, o exemplo a seguir:

```
... WHERE col1 = 5 AND
    NOT (col3 = 7 OR col1 = col2)
```

Esta mesma expressão poderia ser reescrita para que a mesma obtivesse um ganho. Podendo ser escrita da seguinte forma:

```
... WHERE col1 = 5
    AND col3 <> 7
    AND col2 <> 5
```

A utilização de funções nulas, ou seja, função que não contém argumentos, a exemplo CURDATE(), é outro fator que leva a perda de performance em pesquisas devido à construção das condições. Para exemplificar, a seguir:

Expressão #1:

```
... WHERE date_column = CURDATE()
   AND valor * 5 > 100
```

Expressão #2:

```
... WHERE date_column = DATE “2005-05-21”
    AND valor * 5 > 100
```

Nestes exemplos são utilizados valores constantes, o que faz acelerar os acessos, porém, para este caso, se faz necessário uma alteração diária na consulta para que o resultado esperado seja obtido com sucesso.

“O otimizador de um SGBD é preparado para analisar expressões com AND da esquerda para a direita” (CASTOLDI, 2005). Tirando proveito desta característica, pode-se colocar as expressões que forem julgadas como menos prováveis primeiro e, se ambas forem julgadas como igualmente menos prováveis, a expressão que for menos complexa deve ser colocada primeiro. Outra forma de obter proveito desta característica é, se a primeira expressão for uma condição falsa, o SGBD não avaliará a segunda expressão que segue e, desta forma, não perderá tempo realizando essa tarefa.

Conforme regra disponível (SQL Magazine edição Nº 11 *apud* CASTOLDI 2005, p. 36) para as expressões com OR, a regra é oposta ao recomendado para expressões AND, ou seja, a expressão menos provável deve ser colocada à esquerda. Isso ocorre devido ao OR gerar testes adicionais quando a primeira expressão for falsa, ao contrário do que ocorre com AND visto que esta só gerará testes adicionais se a primeira expressão for verdadeira.

Outra regra disponível (SQL Magazine edição Nº 11 *apud* CASTOLDI 2005, p. 36) refere-se ao uso de expressões NOT. Estas expressões devem ser evitadas e, em seu lugar, é aconselhável o uso de expressões menos complexas. Ou ainda, podem-se transformar expressões NOT em condições mais simples, a exemplo:

Expressão #1:

```
... WHERE NOT (cdgrupo > 5)
```

Transformada para:

Expressão #2:

```
... WHERE cdgrupo <= 5
```

Ou ainda, utilizando as consultas abaixo na base de dados Venda que retornarão os itens cuja quantidade em estoque (qteest) está entre 3 e 6:

Expressão #1:

```
SELECT nmitem, qteest FROM item
WHERE qteest NOT BETWEEN 3 AND 6
```

Transformada para:

Expressão #2:

```
SELECT nmitem, qteest FROM item
WHERE qteest < 3 OR qteest > 6
```

Por outro lado, uma condição mais complexa necessita de uma análise mais delicada. Para estes casos, pode-se aplicar o Teorema de De Morgan, que diz o seguinte:

NOT (A AND B) = (NOT A) OR (NOT B)

e

NOT (A OR B) = (NOT A) AND (NOT B)

Desta forma, pode-se transformar a condição de pesquisa:

```
... WHERE NOT (coluna > 5 OR coluna2 = 7)
```

em

```
... WHERE coluna <= 5 AND coluna2 <> 7
```

Em condições que utilizam OR, com o intuito de obter ganho, pode haver uma substituição de consultas que usam OR por condições IN, a exemplo:

Condição #1:

... WHERE cdgrupo = 5 OR cdgrupo = 6

Condição #2:

... WHERE cdgrupo IN (5,6)

Segundo (CASTOLDI 2005), vale uma ressalva em relação ao operador IN: quando este possui uma série formada apenas por números inteiros, é melhor não utilizá-lo. Desta forma pode-se ter uma transformação como segue:

Condição #1:

... WHERE coluna IN (1, 3, 4, 5)

Transformada para:
Condição #2:

... WHERE coluna BETWEEN 1 AND 5 AND coluna <> 2

Conforme CASTOLDI (2005):

> Para condições de pesquisa que façam uso do operador LIKE, deve-se estar atento a que a maioria dos SGBDs tratará de fazer uso de índices. Porém, isto só será tentado se a cláusula começar com um caractere real. Em casos que a cláusula começar com um caractere curinga (% , _ ou *) o otimizador do SGBD evitará o uso de um índice.

Estas demonstrações nas alterações da sintaxe das condições de pesquisa podem ser consideradas pequenas, porém, quando se estuda o aprimoramento de desempenho de consultas SQL, percebe-se que mesmo fazendo alterações nos ajustes de configuração ou incluir hardware mais potente e atualizado, as mudanças com relação ao aplicativo frequentemente surtem maiores efeitos de desempenho junto às consultas. Os sistemas de banco de dados, sejam de

qualquer plataforma, podem ser extraordinariamente velozes e eficientes com aplicativos bem planejados e implementados. Por outro lado, em um sistema onde os aplicativos estejam mal planejados ou implementados de uma forma não eficiente, o banco de dados terá prejuízos e desempenho abaixo do esperado.

2.4.2 Otimizador de Consulta

Existem várias formas de escrever uma mesma consulta, assim como existem muitas formas de executar esta consulta. Da mesma forma que existem consultas mais eficientes que outras, também existem as execuções mais eficientes que outras devido a fatores internos da organização e tamanho dos dados em cada banco de dados. A escolha pela consulta mais adequada é decisão do SGBD e, o componente do banco de dados que determina o modo mais eficiente de executar essa consulta é denominado otimizador de consultas.

Para decidir, o otimizador de consultas baseia-se em estatísticas do banco de dados que são armazenadas nas tabelas de catálogos do sistema, que guarda informações do sistema e informações específicas da tabela. Essas informações devem estar atualizadas, pois se isso não acontecer, o plano de acesso escolhido como o mais eficiente poderá não ser o correto.

Segundo KEIDANN (2009), “o processamento de uma consulta consiste basicamente em interpretar a consulta, otimizá-la e executá-la, sendo que no penúltimo passo é que os planos são gerados e os custos estimados com base nos dados do catálogo de sistemas”.

Ainda segundo KEIDANN (2009), um dos parâmetros que é avaliado em uma consulta, é/são o(s) operador(es) relacional(is) utilizado(s), que trabalham com ideias como indexação, iteração e particionamento. “Em uma tabela particionada, no momento em que os planos de acesso são gerados e os custos estimados, é que as ranges da tabela são levadas em consideração, pois representa caminho de acesso para os dados solicitados em uma consulta”. Na Figura 4, é demonstrado o processamento de uma consulta.

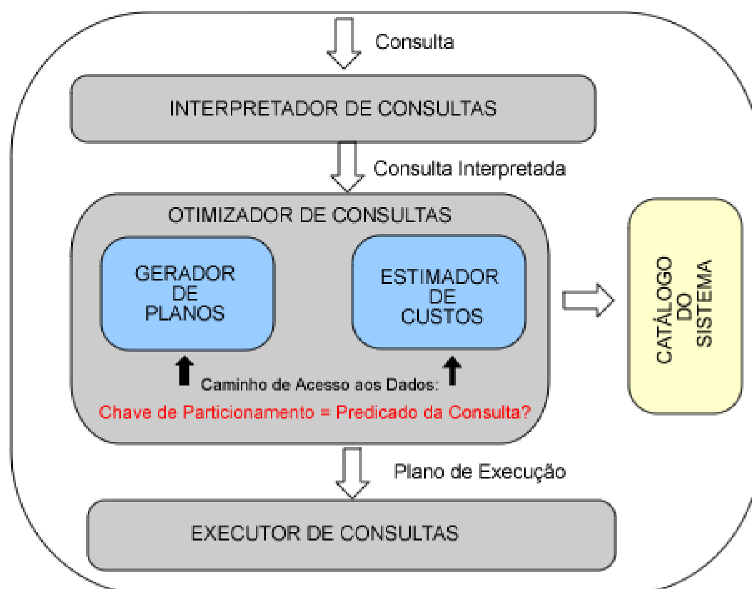


FIGURA 4 - Processamento de uma Consulta

Fonte: KEIDANN (2009)

2.5 Considerações do Capitulo

A sintonia em banco de dados refere-se ao ajuste do SGBD para melhor utilização dos recursos, dessa forma, prover um uso eficaz e eficiente deste SGBD. Esse processo pode envolver mudanças em vários pontos, desde a troca de hardware, configuração de software ou alterações na modelagem dos dados.

Para melhor compreensão do tema, este capítulo se fez necessário para contextualizar a área de sintonia em banco de dados, por isso, o levantamento bibliográfico até este ponto. Nos capítulos que seguem, o tema será focado no particionamento de tabelas, que é um ponto da sintonia.

3 PARTICIONAMENTO DE TABELAS

O Particionamento de Tabelas, segundo (KEIDANN, 2009) "é uma funcionalidade de alguns SGBDs que faz parte da projeção de um banco de dados, mais especificamente do design das tabelas ou objetos grandes de um banco de dados".

Esta técnica melhora o desempenho e torna a manutenção mais simplificada, porém, esses benefícios só serão percebidos em bancos de dados que possuam tabelas com grande número de linhas. Para tabelas menores, esse não trará um benefício, pelo contrário, pode ser prejudicial ao desempenho. Por ter essa característica de grande volume de dados, é bastante mencionada para tabelas de *Data Warehouse*, que é um sistema de computação que possibilita a análise de grande volume de dados, onde é comum existir quantidade de dados próximos a gigabytes (GB) e, em muitos casos, até terabytes (TB). O particionamento normalmente ocorre em tabelas secundárias que possuam grande movimentação e que tenham crescimento em ritmo elevado. Já nas tabelas primárias que, usualmente, são apenas cadastrais, o particionamento é dispensável.

Segundo CHEN (2007), particionar uma tabela consiste em dividi-la ou organizá-la em pedaços menores de acordo com os valores de uma ou mais colunas, a fim de ganhar no acesso e gerenciamento destes "pedaços", mantendo suas características primárias, como a integridade dos dados. Cada um desses "pedaços" ou unidades lógicas de dados devem estar localizados no mesmo banco de dados.

Para exemplificar, é utilizada a base de dados VENDA, a qual está disponível na seção 'Anexos'. Os dados disponíveis são apenas para exemplificar.

TABELA 4 - Tabela Venda Sem Particionamento

numnf [PK] integer	cdcli integer	cdvend smallint	totnf numeric (9,2)	Dtvenda date
2	4	11	3.14	2011-02-02
51	5	5	183.99	2011-05-12
62	4	4	29.44	2011-06-06
122	2	5	31.64	2011-04-11
191	1	15	62.66	2011-03-03
361	1	4	73.44	2011-01-21
441	5	1	123.44	2011-10-14
442	2	4	58.14	2011-06-14
541	4	1	51.44	2011-07-22
642	5	11	23.14	2011-08-25
1092	5	10	221.93	2011-08-17
2941	7	10	12.66	2011-09-30
12941	3	5	28.49	2011-01-04
12942	3	5	1921.72	2011-03-24

Dada à tabela 'Venda' em sua forma original, se a mesma recebesse o particionamento, seria subdividida em 12 partições como é demonstrado nas Tabelas 5 a 14. Nota-se que esta apresenta apenas 10 partições, isso se dá pelo fato de que estas não possuem nenhuma data de venda referente aos meses de novembro e dezembro. Também é importante ressaltar que são apenas ilustrações com o intuito de exemplificar, por isso, possuem poucas tuplas, o que não reflete de forma real o cenário de particionamento que possuem geralmente milhares (ou milhões) de tuplas.

TABELA 5 - Partição Referente à Janeiro da Tabela Venda

numnf [PK] integer	Cdcli Integer	cdvend smallint	totnf numeric (9,2)	Dtvenda Date
12941	3	5	28.49	2011-01-04
361	1	4	73.44	2011-01-21

TABELA 6 - Partição Referente à Fevereiro da Tabela Venda

numnf [PK] integer	Cdcli Integer	cdvend smallint	totnf numeric (9,2)	Dtvenda date
2	4	11	3.14	2011-02-02

TABELA 7 - Partição Referente à Março da Tabela Vend

numnf [PK] integer	cdcli integer	cdvend smallint	totnf numeric (9,2)	Dtvenda date
191	1	15	62.66	2011-03-03
12942	3	5	1921.72	2011-03-24

TABELA 8 - Partição Referente à Abril da Tabela Venda

numnf [PK] integer	cdcli integer	cdvend smallint	totnf numeric (9,2)	Dtvenda date
122	2	5	31.64	2011-04-11

TABELA 9 - Partição Referente à Maio da Tabela Venda

numnf [PK] integer	cdcli integer	cdvend smallint	totnf numeric (9,2)	Dtvenda date
51	5	5	183.99	2011-05-12
1292	9	5	321.01	2011-05-13

TABELA 10 - Partição Referente à Julho da Tabela Venda

numnf [PK] integer	cdcli integer	cdvend smallint	totnf numeric (9,2)	Dtvenda date
62	4	4	29.44	2011-06-06
442	2	4	58.14	2011-06-14

TABELA 11 - Partição Referente à Julho da Tabela Venda

numnf [PK] integer	cdcli integer	cdvend smallint	totnf numeric (9,2)	Dtvenda date
541	4	1	51.44	2011-07-22

TABELA 12 - Partição Referente à Agosto da Tabela Venda

numnf [PK] integer	cdcli integer	cdvend smallint	totnf numeric (9,2)	Dtvenda date
1092	5	10	221.93	2011-08-17
642	5	11	23.14	2011-08-25

TABELA 13 - Partição Referente à Setembro da Tabela Venda

numnf [PK] integer	Cdcli integer	cdvend smallint	totnf numeric (9,2)	Dtvenda Date
371	4	11	82.66	2011-09-16
2941	7	10	12.66	2011-09-30

TABELA 14 - Partição Referente à Outubro da Tabela Venda

numnf [PK] integer	cdcli integer	cdvend smallint	totnf numeric (9,2)	Dtvenda Date
441	5	1	123.44	2011-10-14

Há vários motivos para particionar, mas os principais são os de diminuir o tempo e o custo de uma consulta, visto que operações que poderiam levar vários minutos em tabelas sem particionamento conseguiram ser executadas em poucos segundos se fossem particionadas, a exemplo do carregamento de dados de uma tabela que possua um número considerável de registros. Porém, isso ocorrerá apenas se a forma de particionamento escolhida for adequada ao cenário. Não serão todas as tabelas que obterão ganho, apenas em alguns tipos como em tabelas de *Data Warehouse* (DW), para tabelas com esse perfil, haverá menor tempo de execução.

Em tabelas com bilhões de registros, é necessária uma busca sequencial, o que acarreta em maior custo referente a tempo e recursos computacionais, sendo assim, é mais simples realizar buscas em sub-tabelas. Há uma otimização de I/O, pois ao invés de trabalhar com tabelas gigantescas, a exemplo de uma tabela com 4GB, esta poderia ser transformada em partições de algumas dezenas de megabytes. Ou ainda, uma tabela de 10 GB poderia ser transformada em 10 partições de 1GB.

Outra utilidade é a diminuição do custo de manutenção, um exemplo seria a tarefa de reindex em tabelas grandes. Se for utilizado como exemplo uma tabela com um bilhão de registros para realizar a tarefa de reindex, segundo (GURGEL 2010) levaria bastante tempo, pois, é necessário verificar todos os registros para criar um índice novo e, ler todos os registros de uma tabela muito ampla é necessária uma grande carga de trabalho. Quando é necessária a criação de índices ou reindexes em tabelas menores e com poucos registros, essa tarefa será mais rápida podendo ser realizada em paralelo. Para exemplificar, supondo um cenário onde 6 partições de uma tabela recebem muitos *updates* e há uma necessidade de um reindex, então é possível realizar essa tarefa em paralelo, reindexando todas as partições, podendo realizar essa atividade em frações do tempo que seria necessário em uma tabela sem o particionamento.

A Figura 5 ilustra uma tabela onde seus dados não possuem nenhuma forma de particionamento.

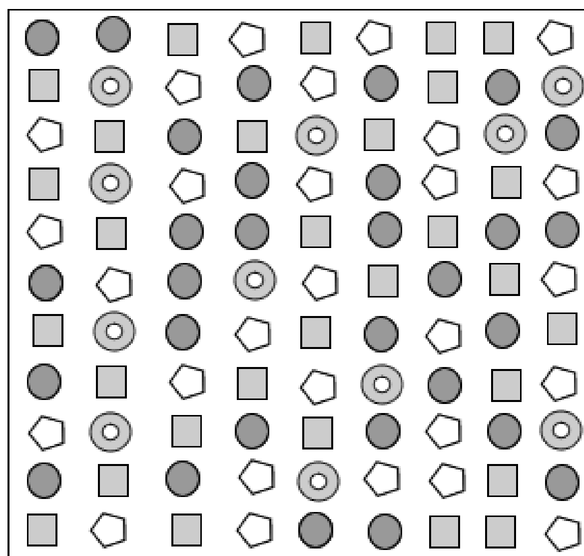


FIGURA 5 - Tabela de dados sem nenhum tipo de particionamento

Fonte: KEIDANN (2009)

Realizando uma consulta na tabela da Figura 5, uma simples busca pode ser demonstrada conforme a Figura 6:

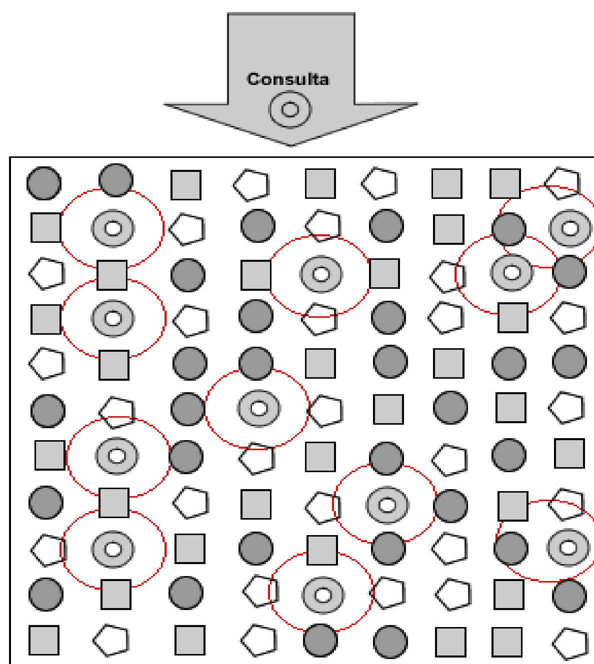


FIGURA 6 - Consulta em tabela sem nenhum tipo de particionamento

Fonte: KEIDANN (2009)

A performance aumenta quando o particionamento é aplicado, pois o benefício do paralelismo de consulta é sentido de forma significativa. Segundo (KEIDANN, 2009), uma

tabela que está particionada e seus dados distribuídos ao longo de 3 partições do BD, tem seu tempo de consulta dividido em 3 (três), dessa forma, o tempo de uma consulta em um ambiente *Data Partition Feature* (DPF), como este caso, é de 1/3 (um terço) do que se comparado a uma consulta em um ambiente sem particionamento, ou seja, sem DPF

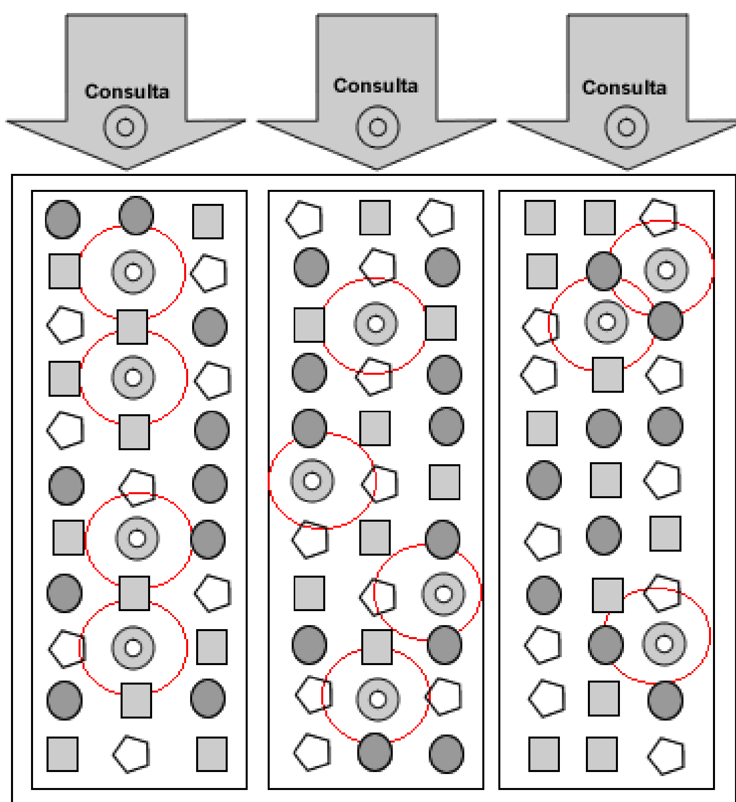


FIGURA 7 - Consulta em tabela distribuída em 3 partições

Outra questão é a possibilidade de trabalhar apenas com uma das partições de uma tabela. Para exemplificar, tendo como cenário o banco de dados VENDA onde há a tabela 'Venda' que armazena a data que ocorreu a venda e, nesta tabela ocorrem grandes atualizações no mês de setembro de 2011 que, para esse exemplo, é considerado o mês vigente, por isso, o grande número de atualizações. Já o mês de agosto não sofre atualizações sendo utilizado apenas para leitura e geração de relatórios, pode-se realizar o reindex e a atualização apenas na partição que representa o mês atual, dispensando a necessidade de trabalhar com todos os registros da tabela original.

Quando uma tabela é particionada e seus dados divididos entre as partições, a tabela inicial (que deu origem as partições) não possuirá nenhuma tupla. Esta tabela servirá apenas como referência para as partições, ficando como *null* e as sub-tabelas da tabela inicial serão as

tabelas físicas. Em SGBDs como PostgreSQL, essa característica é denominada herança, onde há tabelas filhos (sub-tabelas) que herdam as características da tabela pai (inicial). Além disso, os dados que se encontram em partições distintas podem ser acessados sem a preocupação com a localização dos mesmos.

Outra característica interessante do particionamento de tabela é a transparência de dados que possibilita aos usuários permanecerem trabalhando normalmente, sem nem ao mesmo saber que estão trabalhando com uma tabela particionada.

O particionamento de uma tabela é efetuado em sua criação (CREATE TABLE) através da cláusula INHERITS, o que significa dizer que uma vez decidido o particionamento de uma tabela, este deve ser projetado e definido no momento da criação da mesma. Se na carga de dados houver dados que não se enquadrem em nenhum dos intervalos das partições da tabela, estes serão desconsiderados e não serão carregados, por isso é necessária a criação de uma tabela de exceção (KEIDANN, 2009).

3.1 Partição Horizontal

O particionamento horizontal consiste em dividir uma tabela em várias outras. Cada tabela contém o mesmo número de colunas, porém, haverá menos linhas. Para exemplificar, um cenário onde existe uma tabela com 500 milhões de linhas, pode ser decidido particioná-la de forma horizontal em 12 tabelas, com cada tabela gerada representando um mês específico. O particionamento horizontal é o mais utilizado. A exemplo do particionamento apresentado sobre a tabela Vendas.

Decidir como as tabelas serão horizontalmente particionadas depende de uma análise prévia nos dados. Segundo disponível em MICROSOFT (2011), deve-se particionar às tabelas de forma que as consultas façam referência ao menor número possível de tabelas. Caso contrário, consultas com UNION em excesso, usadas para mesclar as tabelas logicamente no momento da consulta, podem afetar o desempenho.

3.2 Partição Vertical

Diferentemente do particionamento horizontal, o vertical faz a divisão de uma tabela em várias outras que contêm um número menor de colunas. Existem dois tipos na forma vertical: a divisão por linhas e a normalização.

O particionamento por linhas faz a divisão da tabela original verticalmente em tabelas com menos colunas. Cada linha lógica em uma tabela dividida coincide com a mesma linha lógica das outras tabelas conforme é identificado por uma coluna UNIQUE KEY idêntica, em todas as tabelas particionadas.

Segundo disponível em MICROSOFT (2011), “o particionamento vertical deve ser considerado com cautela, porque a análise de dados de várias partições requer consultas que unem as tabelas”. Ainda completa: “o particionamento vertical também pode comprometer o desempenho, se as partições forem muito grandes”.

3.3 Quando Particionar?

Toda técnica deve ter planejamento, assim como todo banco de dados. Conforme GURGEL (2011), a documentação de um BD é algo que poucos realizam. Muitos criam o BD e depois de estar em funcionamento, não lembram mais onde a documentação está, ficando sem saber, por exemplo, para que serve uma dada tabela.

Fica evidente que para realizar a tarefa de particionamento é necessário um planejamento. Segundo GURGEL (2010), uma tabela com 50 ou 100 milhões de linhas já pode ser considerada uma excelente candidata ao particionamento. Porém, conforme o manual do PostgreSQL, o ponto exato em que uma tabela irá se beneficiar do particionamento dependerá da aplicação, embora a regra de ouro seja a de que o tamanho da tabela deve exceder a memória física do servidor de BD, o mesmo diz SMITH (2010), “deve-se considerar o particionamento quando uma tabela individual é maior que a quantidade total de memória do servidor, ou quando for atingido 100 milhões de linhas”. Como já foi citado, para particionar é necessário que a tabela candidata seja considerada grande (em relação ao número de linhas), pois, não adiantará aplicar a técnica em uma tabela pequena, visto que poderá acarretar perdas. Quando forem realizadas consultas em uma tabela pequena, que o custo já é considerado baixo, se a mesma for particionada o custo de planejar essa consulta será superior do que se for realizada uma busca na tabela integrada.

3.4 Chave de Partição

Antes de particionar, deve-se considerar se os dados da tabela candidata possuem alguma característica que permita classificá-los, em outras palavras, deve existir um atributo que possibilite um meio para realizar uma separação. Para exemplificar, voltando ao exemplo da tabela ‘Venda’ da base de dados VENDA, ocorre uma classificação através do atributo data, onde o particionamento utiliza a existência de uma progressão de data para realizar uma separação das vendas na forma mensal, em que cada mês do ano é transformado em uma partição específica. Além da progressão de data, podem ser utilizadas informações fixas para realizar a separação, tais como um atributo vendedor de uma tabela venda ou filial de uma tabela lojas, dependendo de como for mais conveniente. Supondo que exista uma empresa com várias filiais onde há uma única tabela com o faturamento mensal de toda a companhia, esta tabela pode ser particionada conforme o número de filiais ou, se for o exemplo do atributo vendedor, em que

necessito gerar relatórios de cada vendedor, então esse atributo pode ser utilizado como termo de separação, acessando só a partição desse funcionário sem a necessidade de acessar as informações dos demais.

Deve ficar claro a necessidade de uma progressão de data ou informações fixas, onde esse campo chave dever ser de fácil verificação, como um campo chave que utiliza o mês de uma data, onde os dados de todo mês podem ser separados. Essas características permitem que o SGBD otimize suas buscas na hora de acessar uma tabela para gerar relatório ou fazer *update*, por exemplo.

A característica de separação deve ser simples como os citados acima, porém existem os não recomendados, que não são igualdades fáceis. Segundo (GURGEL, 2011), um exemplo para esse caso é a separação de nomes por letra inicial, onde é criada uma partição com todos os nomes que começam com a letra “A”, quando for encontrado um nome com essa letra no campo, o mesmo irá para sua partição correspondente. Esse critério não será muito eficiente, possibilitando erros.

Outra particularidade que deve ser considerada, cada registro em uma tabela particionada deve estar associado a uma única partição. A chave de particionamento será o atributo responsável por associar cada registro a sua partição.

Usualmente, a escolha ocorre pelas colunas que contém data, mas existem outros tipos de dados suportados nas colunas de chaves de particionamento. Estas são apresentadas na Tabela 15:

TABELA 15 - Dados Suportados no Particionamento.

Smalint	Integer
Int	Bigint
Float	Real
Double	Decimal
Dec	Numeric
Num	Character
Char	Varchar
Date	Time
Graphic	Vargraphic
Charcter varying	Timestamp
Char varying	Character for bit data
Char for bit data	Varchar for bit data
Character varying for bit data	Char varying for bit data
Tipos definidos pelo usuário	

Fonte: KEIDANN, 2009

Já os dados que não devem ser escolhidos nas colunas chaves de particionamento por não serem suportados são apresentadas na Tabela 16:

TABELA 16 - Dados Não Suportados no Particionamento.

User Defined Types (Structured);	Dbclob
Long Varchar;	Long Vargraphic
Long Varchar For Bit Data	Ref
Blob	Varying Length String for C
Binary Large Object	Varying Length String for Pascal
Clob	*Xml*
Character Large Object	

Fonte: KEIDANN, 2009

3.5 Cláusula de Particionamento

Segundo (KEIDANN 2009), "a cláusula de particionamento é onde se determinam as faixas ou intervalos para as partições". É dada, na clausula de particionamento, a granularidade que pode ser definida como o nível de detalhamento de uma partição, ou sumarização dos dados. Para exemplificar, pode-se citar uma chave de partição formada por um campo data que utiliza o formato (dd/mm/aaaa). Para o exemplo, mesmo possuindo um nível de detalhamento por dia, a granularidade das partições poderia ser semestral ou até anual, conforme for especificado nas cláusulas de particionamento; isso dependerá de como essa partição for planejada, ou seja, de como serão planejados os intervalos de dados. A cláusula de particionamento pode ser considerada a unidade de medida de cada partição.

Ainda segundo (KEIDANN, 2009), este tipo de particionamento atinge melhores resultados quando assume-se que:

- ✓ As queries de consulta coincidam com a lógica do particionamento, fazendo com que os acessos sejam feitos a um mesmo conjunto de dados;
- ✓ Que o tamanho das partições seja uniforme.

O particionamento da tabela deve ser feito durante a criação da mesma, por isso, é importante que o cenário do banco de dados seja conhecido e pensando para projeções futuras (KEIDANN, 2009).

3.6 Partição de Dados

Conforme CHEN (2007), “uma tabela é uma porção de dados, ordenados em colunas e linhas”. Uma partição de dados é uma porção ou um conjunto das linhas de uma tabela, ordenadas dessa forma pela especificação definida na cláusula de partição, baseada por sua vez, na(s) coluna(s) chave de particionamento. Conforme definição, uma partição é um subconjunto de dados de uma tabela, o qual é resultado da divisão conforme uma ou mais colunas que recebem o nome de chave de particionamento.

Uma partição possui a mesma estrutura e tipo da tabela que a originou, visto que recebeu as características através de herança. Porém, a tabela particionada possuí menos linhas que a tabela de origem, o que explica que mesmo estando em tablespaces (subdivisões lógicas) diferentes, tabelas e partição não podem possuir nomes iguais em um mesmo banco de dados, pois ambas são objetos do mesmo tipo (KEIDANN, 2009).

Tablespace é uma subdivisão lógica de um BD com o intuito de agrupar estruturas lógicas relacionadas. Partições diferentes de uma mesma tabela podem ser armazenadas em um mesmo tablespace, em diferentes tablespaces, ou algumas em diferentes tablespaces e outras podem repedir um destes. Quando não há uma especificação, este é criado no primeiro tablespace *default* do BD, no caso do PostgreSQL é o *pg_default*.

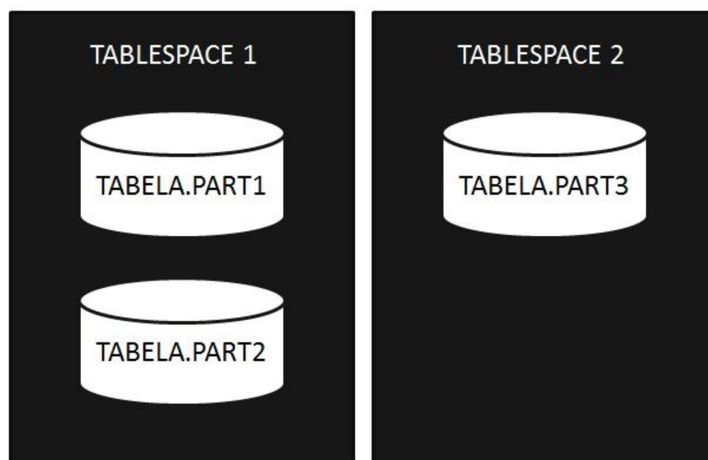


FIGURA 8 - Tabela Particionada em Tablespaces Diferentes

3.7 Escolhendo a Tabela

Para decidir se uma tabela deve ou não ser particionada, (KEIDANN, 2009) cita dois critérios principais: carga de trabalho (*workload*) e velocidade de aumento desta carga.

A carga de trabalho das tabelas particionadas é sempre muito grande, e as consultas sobre as mesmas envolvem, normalmente, períodos de datas e, qualquer operação sobre elas, diminui a performance do sistema.

Conforme KEIDANN (2009) existem dois tipos de tabelas que normalmente enquadram-se nos critérios de particionamento devido às características que apresentam: tabelas de sistemas transacionais e tabelas de sistemas analíticos.

3.7.1 Tabelas de Sistemas Transacionais

Segundo STAIR (1998) “um sistema transacional nada mais é do que o coração da maior parte das organizações empresariais, que dá apoio à monitoração e à realização das negociações de uma organização e gera e armazena dados sobre estas negociações”. Em resumo, pode-se dizer que sistemas transacionais são como o ‘sistema operacional’ de uma organização.

Sistemas transacionais possuem como suas principais características: coleta de dados e armazenamento de informações além da ordenação destes dados, exatamente onde condiz com o primeiro tipo de tabela sujeita ao particionamento; as tabelas de ambientes operacionais dos sistemas transacionais que trabalham com grandes cargas de trabalho, como as tabelas que guardam a movimentação de uma agência bancária, como tabelas de movimento, histórico, empréstimos, conta e até mesmo cliente, pois, em um banco, pode haver milhares de clientes que se cadastram diariamente, abrem contas, realizam empréstimos e outras operações bancário. Ou ainda, uma rede de lojas, que possuí milhares de cliente que realizam compras, possui itens em estoque, realiza vendas e tantos outros dados movimentados durante o dia.

3.7.2 Tabelas de Sistemas Analíticos

Sistemas analíticos também possuem tabelas onde o particionamento é aplicado, mais especificamente em tabelas de *data warehouse* (DW), visto que este tipo de tabela resulta em milhões de linhas. Conforme KEIDANN (2009), tabelas de DW guardam dados históricos e são utilizadas para consultas, além de não sofrerem operações de INSERT, DELETE e UPDATE a todo instante, porém, o desempenho das consultas é fator importante, por essa razão, aplicar o

particionamento torna-se uma técnica importante. É utilizada para gerenciar seus grandes volumes de dados, auxiliando na recuperação destes, monitoramento e indexação. Para estes ambientes, é utilizado o particionamento por intervalo, pois, normalmente, uma tabela é dividida utilizando faixas de tempo (datas) ou números.

O foco desta seção são os sistemas analíticos e, dentre esses sistemas, surgiu à ideia de *data warehouse* que, por sua vez, utilizam frequentemente as técnicas de particionamento. Por essa razão, será dada uma ideia sobre data warehouse e o que é uma tabela de fato.

Segundo DALLANORA (2008), o conceito de DW "pode ser considerado a evolução dos sistemas de apoio à decisões e seu uso é crescente nas organizações a fim de superar um ambiente altamente competitivo através do conhecimento gerado por informações estratégicas".

Um DW proporciona a integração entre os dados de uma empresa, possibilitando a realização das análises gerencias e estratégicas, além de integrar as informações de fontes internas e externas, que nada mais são do que banco de dados.

A integração das informações é feita retirando apenas os dados que serão relevantes para a geração de informações e, a partir desses dados, são criadas as tabelas de fatos e as tabelas de dimensões, nomes estes que indicam suas funcionalidades.

As tabelas de fatos contêm as medições sobre o negócio, a exemplo campos de valores e campos de quantidade, além das chaves para as tabelas de dimensões. DALLANORA (2008) ressalta que "tabelas de fato são extremamente grandes em relação ao número de registro que esta contém, sendo, normalmente, aos milhões".

Já as tabelas de dimensões, segundo definição de (DALLANORA, 2008) é uma forma de modelagem onde as informações se relacionam de forma que possa ser representada como um cubo. Dessa forma, pode-se dividir este cubo e aprofundar-se em cada dimensão ou eixo para extrair mais detalhes sobre os processos internos que ocorrem na empresa, já em um modelo relacional, torna-se muito complicado de serem extraídos e, em muitas vezes, até impossível de serem analisados. O modelo dimensional permite a visualização de dados abstratos de forma simples e relacionar informações de diferentes setores da empresa de forma muito eficaz.

3.8 Quando Aplicar o Particionamento de Tabela

Decidir quando aplicar o particionamento de tabela não é uma tarefa que possa ser considerada simples, pois não existem critérios predefinidos que possam ser tomados como regras. É necessário decidir se o particionamento de tabela será aplicado ou não com base em vários fatores que são definidos para cada cenário em particular, onde o que for aplicável para um não será necessariamente útil para outro cenário distinto.

De maneira geral, pode-se dizer que tabelas grandes normalmente é o foco principal do particionamento a exemplo das tabelas de *data warehouse* que, conforme define (KEIDANN, 2009), é um sistema computacional utilizado para armazenar informações referentes às atividades de uma organização em BD, de forma consolidada. O desenho dessa base de dados fornece os relatórios, a análise de grandes volumes de dados e a obtenção de informações estratégicas que podem facilitar a tomada de decisão. Para exemplificar, de forma teórica, um cenário onde existe um dado site de vendas que deseja que seu cliente, ao acessar a página de venda, veja produtos similares aos que já tenha comprado ou visto. Para que isso ocorra, o site deve traçar o perfil desse cliente com base na trajetória dele pelo site, trajetória essa que deve ter sido armazenada.

Voltando ao ponto da decisão de particionar ou não, o foco, usualmente, são as tabelas grandes, mas segundo (KEIDANN, 2009), existem dois critérios que norteiam essa decisão.

Normalmente, consultas são baseadas em atributos que são fortes candidatos a ser a chave de partição, que determinam faixas de dados que são acessadas com uma frequência maior. Ou seja, a tabelas contém, ou espera-se que contenha, muitos dados que são utilizados de maneiras diferente.

O segundo ponto é o desempenho, se o mesmo não é o esperado e custos de manutenção excedem períodos definidos.

Para exemplificar, utilizando a tabela venda do banco VENDA, tabela que contém um atributo referente à data da venda. No mês vigente, são realizadas várias operações, tais como: INSERT, DELETE e UPDATE, já em meses anteriores, seus dados são apenas para consultas. Por exemplo, se fosse utilizado o particionamento nessa tabela utilizando o mês onde ocorreu essa venda como critério de partição, uma consulta que englobasse dados de apenas um determinado mês, se aplicado à sua respectiva partição, levaria bem menos tempo do que se fosse necessário percorrer toda a tabela inicial que contém todos os dados, não apenas a partição específica referente ao mês.

3.9 Definição do Critério de Particionamento

O critério de particionamento recai totalmente sobre a chave de particionamento, pois é ela quem irá definir as partições de dados (IBM, 2011). Existem, primordialmente, dois fatores que refletem o que se é esperado do particionamento:

1 Será efetuado para agrupar dados por intervalo (ou faixa de valores) tais como: data, região, setor, dentre outros, a fim de realizar operações, alterações e eliminar partições em consultas;

2 Será efetuado para dividir os dados existentes entre as unidades físicas de armazenamento a fim de proporcionar novas soluções aprimoradas de gerenciamento.

Conforme (KEIDANN, 2009), para a projeção de um bom particionamento de tabela e obtenção de benefícios dessa técnica, os seguintes pontos e questionamentos são relevantes:

- ✓ É necessário realizar uma análise na tabela a ser particionada com o objetivo de descobrir se esta é uma tabela grande, a exemplo de uma tabela de DW ou que contenha muitos registros. Porém, KEIDANN (2009) alerta que "(...) tabelas de cadastro dispensam a utilização de práticas de particionamento, pois são tabelas que, se aumentarem a quantidade de seus dados, a proporção não exige a utilização da técnica, podendo reverter os benefícios do particionamento se este for utilizado na tabela".
- ✓ Realizar o particionamento fará alguma diferença?
- ✓ O particionamento será aplicado em qual (quais) coluna(s)? As colunas de datas que definem faixas por intervalos numéricos são as mais utilizadas para o particionamento. Isso ocorre porque essas colunas se enquadram melhor em uma tabela de movimentação que, normalmente, são as que o particionamento é mais utilizado.
- ✓ Quantas partições serão necessárias?
- ✓ Qual a granularidade dessas partições?
- ✓ É importante planejar bem o particionamento de tabelas antes de povoá-la com dados

Além desses pontos, surge a questão dos tablespace que, segundo a documentação do PostgreSQL na seção sobre tablespaces, são subdivisões lógicas que permitem ao DBA do banco de dados definir locais no sistema de arquivos onde serão armazenados arquivos que representam objetos do BD que, uma vez criados, tabelas podem ser referenciadas pelo nome ao criar objetos do BD. Em outras palavras, tablespace é uma subdivisão lógica de um BD utilizado para unir estruturas lógicas. Estas apenas especificam a localização de armazenamento do BD e são armazenadas em arquivos de dados *(datafiles)*, que alocam o espaço especifico na sua criação. Com isso:

- ✓ O particionamento será feito em um mesmo tablespace?
- ✓ Será necessário guardar alguma(s) partição(ões) em mais de um tablespace separados?
- ✓ Definir claramente os tablespaces e seu tamanho.
- ✓ Índices devem ser colocados em tablespaces separados dos dados para melhor performance;

Estes são alguns critérios, porém, critérios a parte podem surgir dependendo do cenário que for analisado.

3.10 Benefícios do Particionamento de Tabelas

No desenvolvimento deste trabalho já foram citados alguns dos benefícios da técnica de particionamento. Nesta seção há uma descrição mais detalhada sobre os benefícios já citados e outros mais.

Os benefícios do particionamento de tabelas que são mencionados no presente trabalho têm como base o documento desenvolvido por (KEIDANN, 2009) e o trabalho de CHEN (2007).

3.10.1 Gerenciamento Aprimorado

O backup de informação pode ser realizado somente da parte da tabela que teve alteração, já a que não sofreu alteração não necessitará de backup, dessa forma, o processo será agilizado e, por isso, o tempo destinado ao backup pode ser reduzido de forma considerável, diminuindo o impacto a falhas.

3.10.2 Desempenho de Consulta Aprimorado

Da mesma forma que ocorre no gerenciamento aprimorado, é possível, através do particionamento de tabelas, efetuar consultas sobre uma faixa de dados específica, dessa forma, o processo será agilizado. Isso ocorre porque o otimizador de consultas considera apenas as partições de dados que são relevantes para a realização da consulta e, obviamente, será um processo que ocorrerá de forma mais ágil. O otimizador de consultas irá selecionar (ou "escolher") quais dados serão varridos para realizar um SELECT, por exemplo, já os dados que não serão necessários para essa operação, não serão acessados, conforme Figura 9.

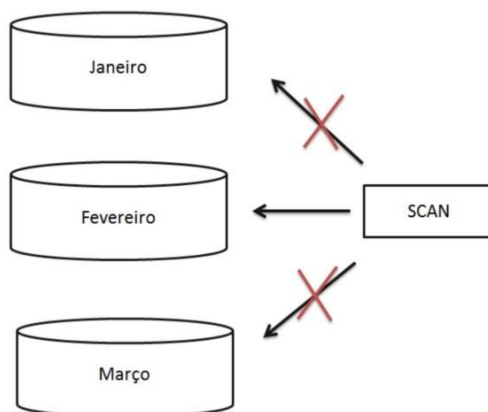


FIGURA 9 - Scan de uma consulta ao mês de fevereiro em uma tabela particionada

Segundo (KEIDANN, 2009), a Álgebra Relacional consegue provar que este pode ser considerado como um dos maiores benefícios do particionamento de tabelas, pois quando uma consulta com critérios de seleção no predicado é solicitada pelo usuário, uma espécie de filtro é aplicado sobre as tuplas da relação e uma porção destas é selecionada, porção esta conhecida como Seletividade do Predicado que, segundo (NAV, 2002), "entende-se como filtro ou critério de seleção de uma consulta".

Ainda seguindo (KEIDANN, 2009), usando uma tabela de 40.000 linhas como exemplo, se esta não for particionada, o otimizador necessitaria escanear todas as linhas para obter o resultado de uma consulta, consulta que poderia exigir apenas a busca em um intervalo de linhas pequeno, isto se estivesse particionada. Caso esta fosse dividida em no mínimo 4 partições de 10.000 linhas, possibilitaria que três partições somando 30.000 linhas não necessitariam de varredura, significando 75% de redução de custos computacionais estimados pelo SGBD. Este ganho de desempenho é demonstrado na Figura 10.

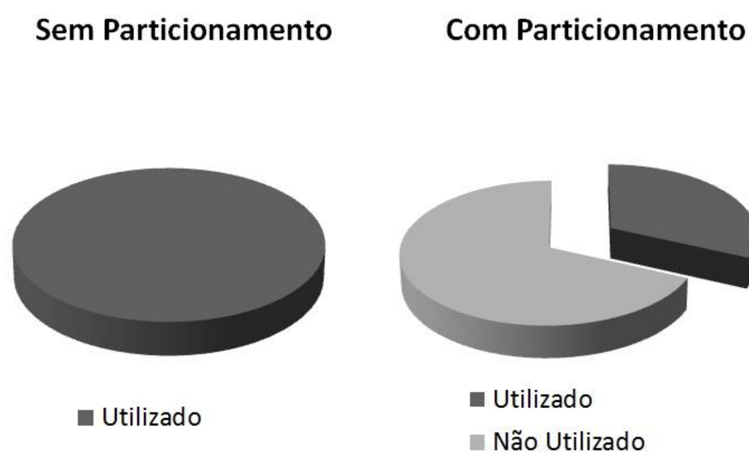


FIGURA 10 - Comparação de Custo Computacional Utilizando em Tabelas

3.10.3 Otimização dos Custos de Armazenamento

Como já citado no Capítulo 2, o hardware tem grande impacto na otimização de um banco de dados e, se for possível à troca do hardware defasado e mais lento por componentes mais rápidos e mais modernos proporcionará grandes benefícios, porém, isso nem sempre será possível principalmente por acarretar custos.

O particionamento dispõe da possibilidade de armazenar dados menos acessados em partições que se encontram localizados no hardware mais lento e mais barato. Esta possibilidade ajudaria a minimizar os problemas de desempenho do exemplo citado anteriormente. E, se for criado a título de exemplo um cenário onde uma tabela tem suas partições divididas entre meses e apenas o mês vigente sofre mais acessos, apenas esta partição poderia encontrar-se armazenada no hardware mais rápido e mais caro, já as partições com menos acessos poderiam estar em hardware mais lento, com isso haveria menor perda de performance.

3.10.4 Capacidade Ampliada de Tabelas

Em todos os SGBDs uma tabela possui capacidade limitada de armazenamento de dados, segundo KEIDANN (2009), o SGBD DB2 possuí tamanho de página de 4 kilobyte (KB), já o SGBD PostgreSQL, segundo seu manual, o tamanho de uma página de dados é de 8192 bytes ou 8 KB.

Porém, quando uma tabela é particionada e seus dados colocados em diversos objetos de armazenamento, como tablespaces, cada uma das tabelas que forem criadas com o particionamento possuirá capacidade de armazenamento igual da tabela não particionada, gerando banco de dados de tamanho virtual ilimitado (KEIDANN, 2009).

3.10.5 Flexibilidade na Alocação dos Índices

Conforme RAMALHO (2005), "índices são estruturas opcionais associadas às tabelas, as quais aumentam o desempenho da recuperação de dados (...)". Por exemplo, o índice de um livro auxilia a encontrar as informações em um menor tempo, índice em um banco de dados fornece um caminho mais rápido para os dados de uma tabela.

Nas tabelas particionadas, os índices são armazenados em seu próprio objeto de particionamento, ou seja, no tablespace separado das partições, e não no tablespace da tabela principal, o que proporciona uma aceleração das operações que envolvam índices, tal como, reorganização de índices.

3.10.6 Contenção do Impacto a Falhas

O backup e a recuperação em tabelas particionadas podem ser realizados de forma independente em cada partição, o que faz acelerar o processo de backup. Dessa forma, há uma redução do risco a falhas, visto que cópias de segurança podem ser realizadas com uma

frequência maior em virtude da demanda de tempo menor. Por exemplo, se ocorrer uma falha de disco, será possível realizar a recuperação dos dados recentes, com períodos de atualizações menores.

3.11 Desvantagens do Particionamento de Tabelas

Toda técnica tem suas desvantagens e, obviamente, o particionamento também apresenta. Essas desvantagens surgem quando este não for utilizado de forma correta e em um ambiente adequado. Conforme (CHEN, 2007), as razões principais são:

- ✓ Dificulta a administração, pois uma tabela pode possuir partições em vários discos de um sistema (computador) ou em vários sistemas;
- ✓ Se não possuir um bom planejamento, a eficácias das queries (consultas que possibilitam acesso a base de dados) pode estar comprometida, visto que se o predicado das consultas não envolver o(s) campos(s) chave(s) de particionamento, o desempenho das mesmas pode piorar ou permanecer igual ao das tabelas não particionadas.

4 PARTICIONAMENTO EM POSTGRESQL

Conforme seu manual, o PostgreSQL é um sistema de gerenciamento de dados objeto relacional (SGBDOR) que, conforme documentação deste SGBD, um banco de dados objeto-relacional (ORD), ou sistema de gerenciamento de banco de dados objeto-relacional (SGBDOR) "é um sistema de gerenciamento de banco de dados relacional que permite aos desenvolvedores integrar ao BD seus próprios tipos de dados e métodos personalizados".

Este SGBD é baseado no POSTGRES Versão 4.2 desenvolvido pelo Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley, sendo o POSTGRES pioneiro em vários conceitos que somente se tornaram disponíveis mais tarde em alguns SGBDs comerciais.

PostgreSQL é desenvolvido como projeto de software livre sob a licença de código aberto BSD (*Berkeley Software Distribution*), este pode ser utilizado, copiado, modificado, estudado e distribuído por qualquer pessoa para qualquer finalidade, seja particular, comercial ou acadêmica, sem restrições e livre de encargos.

Conforme sua documentação, o PostgreSQL é um descendente de código fonte aberto do código original do POSTGRE, que suporta grande parte do padrão SQL, oferecendo muitas funcionalidades modernas, tais como:

- ✓ Comandos Complexos;
- ✓ Chaves Estrangeiras;
- ✓ Gatilhos (*triggers*);
- ✓ Visões (*views*);
- ✓ Integridade Transacional;
- ✓ Controle de Simultaneidade Multiversão.

Além desses fatores, o PostgreSQL pode ser ampliado pelo usuário de várias maneiras distintas, por exemplo, adicionando novos:

- ✓ Tipos de Dados;
- ✓ Funções;
- ✓ Operadores;
- ✓ Funções de Agregação;
- ✓ Métodos de Índice;
- ✓ Linguagens Procedurais.

4.1 Particionamento

Conforme documentação do PostgreSQL, esse SGBD suporta particionamento via herança de tabelas. Cada partição deve ser criada como uma tabela filho de uma tabela pai. A tabela pai em si é normalmente vazia, existindo apenas para representar o conjunto de dados inteiros.

As seguintes formas de particionamento podem ser implementadas em PostgreSQL: particionamento por intervalo e particionamento por lista.

4.1.1 Particionamento por Intervalo

A tabela é dividida em faixas, definida por apenas uma coluna chave ou conjunto de colunas, sem sobreposição entre as faixas de valores atribuídos a diferentes partições. Por exemplo, partição por intervalos de datas, ou por faixas de objetos identificadores referentes a particularidades do negócio.

4.1.2 Particionamento por Lista

A tabela é dividida a partir da declaração explicitada dos valores chave que aparecem em cada partição. Essa forma de particionamento organiza os dados na forma de lista, podendo ser utilizada uma listagem de cidades: São Paulo, Porto Alegre, Brasília e Belo Horizonte. Ou, utilizando uma base de dados onde uma tabela “Grupo” possui os atributos “Cdgrupo” para código do grupo e “Nmgrupo” para nome do grupo, esse segundo atributo é carregado com os seguintes valores: Eletrônicos, Livraria, Informática e Artigos Esportivos. Para esse exemplo, poderiam ser criadas partições que armazenem dados e os organizem para cada um desses grupos.

4.2 Implementando o Particionamento

Estão disponíveis na documentação do PostgreSQL os passos necessários para configurar uma tabela particionada:

1. Inicialmente, é necessário criar o “pai” da tabela, a qual todas as partições herdarão. Esta tabela principal não contém dados.
2. O segundo passo é a da criação de tabelas “filhos”, que herdam da tabela pai.
3. Adicionar restrições à tabela, para que as tabelas particionadas para definir os

valores chaves permitidas em cada partição.

4. Para cada partição, criar um índice na coluna chave, bem como quaisquer índices que sejam necessários. Esse índice de chave não é estritamente necessário, mas na maioria dos casos é útil.
5. Opcionalmente, definir uma *trigger*[1] ou regra para redirecionar os dados inseridos na tabela mestre para a partição específica.
6. Garantir que o parâmetro de configuração *constraint_exclusion* esteja habilitado no *postgresql.conf*. Sem isso, as consultas não serão otimizadas como desejado.

```
CREATE TABLE   VENDA (
       NumNF    INTEGER NOT NULL,
       DtVenda  DATE    NOT NULL,
       CdCli    Integer not null,
       CdVend  smallint not null,
       TotNF    decimal(9,2),
PRIMARY KEY(NumNF),
FOREIGN KEY(CdCli)
        references CLIENTE(CdCLI)
       on update restrict
       on delete restrict,
FOREIGN KEY(CdVend)
         references VENDEDOR(CdVEND)
       on update restrict
       on delete restrict);
```

FIGURA 11- Criação da Tabela VENDA

Criando as tabelas particionadas para inserir os dados, onde devem herdar as características da tabela mestre. É utilizado o comando *()INHERIT*. Este é exemplificado na figura 12.

```
CREATE TABLE venda_jan (

 CHECK (logdate >= DATE '2011-01-01' AND logdate <
DATE '2011-02-01')

 ) INHERITS (venda);
```

FIGURA 12 - Inherits Referente a Janeiro

[1] Recurso de programação executado sempre que o evento associado ocorrer.

A partição referente a janeiro é criada e o comando *inherit* é responsável por esta herdar as características da tabela “pai”, dentro desta tabela é inserido as clausulas de checagem, ou seja, para garantir que cada partição tenha sua faixa de datas corretas.

Posteriormente, é necessário criar a regra. No exemplo, o campo de referência é a data de venda, tendo como base o mês. Também é criada a regra para cada um dos meses, onde cada um dos comandos *insert* realizados na tabela mestre, os dados serão filtrados e inseridos na tabela filho correspondente ao mês. Para isso, é criada uma *trigger*, conforme Figura 13.

```
CREATE OR REPLACE FUNCTION venda_insert_trigger()
 RETURNS TRIGGER AS $$
 BEGIN
     IF (NEW.logdate >= DATE  '2011-01-01' AND
          NEW.logdate < DATE '2011-02-01 ') THEN
         INSERT INTO venda_jan VALUES (NEW. *);


     ELSIF (NEW.logdate >= DATE  '2011-02-01' AND
            NEW.logdate < DATE  '2011-03-01') THEN
         INSERT INTO venda_fev VALUES (NEW. *);
```

FIGURA 13 - Trigger Referente a Janeiro e Fevereiro

5. PROJETO E ANÁLISE DO EXPERIMENTO

Este capítulo destina-se a apresentar os experimentos realizados com o objetivo de validar os conceitos abordados na elaboração do presente trabalho.

5.1 Modelagem do Banco de Dados

Os experimentos para o estudo de caso proposto foram realizados com o banco de dados ilustrado na Figura 24. Este armazena informações de venda e possui quatro tabelas. Este modelo foi criado para a elaboração deste trabalho, o que proporciona um melhor controle sob a base.

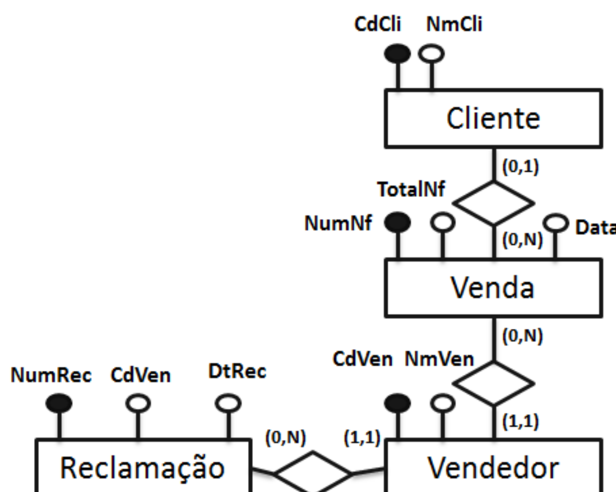


FIGURA 14 - Modelo Conceitual Banco de Dados Venda

A transformação do modelo lógico para o esquema relacional é apresentada nas Tabelas 17 a 20:

TABELA 17- Modelo Relacional: CLIENTE

Tabela Cliente	
Nome Atributo:	Armazena:
CdCli	Código Cliente
NmCli	Nome Cliente

A tabela CLIENTE ilustrada na tabela 17 possui dois atributos:

- CdCli: Coluna que armazena informações referentes ao código dos clientes. Esta é a chave primária.
- NmCli: Coluna que armazena informações referentes ao nome dos clientes;

Esta tabela tem como objetivo armazenar informações a respeito dos clientes, por isso, possui o atributo nome e um código de identificação para cada cliente.

TABELA 18 - Modelo Relacional: VENDA

Tabela Venda	
Nome Atributo:	Armazena:
NumNf	Número Nota Fiscal
TotalNf	Total Nota Fiscal
Data	Data

A tabela VENDA ilustrada na tabela 18 possui três atributos:

- NumNf: Coluna que armazena informações referentes ao número da nota fiscal. Esta é a chave primária;
- TotalNf: Coluna que representa o valor total da venda emitida em cada nota fiscal;

A tabela Venda armazena os dados históricos das transações realizadas. Dessa forma, possui atributo que armazena o número único de nota fiscal, bem como o total desta nota, além da data que ocorreu a venda.

TABELA 19 - Modelo Relacional: VENDEDOR

Tabela Vendedor	
Nome Atributo:	Armazena:
CdVen	Código Vendedor
NmVen	Nome Vendedor

A tabela VENDEDOR ilustrada na tabela 19 possui dois atributos:

- CdVen: Coluna que armazena informações referentes ao código do vendedor Esta é a chave primária;
- NmCli: Coluna que armazena informações referentes ao nome do cliente.

A tabela Vendedor armazena informações em relação ao vendedor, estes possuem um nome e um código único que os identifica.

TABELA 20 - Modelo Relacional: RECLAMAÇÃO

Tabela Reclamação	
NumRec	Número da Reclamação
CdVen	Código Vendedor
DtRec	Data Reclamação

A tabela RECLAMAÇÃO ilustrada na tabela 20 possui três atributos:

- NumRec: Coluna que armazena informações referentes ao número de identificação da reclamação. Esta é a chave primária;
- CdVen: Coluna que armazena informações referentes ao código do vendedor;
- DtRec: Coluna que armazena a data da reclamação.

A tabela Reclamação armazena as reclamações realizadas, esta possui um número que identifica cada reclamação, bem como a data que esta ocorreu, além do código do vendedor que foi alvo desta reclamação.

5.1.1 Estratégia de Particionamento

Para o modelo utilizado, foram experimentadas duas abordagens, uma utilizando partições e outra não. As partições são geradas a partir de campos que armazenam informações de data, sendo que existe uma partição para cada mês do ano, o que totaliza 12 partições. A Figura 15 mostra parte do script usado para criá-las.

```
CREATE TABLE venda_fev (

    CHECK ( logdate >=  DATE '2011-02-01' AND logdate <
DATE '2011-03-01')

 ) INHERITS (venda);
```

FIGURA 15 - Partição na Tabela Venda

O exemplo da figura mostra a criação de uma partição na tabela “Venda”, usando como base o campo “logdate”, e critério de particionamento datas cujo mês é fevereiro. As partições foram criadas sobre as tabelas “Venda” e “Reclamação”, conforme a necessidade e objetivo de cada um dos experimentos. O *script* completo de criação das partições está disponível nos anexos. Observa-se que as demais tabelas não sofreram alterações, assim como todas as tabelas da abordagem sem particionamento.

5.2 Configurações

Esta seção descreve as configurações utilizadas para os testes de desempenho das abordagens com e sem partição. Os tópicos apresentados demonstram como foi realizada a carga de dados nas tabelas de teste e como foram conduzidos os testes. Além disso, são descritas as plataformas de hardware e software utilizadas para realização dos experimentos.

5.2.1 Carga de Dados

Após a criação do modelo, estas tabelas foram populadas conforme a necessidade de cada teste. Para cada teste, se fez necessário uma grande carga de dados, por isso, foi utilizada uma ferramenta para auxiliar na tarefa de geração de dados de testes. Para isso, algumas destas foram avaliadas conforme Tabela 21.

TABELA 21 - Ferramentas de Inserção de Dados

Ferramenta	SGBD	Licença	Limitação	Pago
Databene	Todos	Livre	Não há versão Windows	Não se Aplica
Datanamic	Todos	Proprietário	Gera Apenas 100 Tuplas	Sim
EMS Data Generator	Todos	Proprietário	Gera Apenas 100 Tuplas	Sim
Generator Data	MySQL	Livre	Apenas MySQL	Não se Aplica

Das ferramentas analisadas, a Databene é uma opção de código livre, porém de difícil instalação. Já a Datanamic é proprietária com interface intuitiva e fácil manipulação, além de abranger vários SGBDs, porém, possuí limite de geração de no máximo 100 tuplas em sua versão de avaliação, assim como a EMS Data Generator. E, por fim, o Generator Data é uma opção livre, mas disponível apenas para o SGBD MySQL.

A ferramenta EMS Data Generator foi a escolhida para gerar os dados, pois, dentre todas as ferramentas, as proprietárias se mostraram mais adequadas para esta tarefa, principalmente pela possibilidade mais ampla de geração, além de uma facilidade maior para manipulação e instalação. E, dentre as proprietárias, esta se mostrou mais adequada a este trabalho e apresentou melhor custo-benefício.

Após a escolha da ferramenta, esta foi utilizada para a geração dos *scripts* em formato CSV e, posteriormente, importado para cada banco de dados. Esta abordagem foi adotada para garantir os mesmos dados. Por exemplo, ao gerar um *script* com 1 milhão de tuplas, todos os experimentos que necessitarem desta quantidade de tuplas serão carregados com os mesmo

dados. Ao todo, foram gerados seis *scripts* de tamanhos distintos, como é demostrado na Tabela 22.

TABELA 22 - Número de Tuplas X Tamanho

Número Tuplas	Tamanho
Mil	234 KB
10 Mil	2.28 MB
100 Mil	22.8 MB
1 Milhão	228 MB
10 Milhões	2.23 GB

Cada experimento utiliza uma carga de dados específica, por isso, a distribuição destes dados será descrita em maiores detalhes na seção especifica de cada experimento.

5.2.2 Condução dos testes

Além da ferramenta utilizada para popular as bases, também foi empregada uma solução automatizada para a realização dos testes. A ferramenta de código-aberto Jmeter, projeto pertencente ao Grupo Apache Jakarta, foi utilizada para esse fim. Esta é uma aplicação desktop desenvolvida em Java para ser utilizada na criação e aplicação de testes de performance e de carga em diferentes serviços de um ambiente servidor, dentre estes: HTTP, FTP e SGBD. Seus resultados facilitam na compreensão sobre como a aplicação reagiu aos testes aplicados.

Para este trabalho, foram necessárias 50 iterações para cada consulta e, partir destes resultados, foi possível encontrar a média de tempo de cada consulta. Após a conexão da ferramenta com o SGBD, os experimentos foram realizados através de requisições, ou seja, cada uma das consultas às bases de dados foi realizada pelo Jmeter através de iterações. Conforme VICENZI (2004, p.54), “(...) na prática, a aplicação de uma metodologia de testes está fortemente condicionada à sua automação”, por isso, esta auxiliou na automação dos testes.

5.2.3 Plataformas utilizadas

O sistema operacional Windows foi a escolha para realização dos testes, mais especificamente Windows 7 versão Home Premium de 64 bits da Microsoft. A escolha do SO se deu pelo fato deste ser um sistema bastante conhecido e difundido tanto em ambiente educacional quanto comercial. Outro fator que influenciou nesta decisão foi o fato deste sistema estar disponível durante a realização destes experimentos.

Dentre os sistemas gerenciadores de banco de dados, conforme (COLARES, 2007) “os três principais bancos de dados livres que podem concorrer de fato com os gigantes do mercado, são: Firebird, MySQL e PostgreSQL.” Dentre estes, o PostgreSQL foi escolhido devido as suas funcionalidades, familiaridade no uso e, até então, não ter encontrado trabalhos sobre o assunto que o utilizem.

E, por fim, em relação ao hardware, os experimentos foram realizados utilizando um computador portátil HP Envy 17 3D com as seguintes configurações: processador Intel Core i7 CPU Q 720 de 1.60GHZ, memória RAM de 6 GB DDR3 1333 com 667 MHz e HD de 500 GB e 7200 RPM.

5.3 Plano de Testes

A proposta dos experimentos é apresentar um comparativo de desempenho entre bases particionadas e não particionadas. Ao todo, foram criadas 30 bases de dados, sendo 15 particionadas e 15 sem o particionamento. Foram realizados três experimentos, estes são descritos e seus resultados relatados nesta seção. Demais informações estão presentes nos anexos.

5.3.1 Experimento um: Consulta com Filtro por Data

Este experimento consiste em realizar uma consulta utilizando filtro por data em uma base particionada e uma não particionada, ambas com mesmo número de tuplas. A partição foi realizada considerando os 12 meses de um único ano.

As Tabelas Vendedor e Cliente são “populadas” com 100 vendedores e 100 clientes, respectivamente. A Tabela Reclamação não recebe nenhuma inserção. Já a Tabela Venda recebeu dados referentes às vendas realizadas nos 12 meses do ano de 2011, com as seguintes cargas distintas:

- Mil Tuplas;
- 10 Mil Tuplas;
- 100 Mil Tuplas;
- 1 Milhão Tuplas;
- 10 Milhões Tuplas;

Cada uma dessas cargas foi inserida em uma tabela particionada e outra sem, o que representa 10 testes. Os comparativos são realizados apenas entre tabelas com a mesma carga.

A consulta por filtro de data refere-se a todas as vendas realizadas no dia 29 de setembro de 2011, conforme a Figura 16.

```
Select *
From venda
Where dtvenda = '2011/09/29';
```

FIGURA 16 - Consulta com Filtro por Data

Esta foi aplicada a bases sem partição e com partição. A partir dos resultados obtidos, foi possível realizar um comparativo deste método entre bases com o mesmo número de tuplas. Os resultados em milissegundos (ms) são demostrados no Gráfico 1.

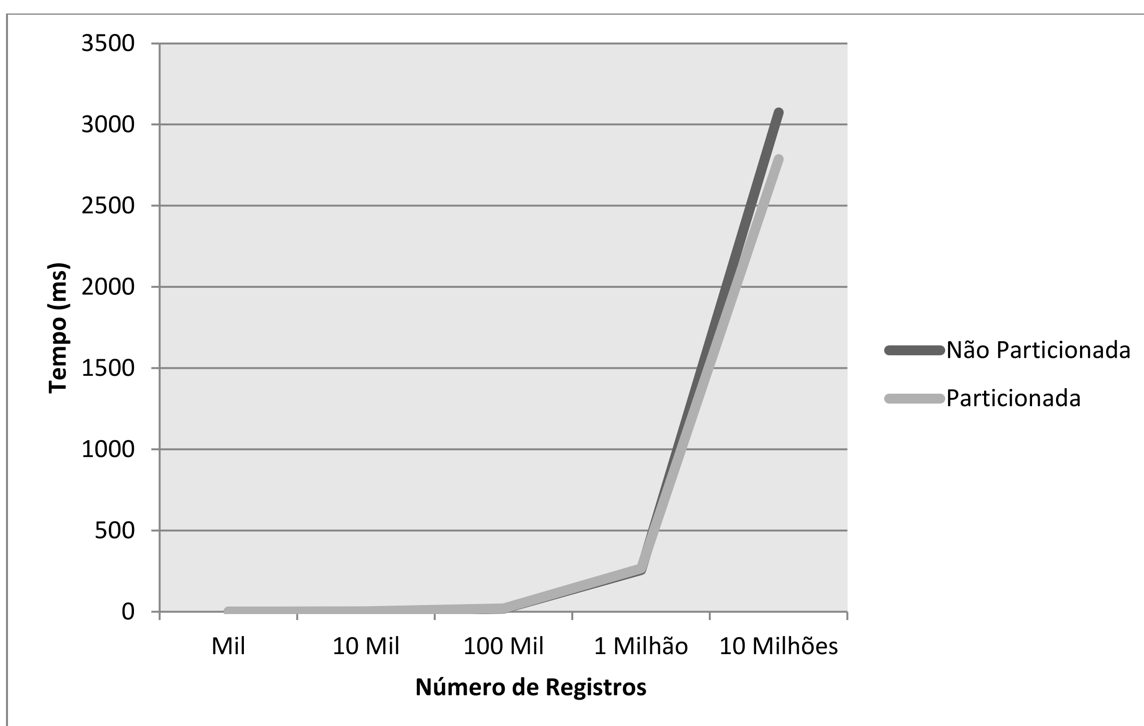


GRÁFICO 1 – Consulta com Filtro por Data Com e Sem Particionamento

Em tabelas sem partição, a busca ocorre através da varredura sequencial e total de todos os seus dados, filtrando os resultados e retornando apenas os registros relevantes, neste caso, as vendas ocorridas no dia 29/09/2011. Já em tabelas particionadas, essa busca ocorre de uma forma

mais complexa, esta acessa inicialmente a tabela “pai” que não contêm dados e, posteriormente, acessa a partição pertencente aos dados que foram solicitados, se estes existirem, são retornados. Nesta segunda tabela haverá uma busca sequencial e, por esta conter menos dados se comparada à tabela original, esta busca varrerá menos registros.

Porém, ao mesmo tempo em que esta acessa menos dados, esta também necessita de um plano de consulta mais elaborado, em outras palavras, em muitos casos, o custo de realizar uma busca completa em uma tabela com um número baixo de registros será menor do que realizar o acesso à tabela “pai” e, posteriormente, acessar a tabela “filho”, isto pode ser observado nos resultados do experimento um, em quase todos os cenários o particionamento se mostrou desvantajoso, ou seja, o tempo de acesso foi maior do que em tabelas não particionadas. Apenas nas consultas realizadas nas tabelas com 10 milhões de tuplas este fato se inverteu. A partir desse número de linhas, o método de particionamento pode ser considerado como prática que acarretará ganho de desempenho.

5.3.2 Experimento dois: Consulta com Filtro usando operador de comparação “Maior Igual”

O experimento dois consiste em filtrar registros através do atributo data, porém, este não mais retornará dados referentes a apenas uma data, desta vez retornará todos os dados ocorridos naquela data mais todos os que ocorreram em datas superiores a este filtro. O objetivo deste experimento é realizar um comparativo entre o tempo de execução ao aplicar este filtro em duas bases de dados iguais, porém, uma com partição e outra sem este recurso.

Na configuração deste experimento, as tabelas Vendedor e Cliente são “populadas” com 100 registros de vendedores e 100 registros de clientes, respectivamente. A Tabela Reclamação não recebe nenhum registro. A tabela Venda recebe diferentes cargas de dados, todos distribuídos entre 12 meses do ano de 2011. Para cada carga de dados existem duas tabelas, com e sem partição, esta distribuição ocorreu da seguinte forma:

- Mil Tuplas;
- 10 Mil Tuplas;
- 100 Mil Tuplas;
- 1 Milhão Tuplas;
- 10 Milhões Tuplas;

Para este experimento foram realizados 10 testes, pois para cada uma dessas cinco cargas foi inserida em uma tabela sem partição e outra com.

A consulta por filtro de data refere-se a todas as vendas realizadas a partir do dia 20 de outubro de 2011, ou seja, esta retornará todas as vendas ocorridas entre o período que compreende o dia 20 de outubro ao dia 31 de dezembro, ambos do ano de 2011. A consulta SQL pode ser vista na Figura 17.

```
Select *
From venda
Where dtvenda >= '2011/10/20'
```

FIGURA 17 – Consulta com Operador de Comparação "maior igual"

A consulta da Figura 17 foi aplicada a bases com e sem partição, a partir dos resultados obtidos, foi possível realizar um comparativo deste método entre bases com o mesmo número de tuplas. Os tempos encontrados em milissegundos (ms) são demostrados no Gráfico 2.

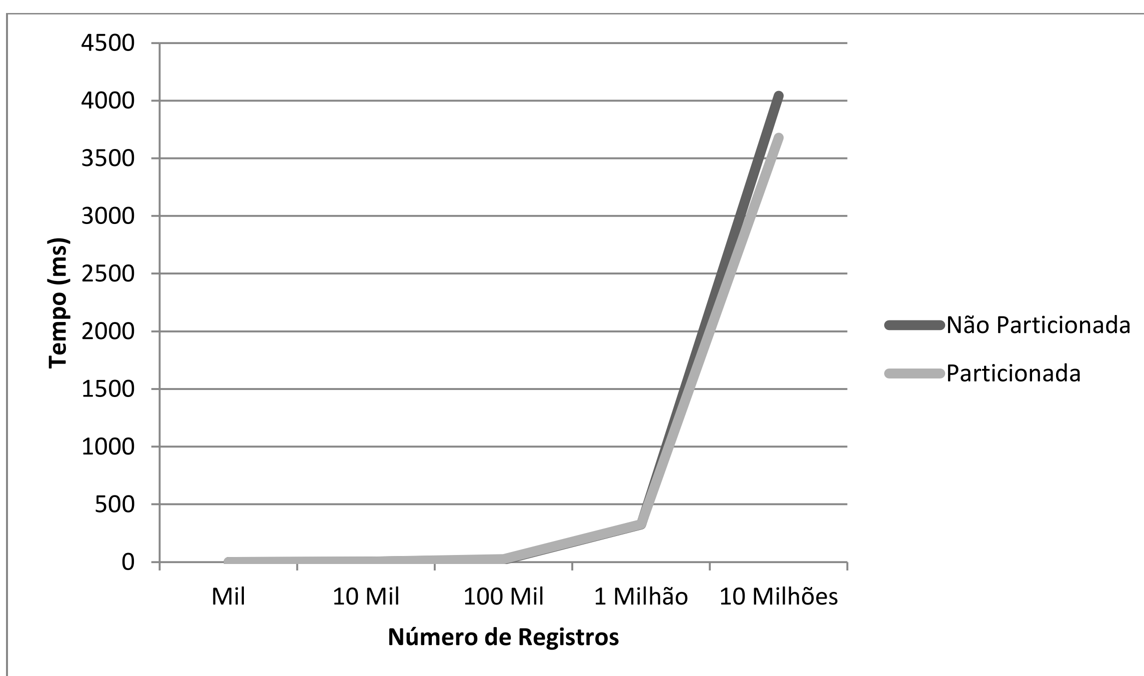


GRÁFICO 2 - Consulta com Filtro por Data Inicial

Na tabela sem partição, a busca ocorre de forma sequencial em toda a tabela principal, onde o filtro de data será responsável pela análise dos dados e retorno das linhas que se enquadrarem nessa classificação.

Já na tabela Venda com partição mensal, a primeira tabela não possui registros, estes estão presentes ao longo das 12 partições. Por isso, as partições “filhas” são herdeiras das tabelas “pais”, quando a tabela Venda é acessada, obrigatoriamente, as tabelas filhos necessárias são requisitadas. Para o experimento dois ao retornar os dados solicitados, será necessário consultar a tabela Venda e três partições, pois são solicitadas datas superiores a 20 de outubro, sendo necessário o acesso da partição de outubro, além de novembro e dezembro.

Assim, como no experimento três, não são todas as tabelas que alcançam ganho, tabelas com menos linhas e particionadas possuem maior custo de consulta, por isso, apenas nos testes com 10 milhões de tuplas, o particionamento começa a ser vantajoso. Isso ocorre, visto que varrer todas as linhas da tabela com poucas tuplas acarreta em menor custo do que planejar o acesso a uma tabela particionada. Como pode ser observado no gráfico, a partir dos 10 milhões de linhas, o particionamento começa a mostrar suas vantagens.

5.3.3 Experimento três: Consulta Utilizando União

Nos experimentos anteriores, as consultas somente acessavam uma única tabela de cada vez, porém, consultas podem acessar várias tabelas ao mesmo tempo, ou ainda, acessar uma mesma tabela de forma que várias linhas desta sejam processadas ao mesmo tempo. Consultas que possuem essas características são chamadas de consulta com união.

Nas configurações do experimento de junções, as tabelas Vendedor e Clientes receberam 100 registros, ou seja, cada uma contém 100 registros de 100 vendedores ou 100 clientes, respectivamente. A tabela Venda, que nos experimentos anteriores recebia carga variada e em metade dos experimentos era particionada, neste novo experimento tem suas características alteradas. Esta tabela recebeu carga fixa de um milhão de registros e em todos os experimentos será particionada. Em compensação, a tabela Reclamação que não recebia carga, neste experimento recebe cinco cargas variadas, conforme indicado abaixo:

- Mil Tuplas;
- 10 Mil Tuplas;
- 100 Mil Tuplas;
- 1 Milhão Tuplas;
- 10 Milhões Tuplas;

O objetivo deste experimento é analisar o comportamento e o tempo de uma consulta com união entre duas tabelas, sendo a tabela Venda particionada e com um milhão de registros, e a tabela Reclamação com cinco cargas, sendo cinco particionadas e cinco não particionadas. A elaboração deste experimento permitirá analisar os resultados e, assim, traçar a melhor forma de particionar tabelas levando em consideração união e grande carga de dados.

A consulta utilizada neste experimento está disponível na Figura 18. Esta consulta realiza o cálculo, através do operador de agregação *count*, de quantas vendas o vendedor de código 33 realizou no período do mês de outubro de 2011. Este valor é unido ao número de reclamações que o vendedor de código 33 recebeu no mês de outubro de 2011.

```
Select count (*) as CdVend
From Venda
Where dtvenda between '2011/10/01' and '2011/10/31'
and Venda.CdVend = 33

union

Select count (*) as CdVend
From Reclamacao
Where DtRecl between '2011/10/01' and '2011/10/31'
and Reclamacao.CdVend = 33
```

FIGURA 18 – Consulta com União

A consulta deste experimento difere das realizadas nos experimentos anteriores, pois esta realiza operações de maior complexidade, mesmo em tabela particionada.

Em ambos os casos, há a necessidade de utilização de um algoritmo de ordenação, conforme indicados nos planos de acesso gerados pelo PostgreSQL. Nos testes, o PostgreSQL utilizou o método QuickSort que, conforme (GOODRICH e TAMASSIA, 2007), baseia-se no método de divisão e conquista, onde os registros são divididos para depois serem rearranjados. Posteriormente, a consulta realiza uma busca sequencial em toda tabela principal se esta não for particionada ou na tabela principal mais na partição correspondente se esta for particionada. Os resultados são demonstrados no Gráfico 3.

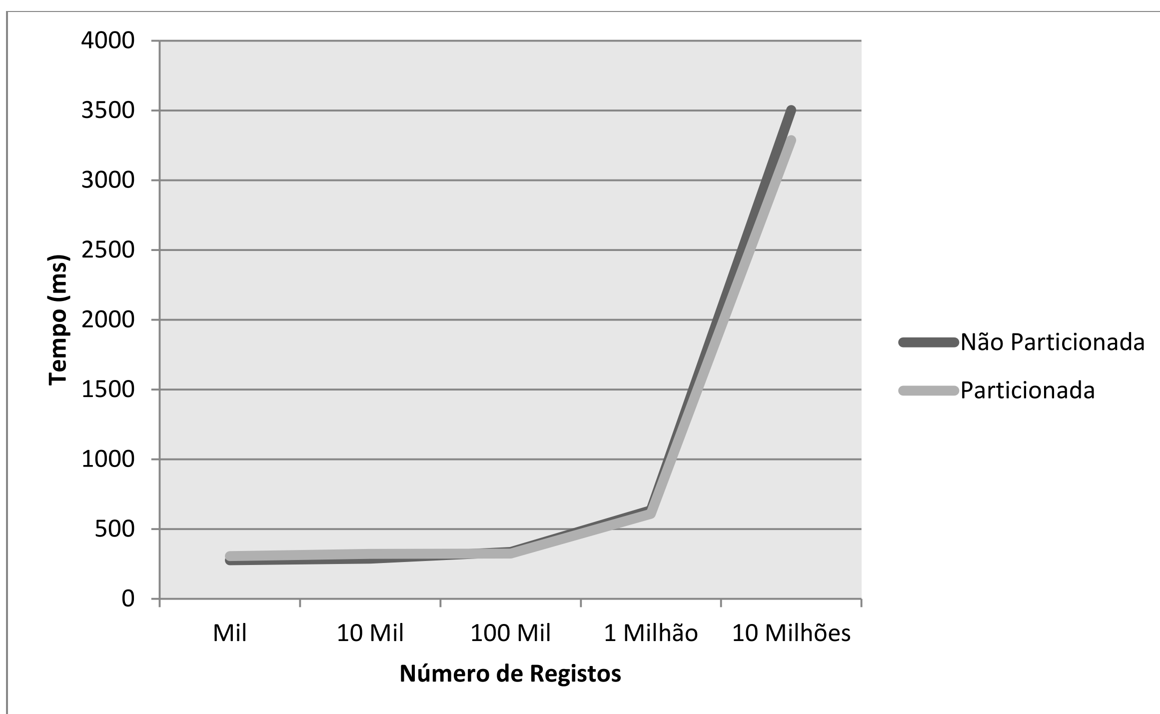


GRÁFICO 3- Consulta Utilizando União

Esta consulta necessita de maior planejamento por parte do SGBD, por isso, mesmo em bases de dados com poucas tuplas, o tempo de execução é alto (os resultados podem ser visualizados nos anexos). Até 100 mil linhas seu custo sofreu pouca alteração. A partir desta carga de dados, o tempo de execução começa a espaçar. Também, a partir dos 10 milhões de tuplas da tabela Reclamação, as tabelas particionadas começam a apresentar seus benefícios. Neste experimento, além da consulta com grau de complexidade maior, a união de duas tabelas com grande carga de dados possibilitou demostrar os benefícios do particionamento.

6 CONSIDERAÇÕES FINAIS

Ao longo desse trabalho, foi possível compreender o quanto é importante para as organizações o trabalho de sintonia realizado pelos DBAs. Também foi possível entender que esta engloba várias práticas, desde a mudança de hardware, passando por otimização do sistema operacional, alterações de parâmetros, melhoria de consultas SQL, até o particionamento de tabelas, tema deste trabalho.

Assim como quase todas as tarefas de sintonia, o particionamento não possui regras rígidas e fixas, apenas recomendações que devem ser consideradas e seguidas como boas práticas. Como exemplo disto esta a pergunta de quando se deve particionar. A única resposta para esta pergunta será: depende! Muito se fala em regra de ouro: de tabelas com 50 milhões de linhas, tabelas que excedem a memória física do servidor de banco de dados ou, simplesmente, que dependerá da aplicação.

Através da criação dos cenários e do uso de ferramentas para popular e testar as cargas, foi possível realizar testes. Nos dois primeiros testes, na grande maioria dos casos não foi possível mostrar os benefícios do método, apenas o com maior carga de dados alcançou seu objetivo. Devido à limitação de hardware, os testes chegaram apenas a 10 milhões de registros. Com o aumento desta carga, os benefícios certamente serão mais visíveis, tanto no experimento 1 quanto no experimento 2. Já no experimento 3, foi possível unir através de uma consulta mais complexa duas tabelas com carga de dados elevada, por isso, os benefícios foram mais visíveis.

Para os experimentos, não foi possível trabalhar com um número tão expressivo de registros, que são próprios de aplicações gigantescas apoiadas por servidores de ponta. Mesmo assim, conseguiu-se trabalhar com uma massa de dados considerável, grande o suficiente para que se pudesse verificar o ganho em desempenho quando a estratégia de particionamento é utilizada.

Por fim, os resultados encontrados nos experimentos foram satisfatórios se for levado em consideração que estes foram realizados em um computador de uso pessoal. Em cenários reais, o particionamento será aplicado com bases de dados compostas por centenas de milhões ou ainda bilhões de dados gerenciados por servidores com hardware potente. Por isso, mesmo com limitações, este estudo cumpriu com seu propósito.

Como contribuição para trabalhos futuros, fica a proposta de testes em ambientes com hardware mais robusto. Outra possibilidade de trabalho contempla o estudo de diferentes fórmulas de particionamento, utilizando critérios de divisão de datas mais flexíveis, e até mesmo considerando informações baseadas em outros tipos de dados.

REFERÊNCIAS

BERBERT, Fábio de Paula, **O Surgimento do Linux**. Disponível em: <http://www.vivaolinux.com.br/artigo/O-Surgimento-do-Linux> Acesso em: SET 2011.

CASTOLDI, Leandro López, **Sintonia em Banco de Dados Sob Diferentes Sistemas de Arquivos.** Ijuí – RS, 2005. 154p.

CHEN, W.; FISHER, A.; LALLA A.; MCLAUCHLAN A. D.; AGNEW D. **Database Partitioning, Table Partitioning, and MDC for DB2 9**. IBM Corp., 2007.

COLARES, Flávio. **Análise Comparativa de Banco de Dados Gratuitos,** Monografia apresentada à coordenação do curso de Ciências da Computação da Faculdade Lourenço Filho, como requisito para obtenção do grau de Bacharel em Ciência da Computação, 2007.

DATE, C. J, **Introdução a Sistemas de Banco de Dados.** 8 ed.: Campus, 2004.

DE AVILA, André. **Monitoramento e Sintonia em Banco de Dados Relacionais.** Ijuí – RS, 2006. 67p.

DALLANORA, Júlio; PRAUCHNER, Renilton; DLL, Sérgio L.; PADOIN, Edson L. **Data Warehouse para Procedimentos Ambulatoriais do SUS.** Universidade Regional do Noroeste do Estado do RS, Ijuí, 2008

GOODRICH, Michael T.; TAMASSIA, R.; **Estruturas de Dados e Algoritmos em Java.** 4 ed.: Bookman. São Paulo, 2007.

IBM Corp. **Centro de Informações do Banco de Dados IBM DB2 para Linux, UNIX e Windows**. Partitioned Table. Disponível em: <http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0021562.htm>. Acesso em: JUL 2011.

IKEMATU, Ricardo Shoiti. **Realizando Tuning na Base de Aplicações.** Disponível em: <http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=1592>. Acesso em: AGO 2011.

[...]. Documentação PostgreSQL 8.0.0. Disponível em: < http://pgdocptbr.sourceforge.net/pg80/ >. Acesso em: AGO 2011.

KEIDANN, Glaucia Luciana. **Particionamento de Grandes Volumes de Dados.** Ijuí – RS, 2009. 111p.

MICROSOFT. **Particionamento.** Disponível em: <http://msdn.microsoft.com/pt-br/library/ms178148.aspx >. Acesso em: ABR. 2011

MOREIRA, Eduardo. **Modelo Dimensional para Data Warehouse.** Disponível em: http://imasters.com.br/artigo/3836/modelo_dimensional_para_data_warehouse>. Acesso em: AGO 2011.

PINCINI, Rodrigo. **Desempenho de Banco de Dados: A Técnica de Otimização com Tuning**. Disponívem em: < http://www.devmedia.com.br/post-11323-Metodologia-Tuning.html >. Acesso em: AGO 2011.

RAMALHO, José Antonio. **Oracle 10g.** São Paulo - SP, Pioneira Thomson Learning , 2005. 385p.

RAMAREZ, A. NAVATHE, B., **Fundamentals of Database Systems**, Assison- Wesley, 1999

SOUZA, Ana Paula dos Santos; Campos, Bruno Fidelis, Dias, Carla Glênia Guedes; Alves, Michel Batista; Vieira, Carlos Eduardo Costa, Carelli, Flávio Campos. e Luiz Fabiano Costa de Sá. **Tuning em Banco de Dados**. Cadernos UniFOA. Volta Redonda, ano IV, n. 10, agosto. 2009. Disponível em: <http://www.unifoa.edu.br/portal_pesq/caderno/edicao/10/19.pdf> Acesso em AGO 2011.

STAIR, Ralph M. **Princípios de Sistemas de Informação. Uma abordagem gerencial.** 2ed, Rio de Janeiro: LTC, 1998.

VICENZI, Auri Marcelo Rizzo**. Orientação a Objeto: Definição, Implementação e Análise de Recursos de Teste e Validação.** 2004. Tese (Mestrado) Universidade de São Paulo, São Carlos, 2004. Disponível em: <http://www.teses.usp.br/teses/disponiveis/5/55134/tde17082004122037/publico/tese.pdf>. Acesso em: AGO 2011.

WILES, Frank. **Performance Tuning PostgreSQL**. Disponível em: <http://www.revsys.com/writings/postgresql-performance>. Acesso em: AGO 2011.

ANEXOS

Anexo 1: Resultados da Consulta com Filtro por Data

TABELA 23 - Comparativo de Consulta a Mil Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	0.164	0.207
Média	0.260	0.269
Máxima	0.389	0.394

TABELA 24 - Comparativo de Consulta a 10 Mil Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	1.413	1.926
Média	1.734	2.090
Máxima	2.157	2.196

TABELA 25 - Comparativo de Consulta a 100 Mil Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	14.359	17.649
Média	17.663	19.672
Máxima	20.356	21.487

TABELA 26 - Comparativo de Consulta a 1 Milhão Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	214.487	250.589
Média	256.272	266.972
Máxima	284.141	310.564

TABELA 27 - Comparativo de Consulta a 10 Milhões Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	2908.843	2578.843
Média	3073.874	2787.531
Máxima	3833.864	3545.486

Anexo 2: Resultado consulta com Filtro usando operador de comparação "Maior Igual"

TABELA 28 - Comparativo de Consulta a Mil Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	0.256	0.270
Média	0.268	0.337
Máxima	0.295	0.441

TABELA 29 - Comparativo de Consulta a 10 Mil Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	1.793	2.217
Média	2.229	2.570
Máxima	2.287	3.011

TABELA 30 - Comparativo de Consulta a 100 Mil Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	16.579	22.986
Média	19.953	26.400
Máxima	21.901	28.097

TABELA 31 - Comparativo de Consulta a 1 Milhão Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	263.383	270.041
Média	325.901	327.630
Máxima	332.283	342.188

TABELA 32 - Comparativo de Consulta a 10 Milhões Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	3180.073	2752.124
Média	3384.753	2974.458
Máxima	4042.145	3678.072

Anexo 3: Resultados da Consulta Utilizando União

TABELA 33 - Comparativo de Consulta a Mil Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	236.828	246.654
Média	274.542	304.202
Máxima	291.385	366.650

TABELA 34 - Comparativo de Consulta a 10 Mil Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	240.770	287.705
Média	286.239	322.353
Máxima	322.866	381.336

TABELA 35 - Comparativo de Consulta a 100 Mil Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	324.650	261.428
Média	334.466	322.553
Máxima	344.844	329.700

TABELA 36 - Comparativo de Consulta a 1 Milhão Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	545.371	534.778
Média	634.326	609.16
Máxima	665.767	624.520

TABELA 37 - Comparativo de Consulta a 10 Milhões Tuplas

	Não Particionada	Particionada
Amostras	50	50
Mínimo	3006.731	2759.006
Média	3502.538	3286.427
Máxima	4245.689	3471.858

Anexo 4 : Banco de Dados Venda Sem Partição

```
------------------------------------------------------------------------------------------------------------------------

CREATE TABLE CLIENTE (
        Cdcli    integer NOT NULL,
        NmCli  CHAR(30),
PRIMARY KEY(CdCli));

------------------------------------------------------------------------------------------------------------------------

CREATE TABLE   VENDEDOR (
        CdVend      smallint NOT NULL,
        NmVend       CHAR(30),
PRIMARY KEY(CdVend));

------------------------------------------------------------------------------------------------------------------------

CREATE TABLE VENDA(
        NumNF                    INTEGER NOT NULL,
        DtVenda                  DATE NOT NULL,
        CdCli            Integer not null,
        CdVend                   smallint not null,
        TotNf            decimal(9,2),
PRIMARY KEY(NumNF),
FOREIGN KEY(CdCli)
     references CLIENTE(CdCLI)
```

```
        on update restrict
        on delete restrict,
FOREIGN KEY(CdVend)
      references VENDEDOR(CdVEND)
        on update restrict
        on delete restrict);
```

```
CREATE TABLE RECLAMACAO (
        NumRec          INTEGER NOT NULL,
        CdVend     smallint NOT NULL,
        DtRecl     DATE NOT NULL,
        Descr      VARCHAR(30),
PRIMARY KEY (NumRec),
FOREIGN KEY (CdVend)
        references VENDEDOR(CdVEND)
        on update restrict
        on delete restrict);
```

Anexo 5: Banco de Dados Venda Com Partição

```
CREATE TABLE CLIENTE (
        Cdcli   integer NOT NULL,
        NmCli  CHAR(30),
PRIMARY KEY(CdCli));
```

```
CREATE TABLE   VENDEDOR (
```

```
        CdVend      smallint NOT NULL,
        NmVend      CHAR(30),
PRIMARY KEY(CdVend));
--------------------------------------------------------------------------------------------------------------
CREATE TABLE VENDA(
        NumNF                   INTEGER NOT NULL,
        LogDate                 DATE NOT NULL,
        CdCli           Integer not null,
        CdVend                  smallint not null,
        TotNf           decimal(9,2),
PRIMARY KEY(NumNF),
FOREIGN KEY(CdCli)
     references CLIENTE(CdCLI)
        on update restrict
        on delete restrict,
FOREIGN KEY(CdVend)
     references VENDEDOR(CdVEND)
        on update restrict
        on delete restrict);
--------------------------------------------------------------------------------------------------------------
 CREATE TABLE venda_jan (
    CHECK (logdate >= DATE '2011-01-01' AND logdate < DATE '2011-02-01')
 ) INHERITS (venda);
--------------------------------------------------------------------------------------------------------------
 CREATE TABLE venda_fev (
     CHECK ( logdate >=  DATE '2011-02-01' AND logdate < DATE '2011-03-01')
```

```
) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
CREATE TABLE venda_mar (
    CHECK (logdate >= DATE '2011-03-01' AND logdate < DATE '2011-04-01')
) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
CREATE TABLE venda_abr (
    CHECK (logdate >= DATE '2011-04-01' AND logdate < DATE '2011-05-01')
) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
  CREATE TABLE venda_mai (
    CHECK (logdate >= DATE '2011-05-01' AND logdate < DATE '2011-06-01')
) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
CREATE TABLE venda_jun (
    CHECK (logdate >= DATE '2011-06-01' AND logdate < DATE '2011-07-01')
) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
CREATE TABLE venda_jul (
    CHECK (logdate >= DATE '2011-07-01' AND logdate < DATE '2011-08-01')
) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
CREATE TABLE venda_ago (
    CHECK (logdate >= DATE '2011-08-01' AND logdate < DATE '2011-09-01')
) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
```

```
CREATE TABLE venda_set (
    CHECK (logdate >= DATE '2011-09-01' AND logdate < DATE '2011-10-01')
 ) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
CREATE TABLE venda_out (
    CHECK (logdate >= DATE '2011-10-01' AND logdate < DATE '2011-11-01')
 ) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
CREATE TABLE venda_nov (
    CHECK (logdate >= DATE '2011-11-01' AND logdate < DATE '2011-12-01')
 ) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
CREATE TABLE venda_dez (
    CHECK (logdate >= DATE '2011-12-01' AND logdate < DATE '2012-01-01')
 ) INHERITS (venda);
-------------------------------------------------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION venda_insert_trigger()
 RETURNS TRIGGER AS $$
 BEGIN
    IF (NEW.logdate >= DATE  '2011-01-01' AND
        NEW.logdate < DATE '2011-02-01 ') THEN
       INSERT INTO venda_jan VALUES (NEW. *);

    ELSIF (NEW.logdate >= DATE  '2011-02-01' AND
          NEW.logdate < DATE  '2011-03-01') THEN
       INSERT INTO venda_fev VALUES (NEW. *);
```

```
ELSIF (NEW.logdate >= DATE  '2011-03-01' AND
        NEW.logdate < DATE '2011-04-01') THEN
    INSERT INTO venda_mar VALUES (NEW. *);

ELSIF (NEW.logdate >= DATE  '2011-04-01 ' AND
        NEW.logdate < DATE '2011-05-01') THEN
    INSERT INTO venda_abr VALUES (NEW. *);

ELSIF (NEW.logdate >= DATE  '2011-05-01 ' AND
        NEW.logdate < DATE '2011-06-01') THEN
    INSERT INTO venda_mai VALUES (NEW. *);

ELSIF (NEW.logdate >= DATE  '2011-06-01' AND
        NEW.logdate < DATE '2011-07-01 ') THEN
    INSERT INTO venda_jun VALUES (NEW. *);

ELSIF (NEW.logdate >= DATE  '2011-07-01' AND
        NEW.logdate < DATE '2011-08-01') THEN
    INSERT INTO venda_jul VALUES (NEW. *);

ELSIF (NEW.logdate >= DATE  '2011-08-01' AND
        NEW.logdate < DATE '2011-09-01 ') THEN
    INSERT INTO venda_ago VALUES (NEW. *);

ELSIF (NEW.logdate >= DATE  '2011-09-01' AND
```

```
          NEW.logdate < DATE '2011-10-01') THEN
        INSERT INTO venda_set VALUES (NEW. *);

ELSIF (NEW.logdate >= DATE  '2011-10-01' AND
          NEW.logdate < DATE '2011-11-01') THEN
        INSERT INTO venda_out VALUES (NEW. *);

ELSIF (NEW.logdate >= DATE  '2011-11-01' AND
          NEW.logdate < DATE '2011-12-01 ') THEN
        INSERT INTO venda_nov VALUES (NEW. *);

ELSIF (NEW.logdate >= DATE  '2011-12-01 ' AND
          NEW.logdate < DATE '2011-01-01') THEN
        INSERT INTO venda_dez VALUES (NEW. *);

    ELSE
       RAISE EXCEPTION 'Date out of range.  Fix the venda_insert_trigger() function!';
   END IF;
   RETURN NULL;
END;
$$
LANGUAGE plpgsql;
---------------------------------------------------------------------------------------------------------------------
```

```
CREATE TABLE RECLAMACAO (
        NumRec              INTEGER NOT NULL,
        CdVend      smallint NOT NULL,
        DtRecl      DATE NOT NULL,
        Descr       VARCHAR(30),
PRIMARY KEY (NumRec),
FOREIGN KEY (CdVend)
        references VENDEDOR(CdVEND)
        on update restrict
        on delete restrict);




CREATE TABLE rec_jan (
   CHECK (DtRecl >= DATE '2011-01-01' AND DtRecl < DATE '2011-02-01')
) INHERITS (RECLAMACAO);

 CREATE TABLE rec_fev (
    CHECK ( DtRecl >=  DATE '2011-02-01' AND DtRecl < DATE '2011-03-01')
) INHERITS (RECLAMACAO);

 CREATE TABLE rec_mar (
   CHECK (DtRecl >= DATE '2011-03-01' AND DtRecl < DATE '2011-04-01')
) INHERITS (RECLAMACAO);
 CREATE TABLE rec_abr (
   CHECK (DtRecl >= DATE '2011-04-01' AND DtRecl < DATE '2011-05-01')
```

```
) INHERITS (RECLAMACAO);

CREATE TABLE rec_mai (
    CHECK (DtRecl >= DATE '2011-05-01' AND DtRecl < DATE '2011-06-01')
) INHERITS (RECLAMACAO);

CREATE TABLE rec_jun (
    CHECK (DtRecl >= DATE '2011-06-01' AND DtRecl < DATE '2011-07-01')
) INHERITS (RECLAMACAO);

CREATE TABLE rec_jul (
    CHECK (DtRecl >= DATE '2011-07-01' AND DtRecl < DATE '2011-08-01')
) INHERITS (RECLAMACAO);

CREATE TABLE rec_ago (
    CHECK (DtRecl >= DATE '2011-08-01' AND DtRecl < DATE '2011-09-01')
) INHERITS (RECLAMACAO);
CREATE TABLE rec_set (
    CHECK (DtRecl >= DATE '2011-09-01' AND DtRecl < DATE '2011-10-01')
) INHERITS (RECLAMACAO);
CREATE TABLE rec_out (
    CHECK (DtRecl >= DATE '2011-10-01' AND DtRecl < DATE '2011-11-01')
) INHERITS (RECLAMACAO);
CREATE TABLE rec_nov (
    CHECK (DtRecl >= DATE '2011-11-01' AND DtRecl < DATE '2011-12-01')
) INHERITS (RECLAMACAO);
```

```
CREATE TABLE rec_dez (
    CHECK (DtRecl >= DATE '2011-12-01' AND DtRecl < DATE '2012-01-01')
 ) INHERITS (RECLAMACAO);



CREATE OR REPLACE FUNCTION RECLAMACAO_insert_trigger()
 RETURNS TRIGGER AS $$
 BEGIN
    IF (NEW.DtRecl>= DATE  '2011-01-01' AND
        NEW.DtRecl < DATE '2011-02-01 ') THEN
       INSERT INTO rec_jan VALUES (NEW. *);

    ELSIF (NEW.DtRecl >= DATE  '2011-02-01' AND
          NEW.DtRecl < DATE  '2011-03-01') THEN
       INSERT INTO rec_fev VALUES (NEW. *);

 ELSIF (NEW.DtRecl >= DATE  '2011-03-01' AND
          NEW.DtRecl < DATE '2011-04-01') THEN
       INSERT INTO rec_mar VALUES (NEW. *);

 ELSIF (NEW.DtRecl >= DATE  '2011-04-01 ' AND
          NEW.DtRecl < DATE '2011-05-01') THEN
       INSERT INTO rec_abr VALUES (NEW. *);

 ELSIF (NEW.DtRecl >= DATE  '2011-05-01 ' AND
          NEW.DtRecl < DATE '2011-06-01') THEN
```

```
        INSERT INTO rec_mai VALUES (NEW. *);

 ELSIF (NEW.DtRecl >= DATE  '2011-06-01' AND
           NEW.DtRecl < DATE '2011-07-01 ') THEN
        INSERT INTO rec_jun VALUES (NEW. *);

ELSIF (NEW.DtRecl >= DATE  '2011-07-01' AND
           NEW.DtRecl < DATE '2011-08-01') THEN
        INSERT INTO rec_jul VALUES (NEW. *);

ELSIF (NEW.DtRecl >= DATE  '2011-08-01' AND
           NEW.DtRecl < DATE '2011-09-01 ') THEN
        INSERT INTO rec_ago VALUES (NEW. *);

ELSIF (NEW.DtRecl >= DATE  '2011-09-01' AND
           NEW.DtRecl < DATE '2011-10-01') THEN
        INSERT INTO rec_set VALUES (NEW. *);

ELSIF (NEW.DtRecl>= DATE  '2011-10-01' AND
           NEW.DtRecl < DATE '2011-11-01') THEN
        INSERT INTO rec_out VALUES (NEW. *);

ELSIF (NEW.DtRecl >= DATE  '2011-11-01' AND
           NEW.DtRecl < DATE '2011-12-01 ') THEN
        INSERT INTO rec_nov VALUES (NEW. *);
```

```
ELSIF (NEW.DtRecl >= DATE  '2011-12-01 ' AND
          NEW.DtRecl < DATE '2011-01-01') THEN
       INSERT INTO rec_dez VALUES (NEW. *);


   ELSE
      RAISE EXCEPTION 'Date out of range.  Fix the venda_insert_trigger() function!';
   END IF;
   RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```